Auto-scaling Web Applications in Clouds: A Cost-Aware Approach

Mohammad Sadegh Aslanpour[a, *], Mostafa Ghobaei-Arani[b, *], Adel Nadjaran Toosi[c]

[a] *Department of Computer Engineering, Jahrom Branch, Islamic Azad University, Jahrom, Iran*
[b] *Department of Computer Engineering, Qom Branch, Islamic Azad University, Qom, Iran*
[c] *Cloud Computing and Distributed Systems Laboratory, School of Computing and Information Systems, The University of Melbourne, Australia*

Abstract

The elasticity feature of cloud computing and its pay-per-use pricing entice application providers to use cloud application hosting. One of the most valuable methods, an application provider can use in order to reduce costs is resource auto-scaling. Resource auto-scaling for the purpose of preventing resource over-provisioning or under-provisioning is a widely investigated topic in cloud environments. The Auto-scaling process is often implemented based on the four phases of MAPE loop: Monitoring (M), Analysis (A), Planning (P) and Execution (E). Hence, researchers seek to improve the performance of this mechanism with different solutions for each phase. However, the solutions in this area are generally focused on the improvement of the performance in the three phases of the monitoring, analysis, and planning, while the execution phase is considered less often. This paper provides a cost saving super professional executor which shows the importance and effectiveness of this phase of the controlling cycle. Unlike common executors, the proposed solution executes scale-down commands via aware selection of surplus virtual machines; moreover, with its novel features, surplus virtual machines are kept quarantined for the rest of their billing period in order to maximize the cost efficiency. Simulation results show that the proposed executor reduces the cost of renting virtual machines by 7% while improves the final service level agreement of the application provider and controls the mechanism's oscillation in decision-making.

*Keywords***:** Auto-Scaling, Resource Provisioning, Cloud Resource, Cost-Aware, Web Application, Service Level Agreement (SLA)

1. Introduction

With the rapid development of cloud computing, nowadays, instead of purchasing computing infrastructure, many application providers (APs) tend to host their applications on cloud resources offered by cloud providers (CPs). Cloud providers such as Amazon EC2 [1] offer resources to the AP in the form of Virtual Machines (VMs) with the scalability feature and pay-per-use charging model [2-4]. The AP, the application, and application users can be a webmaster, online store website, and end users, respectively.

Since the AP, in particular, the Web application provider is aware of the dynamics of the Web environment and end users requests, static resource provisioning is not efficient. The reason is that in static resource provisioning, with increased rate of incoming user requests, resource under-provisioning occurs which consequently results in interruption or delayed response to user requests. On the other hand, in the period of reduced traffic, the issue of resource over-provisioning occurs and as a result increased AP costs arise [4, 5]. Therefore, considering the various pricing models in the cloud [4, 6], the AP usually prepays a minimum number of resources for its permanent and long-term needs to receive a discount for this type of rental (for example, reserved instances in EC2 receive a discount of up to 75%). Consequently, with load fluctuations, the AP seeks to use the short-term rental model to cover its temporary needs (for example on-demand machines in the form of pay per hourly use). However, this approach is not enough as it requires a mechanism capable of automatically determining the capacity and the number of rented on-demand resources proportional to the incoming load [7].

Presentation of an efficient auto-scaling mechanism is a research topic which is mainly faced with the challenge of maintaining a balance between cost reduction and the Service Level Agreement (SLA). IBM proposes a model for autonomous management of auto-scaling mechanism in the form of MAPE (Monitor-Analyze-Plan-Execute) loop as a reference model [8]. The MAPE loop model can be applied to implement a cloud web application system which knows its state and reacts to its changes. Therefore, the majority of auto-scaling mechanisms are based on the MAPE loop [2, 4, 9-11]. MAPE-based mechanisms constantly repeat the four general processes of the monitoring, analysis, planning, and execution, in a way that a monitor iteratively gathers information about resources, for example, the status of resource utilization. After monitoring, the auto-scaling mechanism indicates the analysis process [4] to start which can

* Corresponding author.
E-mail address: (M. S. Aslanpour)aslanpour.sadegh@gmail.com; (M. Ghobaei-Arani)mostafaghobaye@yahoo.com, m.ghobaei@qom-iau.ac.ir

be simple or complicated; simple analysis is the use of raw information obtained by the monitor, while complex analysis discovers knowledge from information using methods such as artificial intelligence or machine learning [7, 12]. Afterward, by matching obtained analyses to a series of rules predefined by the AP, the planner makes scale-up or down decisions (rule-based planner [4]). The final phase of the MAPE cycle is the execution of the decision by the executor. This is when the auto-scaling mechanism needs to send a request for instantiation of a new VM or release of the one of the VMs previously rented from the CP. This research focuses on improving the performance of the executor in the resource auto-scaling mechanism with a cost-aware approach.

The motivation behind the improvement of the executor's performance lies in the following: Thanks to the possibility of selecting different types of VMs with various capacities, APs can rent a large number of VMs of different types simultaneously; considering the intense workload fluctuation in the Web environment, this is highly possible to happen [13]. That said, if the auto-scaling mechanism makes a scale-down decision, the executor needs to select from a diverse set of rented VMs and release one. The basic question posed here is whether it matters which VM is selected? If the answer is yes, what policy is the best to be used for this selection? Unlike Amazon's auto-scaling policy that always selects the oldest VM for release (as default executor) and according to the dark spots seen in related research [10, 14-23], this selection should be made cautiously and rigorously. This is because, firstly, the CP calculates partial billing as full billing (billing cycle) [24, 25]. For example, in the EC2 service, billing is carried out on an hourly basis and the release of a VM for a duration of 4 hours and 1 minute would result in billing for 5 hours. Therefore, policy making for minimizing the minutes wasted in the release of surplus VMs is an important economic matter for the AP. Secondly, due to unresolved load balancing challenges [26], candidate VMs are probably processing different workloads and the influence of releasing each VM on the SLA would vary. Hence, the first purpose of the present research is to employ novel policies, especially cost saving ones, in the selection of surplus VMs (professional executor).

A research gap can be still seen after the selection of the surplus VM and before its release. On the one hand, it is likely that the selector did not manage to find a VM with exactly X hours of use. In this situation, the release of that VM would impose extra costs. On the other hand, a scale-up decision may be made immediately after the release of the surplus VM; in this case, the delayed startup of the new VM is a challenge which negatively affects SLA [2-4]. Due to the unpredictability of the Web environment [27, 28] or maladjustment of scaling rules [4], it is highly likely that the mechanism to be affected by contradictory actions when the mechanism is in an oscillation condition [2, 4]. As a result, the following hypothesis is put forward: If the selected surplus VM stays rented by the AP until the last minute of the bill, it can possibly be used for the improvement of the scaling mechanism's performance. Therefore, the other goal of this research is to offer an executor with the ability to quarantine the surplus VM until the billed hour is completed in order to resolve the challenge of delayed VM startup (super professional executor - Suprex). This is while to date, researchers merely considered benefits of vertical scaling or applying cooling time in the execution of the commands as the method for overcoming this challenge [4].

This paper presents a scaling mechanism equipped with a super professional executor (Suprex) with a cost-aware approach. We seek to show that the execution phase of the MAPE cycle can play an effective role in cost saving. We explain all four phases as they are required for the full implementation of the auto-scaling mechanism. This is also required for better understanding of the paper. The auto-scaling mechanism offered in this research is different from others where the focus is mainly on improving the mechanism's performance in the monitoring, analysis, and planning phases rather than the execution phase. The reason why the execution phase was overlooked lies in the fact the actions are often considered under the control of the CP and the CP is considered as a black box. However, by applying an architecture with full control [29] from the AP's perspective, the power is granted to the AP in the execution of all scaling commands. The **main contributions** of this research are as follows:

- We designed an auto-scaling mechanism based on the MAPE concept for Web applications,
- We enhanced the effectiveness of the execution phase of the control MAPE loop with a cost-aware approach,
- We provided an innovative solution for overcoming the challenges of delayed VM startup,
- We designed an executor in order to mitigate oscillation and increase the stability of the mechanism, and
- We conducted a series of experiments to evaluate the performance of proposed approach under real-world workload traces for different metrics.

The rest of the article is organized as follows: Section 2 provides the necessary background. Section 3 includes related work; Section 4 fully explains the proposed approach. Section 5 simulates and evaluates the performance of the Suprex executor. Finally, Section 6 presents conclusions and future work.

2. Background

This section provides a brief overview of autonomic computing and application adaptation.

2.1. Autonomic Computing

The increasing complexity of computer systems makes it essential to handle them autonomically. IBM’s Autonomic Computing Initiative has helped to define the four properties of autonomic systems: self-configuration, self-optimization, self-healing, and self-protection [8]. Cloud computing providers manage their data centers in an efficient way, taking cues from well-established autonomic computing practices. Particularly, tasks like VM provisioning, disaster recovery, capacity management, etc. are performed autonomically [30]. To effectively manage cloud deployed web applications, we need to react to regularly occurring or anticipated events in the system [31]. In [8], IBM proposes a model for autonomous management in the form of an autonomic MAPE-K loop as a reference model. The MAPE-K loop model can be applied to implement a cloud web application system which knows its state and reacts to changes in it. This model details different components that allow an autonomic manager to self-manage properties, namely Monitor, Analyze, Plan, Execute and Knowledge [32]. The MAPE-K loop model is depicted in Fig.1 and discussed in the remaning part of this section.

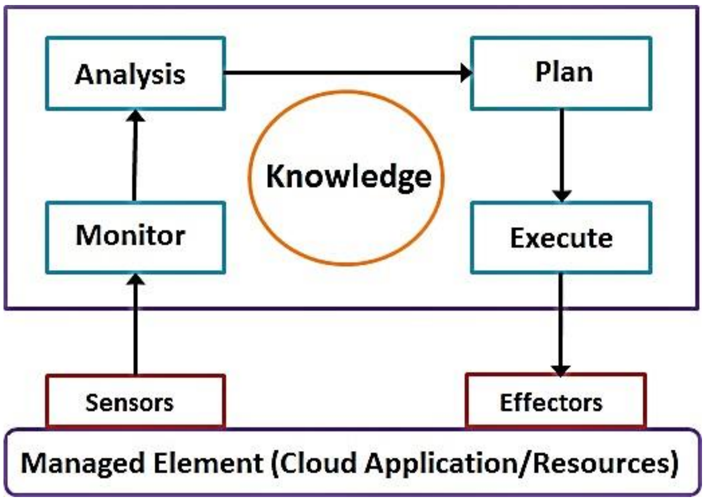


Fig. 1: The control MAPE-K loop.

The MAPE-K loop model is applied to auto-scaling of web applications in the cloud environment. The auto-scaling process of web applications matches the MAPE-K loop model, which dynamically adapt the resources assigned to the web applications, depending on the input workload [4, 33]. The autonomic manager interacts with the managed element (e.g., web application resources) through two interfaces that are the sensors and effectors to supervise and act on the system, respectively

The MAPE-K loop model consists of four phases: Monitoring (M), Analysis (A), Planning (P) and Execution (E). Monitoring phase involves capturing properties of the managed element that can be software or hardware components used to perform monitoring; they are called sensors. The monitor component is responsible for collecting information about the metrics of the low-level metrics (e.g., CPU utilization, memory usage, and network traffic, etc.) and high-level metrics (e.g., rate of request arrival, type of requests, size of requests, etc.) during the monitoring phase. These different sets of monitoring parameters are stored in a knowledge base for use by other components. The knowledge base in the MAPE-K loop model shares data between the actual MAPE phases. The analysis phase is responsible for processing the information gathered directly from the monitoring phase. During the analysis phase, the system determines whether it is necessary to perform scaling actions based on the monitored information. The planning phase is responsible for the estimation of the total number of resources to be provisioned/de-provisioned in the coming scaling action. It follows rules that can be as simple as an event-condition-action policy, which is easy to implement and fast to compute, or take the form of utility functions, which try to optimize a given property of the managed systems. The execution phase is responsible for executing the actions decided in the planning phase. It can be implemented by the automation APIs (i.e., effectors) available for the environment and the runtime configurability of the cloud web application.

2.2. Application Adaptation

The migration of existing applications to the cloud environment requires adapting them to a new computing paradigm [34]. Many applications are not ready to be moved to cloud because the environment is not mature enough for this type of applications [35, 36]. The focus of most existing works is on migration of the entire application to the cloud environment based on virtualization technology and using of VMs as the means for migration and running it in the cloud environment. This way, the adaptation of the application is limited to the method that the application manages its resources [34]. Application adaptation in this domain is designated to how to manage a dynamic amount of VMs in trade-off with the cost of resources. Nowadays, the cloud web applications are often deployed and executed on a cloud infrastructure which provides a convenient on-demand approach for renting resources and easy-to-use horizontal scaling capabilities. The workload of cloud web applications is continuously changing over time and unexpected peaks of requests can happen which makes the system incapable of responding to these unpredicted changes. For this reason, the autonomic adaptation is an emerging solution to automatically adapt the resources allocated to the application according to the incoming traffic, CPU-utilization, and other metrics [37].

In this paper, we apply the MAPE-K loop model to the autonomic adaptation of cloud web applications. A concrete example of automatic infrastructure management is Amazon's Auto Scaling [38], which manages when and how an application's resources should be dynamically increased or decreased. Furthermore, this paper focuses specifically on the Execution phase. In other words, the system is monitored, analyzed and the adaptation plan is produced that is executed by a super professional executor (Suprex) with a cost-aware approach.

3. Related Works

This section is an overview of related works in the field of auto-scaling for web applications. It is centered on the studies focused on each MAPE phase and applied techniques. Note that research is considered to be focused on the analysis phase if it benefits from complex analysis, e.g., neural networks; research is considered to be focused on the planning phase if planning regarding the capacity and the number of resources is conducted based on several scaling indicators. In the analysis phase, the analysis is carried out with reactive or proactive policies. In the planning phase, the decisions are made by two categories of scaling indicators: (1) low-level scaling indicators, at the hypervisor/physical level; (2) high-level scaling indicators, which are related to the application criteria. For scaling decision-making in the planning phase, different techniques such as rule-based decisions, analytical modelling, and machine learning are often used for resource estimation. Moreover, in the execution phase, scaling is done with horizontal and vertical methods which work with replication-based and resizing-based techniques, respectively.

3.1. Works focused on the analysis phase of MAPE

Focusing on the analysis of resource utilization using the neural network and linear regression, Islam *et al.*[14] studied the issue of resource provisioning. Huang *et al.* [15] also paid attention to proactive analysis of scaling indicators and took advantage of Double Exponential Smoothing to predict resource utilization. Bankole *et al.* [16] and Ajila *et al.*[17] focused on the analysis of indicators with several neural networks. Moreover, other than utilization, they considered the scaling indicators of throughput and response time. In another work, Kaur and Chana [39] sought to focus on analysis and planning phases (with more effort in the analysis phase). They offered a complete analysis of effective indicators and conducted resource capacity planning using a decision tree. Mohamed *et al.* [9] investigated a MAPE cycle-based scaling mechanism with a purely reactive policy using the scaling indicator of the response time. Because their method is reactive, it faces challenges at the time of reconfiguration. Herbst *et al.* [18] analyzed the workload in a way that in each monitoring stage, a prediction method is selected with the least errors (proactive policy); the categorization of prediction methods from simple to complex is an important contribution of this study. In a different work, with a focus on the analysis of workload properties, Singh and Chana [21] conducted a study on workload clustering and its distribution between resources considering the QoS features of each workload. Assunção *et al.* [23] analyzed user patience on auto-scaling with a proactive policy. Toosi *et al*. [40] propose a reactive auto-scaling mechanism working based on threshold-based rules to help their load balancing approach for utilization of renewable energy.

3.2. Works focused on the planning phase of MAPE

Casalicchio and Silvestri [29] focused on both analysis and planning phases (with more focus on the planning phase) and studied the performance of different architectures by presenting 5 scaling rules (rule-based resource estimation). García *et al.* [20] tried to improve the planner to reduce the cost and maximize QoS with a reactive SLA-aware policy. Qavami *et al.* [19] also tried to improve the planner, but they used a resource-aware approach utilizing machine learning. Focusing on the analysis of resource capacity, Fallah *et al.* [22] used learning automata and conducted resource planning with a reactive and proactive method. Beltrán [41] considered planning for vertical and horizontal scaling; he provided an automated resource provisioning framework for multi-tier cloud applications with an approach to the realization of user needs and reducing AP costs. Moltó *et al.* [42] conducted scaling planning with the possibility of horizontal and vertical scaling. They improved planning using oversubscription and live migration techniques. In another effort, Ghobaei-Arani *et al.* [10] provided a resource provisioning framework based on the MAPE concept; their focus was on the planning phase where they predicted the future of resource utilization with the help of a learning automata. Aslanpour and Dashti [43, 44] sought to improve the mechanism's performance with two-sided planning based on resource utilization and SLA status with a combined reactive and proactive policy. Lastly, Moldovan *et al.* [24] analyzed cost and cost-aware decision-making for releasing surplus VMs based on cost saving, while none of the other indicators effective in decision-making, such as resource utilization or user response time were considered.

In general, the majority of research investigate scaling indicators [14-18, 21-23], most others improve the planner [9, 10, 19, 20, 41-43], and some improve both phases [29, 39]. Since the monitoring phase is merely the collection of information for other steps, it cannot be considered as a research focus. Anyway, solutions are rarely provided for the improvement of the executor. Casalicchio and Silvestri [29] were the only ones who considers the execution of scale-down commands according to the status of the last hour. Moltó *et al.* [42] executed scale-up commands using the unused capacity of other VMs with an oversubscription policy. In studies by Aslanpour and Dashti [43], Aslanpour and Dashti [44], and Moldovan *et al.* [24], scale-down decisions are executed by selecting the surplus VM with Last Up First Down (LUFD or the youngest VM), First Up First Down (FUFD or the oldest VM), and cost-aware methods, respectively. Table 1 shows a summary of related works according to their proposed approaches, the extent of their focus on the four phases of MAPE, and used techniques.

Table 1: Overview of related works in the field of auto-scaling

Study (Research)	**Approach**	**Focused Phase**				**Technique**			
		Monitoring	**Analysis**	**Planning**	**Execution**	**Policy**	**Scaling Indicator**	**Resource Estimation**	**Scaling Method**
Islam *et al.* [14]	Load prediction		✓			Proactive	Low level	Machine learning	Horizontal
Huang *et al.* [15]	Resource-aware		✓			Proactive	Low level	Machine learning	Horizontal
Bankole and Ajila [16]	Resource-aware		✓			Proactive	Hybrid	Machine learning	Horizontal
Ajila and Bankole [17]	Resource-aware		✓			Proactive	Hybrid	Machine learning	Horizontal
Casalicchio and Silvestri [29]	Resource-aware			✓		Reactive	Hybrid	Rule-based	Horizontal
Mohamed *et al.* [9]	SLA-aware			✓		Reactive	High level	Rule-based	Horizontal
Herbst *et al.* [18]	Load prediction		✓			Proactive	High level	Rule-based	Horizontal
García *et al.* [20]	SLA-aware			✓		Reactive	High level	Rule-based	Horizontal
Qavami *et al.* [19]	Resource-aware			✓		Proactive	Low level	Machine learning	Horizontal
Singh and Chana [21]	SLA-aware		✓			Proactive	Hybrid	Analytical modelling	Horizontal
Fallah *et al.* [22]	Resource-aware			✓		Hybrid	Low level	Machine learning	Horizontal
Kaur and Chana [39]	SLA-aware		✓			Hybrid	Hybrid	Analytical modelling	Horizontal
Beltrán [41]	Resource-aware			✓		Reactive	High level	Analytical modelling	Hybrid
Ghobaei-Arani *et al.* [10]	Resource-aware			✓		Proactive	Low level	Machine learning	Horizontal
Assunção *et al.* [23]	Load prediction		✓			Proactive	Hybrid	Rule-based	Horizontal
Moldovan *et al.* [24]	Cost-aware		✓			Proactive	Low level	Rule-based	Horizontal
Moltó *et al.* [42]	Resource-aware			✓		Reactive	Low level	Analytical modelling	Hybrid
Aslanpour and Dashti [43, 44]	SLA-aware			✓		Hybrid	Hybrid	Rule-based	Horizontal
Toosi *et al.*[40]	Resource-aware		✓			Reactive	Low level	Rule-based	Horizontal
Proposed	**Cost-aware**				✓	**Hybrid**	**Hybrid**	**Rule-based**	**Horizontal**

4. Proposed approach

This section details the proposed scaling mechanism equipped with a Suprex executor. This mechanism conducts its operation based on the MAPE concept with an approach to cost-saving and exploitation of surplus resources. Afterward, the problem formulation is explained and the algorithm needed for the implementation of the proposed mechanism is presented.

4.1. Auto-scaling mechanism

An overview architecture of cloud-based web applications can be seen in Fig. 2. In this architecture, the three entities of the end user, AP, and CP interact with each other using a top-down strategy. The end users send their request to the Web application via devices connected to the Internet. The AP forwards the incoming request to the VM provisioned for the application tier via the load balancer. The web application has a three-tier architecture and an independent VM is generally provisioned for each tier. User requests can have access to different tiers for response. After processing, VMs deliver the appropriate response to the user. The AP has an auto-scaling mechanism by which it determines the number of required VMs considering the fluctuation in the incoming load. The contribution of this research lies in the improvement of the auto-scaling mechanism in the execution phase. CP is in the lowest layer of the cloud environment architecture and provides the AP with computational resources in the form of VMs with the pay per use charging model.

Cloud Resource Auto-scaling is constantly possible within four phases of MAPE (see Fig. 2). If the planner's interpretation of the analyses is resource under-provisioning, the executor acts on adding a new VM. On the other hand, if the planner's interpretation of the analyses is resource over-provisioning, the executor acts on releasing a VM by selecting a VM from on-demand VMs and releasing it. This section continues with the explanation of the innovative aspects of the proposed mechanism - the improvement of the execution of scaling commands by Suprex.

In order to differentiate the novel features of Suprex, the common (default) executor in auto-scaling mechanisms is first shown. Afterward, the aware surplus VM selection feature in the execution of scale-down commands is added to it for cost saving purposes (professional executor). Finally, the proposed Suprex executor is completed by adding the surplus VM quarantine feature in order to overcome the challenge of delayed startup of new VMs.

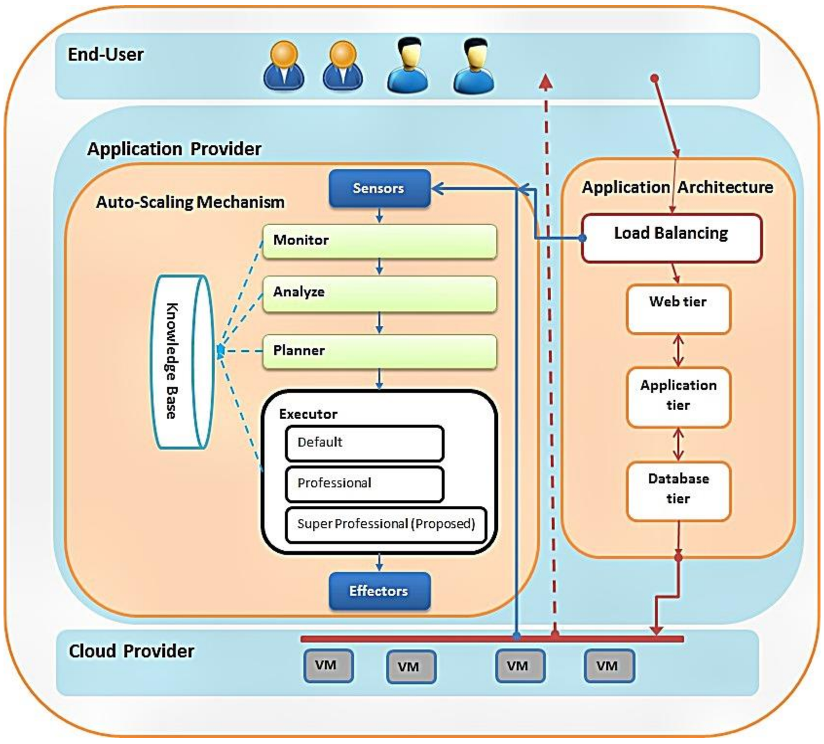


Fig. 2: An overview architecture of cloud-based web applications

4.1.1. Default executor

The common executor in research adopts the scaling commands from the planner without applying any specific policy (see Fig. 3a). However, the most important drawback of this approach is that selection of surplus VMs for release happens without specific strategy; that is, in the Default executor, the selection is conducted randomly or in a better case, with LUFD [43] or FUFD [44] policies.

4.1.2. Professional executor

Through an aware selection of surplus VMs while executing scale-down commands, the Professional executor fills the gap in the behavior of the Default executor. Thus, as it can be seen in Fig. 3b, this executor benefits from an internal part called *surplusVMSelector* for aware selection of surplus VMs. This is important from two aspects:

- Since CPs consider partial billing cycles for rented VMs as a full billing cycle [24], why not select the VM with greater use of its last billing cycle? This fact and the attempt to minimize wasted minutes lays the groundwork for providing a *cost-aware* surplus VM selection policy.
- There are VMs with unequal loads, the release of each has a different impact on the SLA. This is inevitable for two reasons: firstly, load balancing challenges in the cloud [26], and secondly three-tiers architecture of Web applications which requires a different service time for each tier [4, 39]. This issue lays the groundwork for the *load-aware* surplus VM selection policy.

Therefore, aware selection of surplus VMs is realized with *cost-aware* and *load-aware* policies in the proposed Professional executor.

4.1.3. Super Professional Executor

In the surplus VM selection policy, the Professional executor aims to select the VM with the greatest utilization of the last hour of use. Since it is possible that none of the candidate VMs have exactly X hours of use (according to the Amazon's pattern) [24], waste of cost is still possible for the AP. The solution for this issue and turning it into an opportunity is to quarantine the surplus VM for the AP until the invoice period is completed (see Fig. 3c). Therefore, other than applying specific policies in surplus VM selection, Suprex sends surplus VMs to quarantine (rather than immediate release).The advantage of quarantining surplus VMs is that if, for example, the elapsed time of renting the VM is 5 hours and 20 minutes, by quarantining it during the remaining 40 minutes before the release deadline, it can be utilized if the planner makes a scale-up decision during this 40 minutes (due to the intensity of the incoming load, maladjusted rules, or contradictory actions in the mechanism), this VM can be restored and utilized.

In order to manage quarantined VMs, Suprex requires an updater to check the release deadline of quarantined VMs every minute and release them on the due time. The updater is placed at the end of the MAPE cycle so that if scale-up decisions and the release deadline of a VM coincide, the VM is restored before it is released (Fig. 3c). This can prevent SLA violation caused by the startup of a new VM through a timely restore of the quarantined VM. Thus, Suprex overcomes the challenge of delayed startup of a new VM.

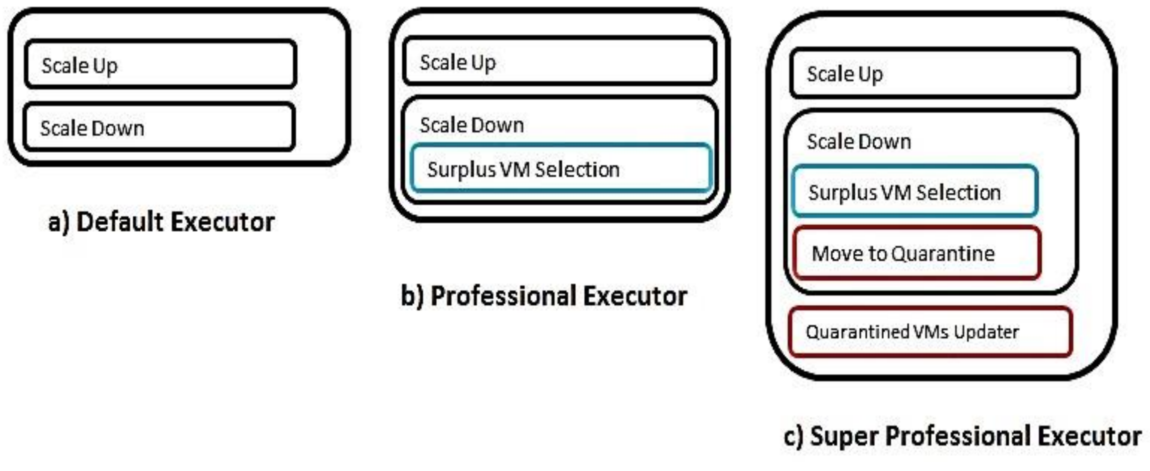


Fig. 3: The executing component in the MAPE-based auto-scaling mechanism; part a) is the Default executor, part b) is the Professional executor, and part c) is the Super Professional executor

How and why are surplus VMs quarantined? In the EC2 service, VMs experience different statuses throughout their life cycle (see Fig. 4). They are first requested by the AP, and then put in the *Pending* (*Startup*) status. Depending on the time of the day, the number of requested VMs, operating system, VM size, and the server's geographical location, this status can take up several minutes [45]. After *Startup*, the VM is put in the *InService* status and is able to receive the load sent by the load balancer and process it; this means that from this time onwards, the AP can leave end user requests to the VM for processing. By default, if the AP intends to quarantine the VM in a way that it is out of reach of the load balance, it has to change its status to *Stopped* or *Standby*. Nonetheless, restoring the VM from these statuses to the *InService* status is faced with difficulties such as:

(1) Restoring VMs from *Standby* and *Stopped* statuses requires going through the delayed startup.
(2) When a stopped VM wants to be restored, it is likely that CP uses another available host for the resource allocation. This can cause more delay in the VM startup.
(3) Restoring the VM from the *Stopped* status results in billing for the AP.
(4) If the VM in the *Stopped* status becomes unhealthy, this problem remains hidden from Amazon until the AP requests switching the VM status to *InService*. In this condition, Amazon starts an alternative VM for the AP that leads to the deletion of the data on it which would probably result in further delay in VM startup.

Therefore, the mechanism is faced with the issue of delayed startup after switching to *Standby* and *Stopped* statuses (Fig. 4). These problems encouraged us to propose a new status in the system: *Quarantined*. VMs in the *Quarantined* status are in operational conditions, but they are free from workload (Fig. 4). Since switching from *InService* to *Quarantined* status does not make the VM inaccessible for the AP, it would still be possible for the AP to continue to manage it.

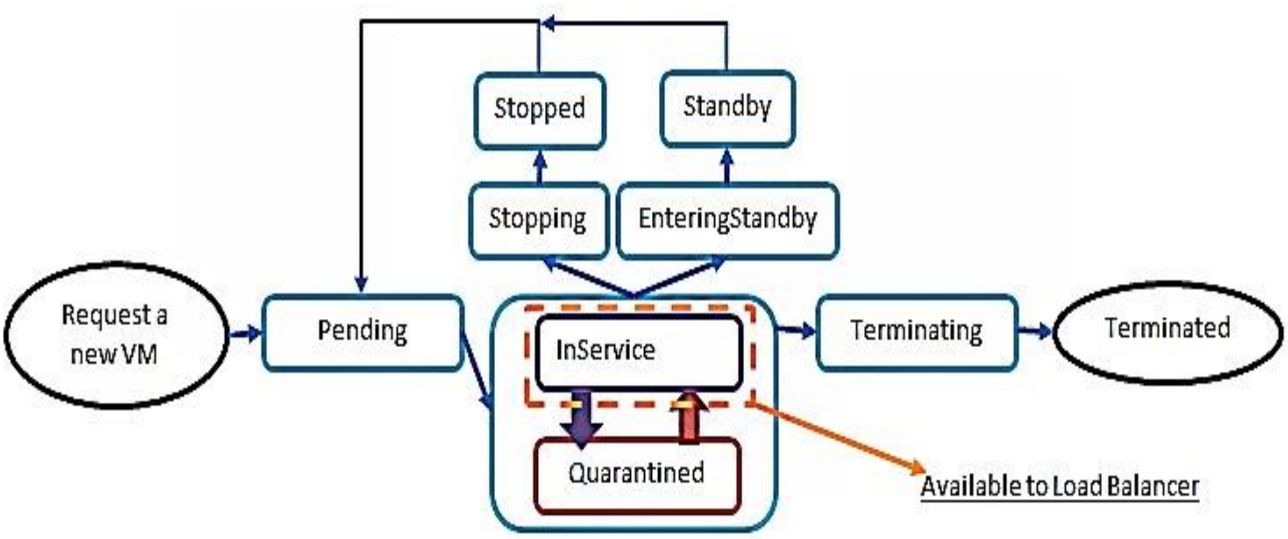


Fig. 4: The Life Cycle of a virtual machine after adding the quarantine status

4.2. Problem formulation

This section explains the notations used in our proposed approach (see Table 2). Rented VMs are divided into three categories: Requested VMs that have not been started (PL), In-service VMs (SL) and VMs switched to the Quarantined status by Suprex (QL). In addition, all VMs under the rental of the AP are in the AL list (list of all VMs) (see (1)).

$$AL = PL + SL + QL \quad (1)$$

The auto-scaling mechanism is regularly run every minute for the monitoring phase while it is run at specific time intervals (Δt) for analysis, planning, and execution phases (except for the updater component of the executor). In the monitoring phase, the scaling indicators of resource utilization (M_i^u) and response time (M_i^{rt}) are measured every minute (*i*). The mean resource utilization at time *i* is calculated based on (2).

$$M_i^u = \frac{\sum_{j=1}^{AL} VM_j \ Utilization}{AL} \quad (2)$$

In this equation, the utilization of all VMs is calculated and divided by the total number of all rented VMs (AL). Equation (3) is also used to calculate the utilization of each VM separately at time *i*.

$$VM_i \ Utilization = \frac{\sum_{j=1}^{CurrentCloudlets}(CloudletPEs_j * CloudletLength_j)}{PEs * MIPS} \quad (3)$$

where $CurrentCloudlets$ is the number of requests (tasks) being executed by the VM, $CloudletPEs$ is the number of processors required for a request, and $CloudletLength$ is the number of instructions of a running request. Moreover, PEs is the number of the VM's processing elements and $MIPS$ (Million Instructions Per Second) is the processing power of each element.

On the other hand, the mean response time in minute i is monitored and stored according to (4). Here, the response time means the latency in the response to user requests.

$$M_i^{rt} = \frac{\sum_{j=1}^{Req^{answered}} W_j}{Req^{answered}} \tag{4}$$

The total waiting time W_j of answered requests ($Req^{answered}$) in minute i is measured and then divided by the number of answered requests. The waiting time of each request is calculated according to (5).

$$W_{\mathrm{J}} = \left(FinishTime_j - ArrivalTime_j\right) - ServiceTime_j \tag{5}$$

Here, $ArrivalTime_j$ and $FinishTime_j$ are the times the request is received from user i and the time the response is received by its VM, respectively. $ServiceTime_j$ is also the estimated service time required by the incoming user request which is calculated according to (6). If the time between $ArrivalTime_j$ and $FinishTime_j$ is larger than $ServiceTime_j$, there has been latency in the response to the user request.

$$ServiceTime = \frac{cloudletLength * cloudletPEs}{VM.getNumberOfPEs * Vm.getMIPS} \tag{6}$$

In the monitoring phase, other parameters are determined for the measurement of performance criteria; such as T as the throughput which, according to (7), shows the ratio of answered requests ($Req_i^{answered}$) to received ones ($Req_i^{received}$) during minute i.

$$T = \frac{Req_i^{answered}}{Req_i^{received}} \tag{7}$$

Requests experiencing the response time (W) of more than the desired limit (DRT) are considered as SLA violations. According to (8), the sum of these seconds in hours (here '%' is the modulus operator) is considered as the hours of SLA violation ($SLAV^{hour}$).

$$SLAV^{hour} = Sum\ SLA\ Violation\ (in\ hour) = \left(\sum_{j=1}^{totalUserRequests} W_j - DRT\right) \%\ anHour \tag{8}$$

In the analysis phase, the analyzer takes actions to analyze the scaling indicators (resource utilization and response time), the result of which is presented in the form of A_i^u and A_i^{rt}. Afterward, the planner makes the scaling decision (D) by matching the utilization analysis (A_i^u) with the upper threshold (U^{up_thr}) and lower threshold (U^{low_thr}) as well as the analysis of response time (A_i^{rt}) with the upper threshold (RT^{up_thr}) and lower threshold (RT^{low_thr}). The planner's decision is sent to the executor and could be *ScaleUp*, *ScaleDown*, or *DoNothing*.

In the execution of the command, the executor is connected to a few variables including: *MaxVM* which is the maximum number of VMs allowed to be rented from the CP. *MaxVM* is used to avoid the imposition of heavy costs on the AP in conditions of confusion in the mechanism. The variable *SU*, according to the time recorded for the startup of a VM, shows whether or not it had been started before (Started VMs have the value of -1). *SU* is checked when restoring the VM from QL to SL. The α variable determines the surplus VM selection policy in the execution of scale-down decisions which can be *FUFD*, *LUFD*, *load-aware*, or *cost-aware*. Moreover, some of these policies may need the elapsed time from the last billing cycle in their selection, and the variable *PT* shows this period as one of the attributes of any VM. Finally, *MinPT* and *MaxPT* show the minimum and maximum elapsed times since the last hour of use among VMs, respectively.

Table 2: Summary of the notations used in this article

Symbol	Description	Related to
PL	List of pending virtual machines	Rented Resources
SL	List of InService virtual machines	
QL	List of quarantined virtual machines	
AL	List of all virtual machines = (PL + SL + QL)	
Δt	Scaling Interval	Calling Analyzer, Planner, and Executor
$Clock$	Simulation timer	
M_i^u	Monitored Average Resources utilization (percent) at time *i*	Monitor
W	Waiting time of a user request	
M_i^{rt}	Monitored Average response time during time *i*	
$Req_i^{received}$	The number of received user requests during time *i*	Performance Metric
$Req_i^{answered}$	The number of answered user requests during time *i*	
T	Resource throughput in last minute	
DRT	Desire (acceptable or justified) response time = SLA contract	
$SLAV^{hour}$	The total response times of more than *DRT* (in hour) for all requests during the experiment	
A_i^u	Analyzed CPU Utilization at time *i* (based on history)	Analyzer
A_i^{rt}	Analyzed Response Time during time *i* (based on history)	
U^{up_thr}	CPU utilization threshold for under-provisioning (Upper)	Planner
U^{low_thr}	CPU utilization threshold for over-provisioning (Lower)	
RT^{up_thr}	Response time threshold for under-provisioning (Upper)	
RT^{low_thr}	Response time threshold for over-provisioning (Lower)	
D	Planner Decision (Scale-Up, Scale-Down, and Do Nothing)	
SU	Shows that a virtual machine has passed the startup delay (-1 = no and others = yes)	Executor
$maxVM$	Maximum number of on-demand virtual machines the executor allows for renting (Scaling Limitation)	
α	Surplus VM selection policy (FUFD, LUFD, *Load-aware*, and *Cost-aware*)	
PT	Passed time from last hour (for a VM time)	
$MinPT$	Minimum passed minute from last hour (for a VM)	
$MaxPT$	Maximum passed minute from last hour (for a VM)	

4.3. Proposed algorithm

Although the main focus of the proposed approach is the execution phase, in this section, we describe other algorithms required to fully implement the auto-scaling mechanism. All of the operations of the auto-scaling mechanism are controlled by Algorithm 1 (see Fig. 5), which manages the MAPE cycle. Algorithm 1 is responsible for controlling the chronological order upon calling each MAPE phase, each of which is implemented with an independent algorithm. Fig. 5 shows how these algorithms work with each other and the chronological order of their execution.

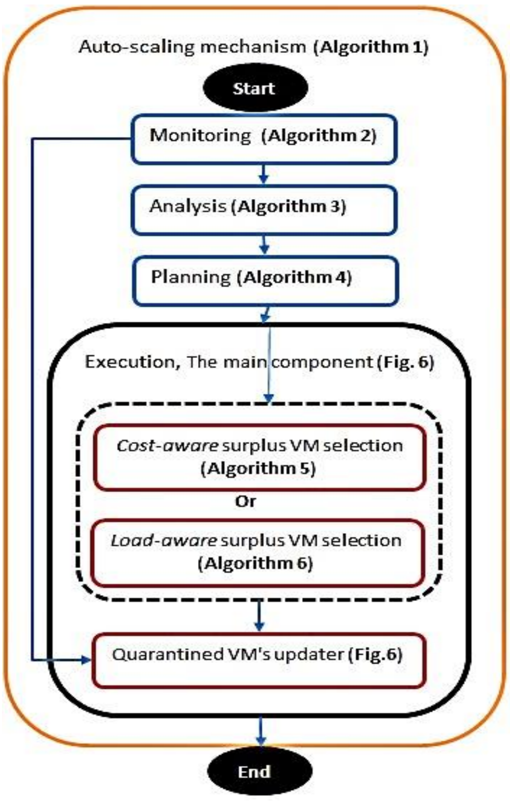

Auto-scaling mechanism (Algorithm 1)
Start
Monitoring (Algorithm 2)
Analysis (Algorithm 3)
Planning (Algorithm 4)
Execution, The main component (Fig. 6)
Cost-aware surplus VM selection (Algorithm 5)
Or
Load-aware surplus VM selection (Algorithm 6)
Quarantined VM's updater (Fig.6)
End

Fig. 5: Algorithms used to implement the proposed auto-scaling mechanism.

Algorithm 1 shows an overview of the algorithm's operations in a way that it carries out monitoring (line 2) continuously (every minute) while performs analysis, g, and execution (lines 4 to 6) at specified intervals (Δt). Therefore, every time Clock % Δt = 0, it is the turn to perform analysis, planning, and execution. Similar to monitoring, the updating component of quarantined VMs is recalled every minute (line 8). This section provides a detailed explanation of the algorithm for every MAPE phase.

Algorithm 1: Auto-scaling mechanism (Main)

```
1. repeat every 1 minute (while there is an end user request)
2.     Monitoring (); // Stores history of metrics
3.     if Clock % $\Delta t$ = 0 then
4.         Analysis (history of $M^{u}$, history of $M^{rt}$);
5.         Planning ($A^{u}$, $A^{rt}$);
6.         Execution ($D$); // Main component of Suprex
7.     end if
8.     Update Quarantined VMs // Suprex's Surplus VM updater component
9. end repeat
```

4.3.1. Monitoring phase

According to Algorithm 2, the monitor continuously collects and stores information about low-level parameters (infrastructure level) and high-level parameters (application level) [4] during the last minute. Low-level parameters include utilization and the number of resources (line 2), and high-level parameters include categories related to the end user and the SLA, lines 4 and 5, respectively.

Algorithm 2: The monitoring phase of the MAPE cycle

```
1. /* Low Level Parameters */
2. Store $M_i^{u}$, PL, SL, QL, AL // VM parameters
3. /* High Level Parameters */
4. Store $Req_i^{received}$ // End users parameters
5. Store $M_i^{rt}$, $Req_i^{answered}$, $T$, // SLA parameters
```

4.3.2. Analysis phase

With a Hybrid method [3] consisting of proactive and reactive methods, the analyzer analyzes the two scaling indicators of resource utilization (A^u) and response time (A^{rt}), respectively (see Algorithm 3). A^u analyzes utilization during the last minutes after the previous management period (from $i - \Delta t$ to i) by calculating the weighted moving average (WMA) (lines 5 to 12). The Fibonacci numbers technique [46] is used in weighing the observations so that recent observations are given more weight compared to previous observations (line 7). The variables defined in line 3 are used for the purpose of WMA calculations. In addition, the analysis of A^{rt} is taken from the latest observation (line 14). These two analyses are sent to the planner so that it can plan the capacity control (line 15).

Algorithm 3: The analysis phase of the MAPE cycle

```
1.  Input:     $M^u$ (during the past $\Delta t$ minutes), $M^{rt}$ for the last minute
2.  Output:    $A^u$ and $A^{rt}$
3.  Variable:  double cpuUtilization ←0, weight ←0, weightedMoving ←0, totalWeight ←0, fiboNumberA ←0, fiboNumberB
4.  // Proactive section
5.  for j ←0 to j < $\Delta t$ do
6.      cpuUtilization ← $M^u_{i-j-\Delta t}$
7.      Weight ← fiboNumberA + fiboNumberB
8.      fiboNumberA ← fiboNumberB; fiboNumberB ← weight
9.      weightedMoving ← weightedMoving + (cpuUtilization × weight)
10.     totalWeight ← totalWeight + weight
11. end for
12. $A^u$ ← weightedMoving / totalWeight
13. // Reactive section
14. $A^{rt}$ ← $M^{rt}_i$
15. return $A^u$ and $A^{rt}$
```

4.3.3. Planning

The planner makes decisions with a rule-based method and step sizes [4] equivalent to one. According to Algorithm 4, the obtained analysis compares the resource utilization (A^u) and the response time (A^{rt}) with the thresholds defined for each. In the event of upper thresholds violation with both analyses (line 3), a *ScaleUp* decision is made (line 4), and if both violate the lower thresholds (line 5), a *ScaleDown* decision (line 6) is made; otherwise, *DoNothing* decision is made (line 8). In the end, the planner's decision is sent to the executor (line 10).

Algorithm 4: The planning phase of the MAPE cycle

```
1.  Input:   $A^u$, $A^{rt}$, $U^{low\_thr}$, $U^{up\_thr}$, $RT^{low\_thr}$, $RT^{up\_thr}$
2.  Output:  D
3.  if $A^u > U^{up\_thr}$ and $A^{rt} > RT^{up\_thr}$ then
4.      D = ScaleUp
5.  else if $A^u < U^{low\_thr}$ and $A^{rt} < RT^{low\_thr}$ then
6.      D = ScaleDown
7.  else
8.      D = DoNothing
9.  end if
10. return D
```

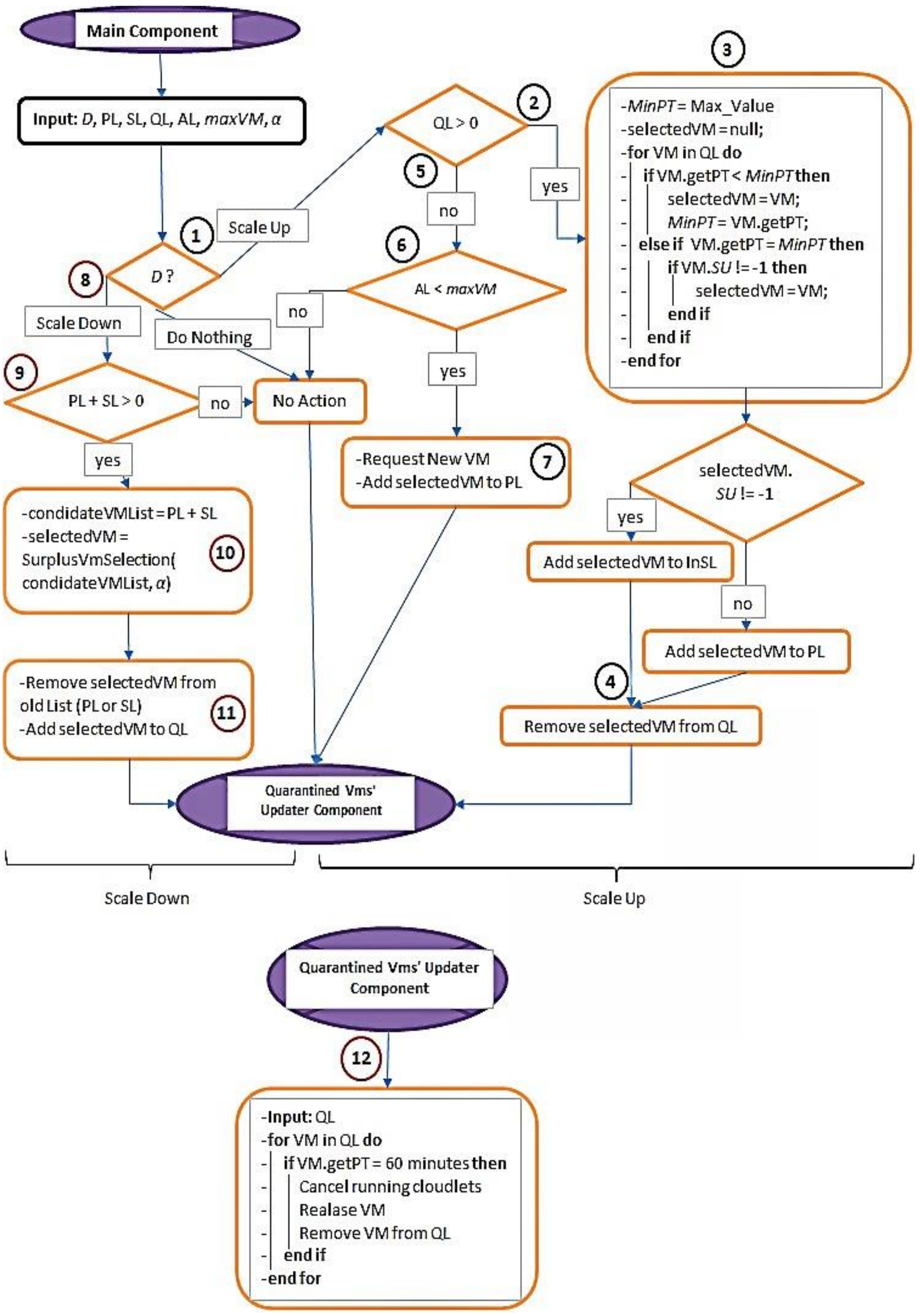
4.3.4. Execution

The methods of the proposed executor can be seen in Fig. 6. Suprex is comprised of two components: the main component and the quarantined VM updater component. The main component takes the following actions:

- Scale-up: In the event of scale-up decision (1), the restoration technique for quarantined VMs is sought. If there is a quarantined VM (2), the VM with the most remaining number of minutes to the completion of the last hour of use is selected (3). In the case of multiple VMs with equal conditions (VM.*getPT* = *MinPT*), the one in the operational status is restored (in order to avoid restoring the VM quarantined before startup). Afterward, the VM status is switched from quarantine and is moved to the list of operational VMs (SL or PL) (4). On the other hand, if there is no quarantined VM (5) and not faced with resource provisioning restrictions (6) the CP takes measures to request a new VM (7).
- Scale-down: In the case of the scale-down decision (8), if there is a candidate VM (PL + SL > 0) (9) then according to policy α, the *surplusVMselector* selects a surplus VM (10) and moves it to quarantined mode (11).

The updater component of quarantined VMs (Fig. 6) checks every minute to see if the deadline for the release (VM.*getPT* = 60 minutes) of a VM is reached so that it can be released (12). At this time, if there are Cloudlets running in the VM, the updater cancels them and sends them to the load balancer again for redistribution, as opposed to the work of Beltrán [41] in which the canceled task is left without a response (see Fig. 6)

In the rest of this section, we introduce *cost-aware* and *load-aware* policies (α) for the selection of the surplus VM by *surplusVMselector*.



```
-MinPT = Max_Value
-selectedVM = null;
-for VM in QL do
-    if VM.getPT < MinPT then
-        selectedVM = VM;
-        MinPT = VM.getPT;
-    else if VM.getPT = MinPT then
-        if VM.SU != -1 then
-            selectedVM = VM;
-        end if
-    end if
-end for
```

```
-Input: QL
-for VM in QL do
-    if VM.getPT = 60 minutes then
-        Cancel running cloudlets
-        Realase VM
-        Remove VM from QL
-    end if
-end for
```

Fig. 6: The proposed executor (Suprex), the main component for the execution of the scaling commands, and the updater component updating quarantined VMs

- *Cost-Aware* surplus VM selection policy
  This policy (Algorithm 5) calculates the elapsed time of the last billing hour (*PT*) (lines 5 to 7) for each VM. The VM has the highest selection priority (lines 8 to 10) if it is exactly at the end of the last hour (for example exactly 3 hours). Afterward, the VM with the greatest *PT* compared to other studied VMs (i.e., the VM with the greatest utilization of the last hour of its life cycle) is selected as the surplus VM (lines 11 to 14).

Algorithm 5: *Cost-Aware* policy in the selection of surplus VM

```
1.  Input:        On-Demand Vm List ($PL + SL$)
2.  Output:       surplusVM
3.  Parameters:   maxPT ←MIN_VALUE, availableTime
4.  for vm in on-Demand VmLis do
5.      PT ← null;
6.      availableTime ← Clock - VM.getRequestTime();
7.      PT ← availabletime % anHour; // % = modulus operation
8.      if PT = 0 then
9.          PT ← anHour;
10.     end if
11.     if PT > maxPT then
12.         maxPT ← PT;
13.         surplusVm ←vm;
14.     end if
15. end for
16. return surplusVm;
```

- *Load-aware* surplus VM selection policy
  The *load-aware* policy is responsible for considering the VM load at the time of selection. According to Algorithm 6, for every VM, the remaining service time of the load in the VM is estimated through an inquiry from its scheduler (entitled *vmRemainedLoad*) (lines 5 to 16). The remaining service time of any VM is calculated from the total remaining service time of its cloudlets. The remaining service time of any cloudlet is also estimated by deducting the amount processing already performed from its total service time (lines 8 to 11). Since in our desired system processing a cloudlet with an X-core processing requirement on a VM with a X+1-core processing requirement would not speed up the process, this issue has also been taken into consideration (lines 12 to 14). Finally, if the VM has a lower load compared to other studied VMs, it is selected as surplus VM (lines 17 to 20).

Algorithm 6: *Load-Aware* policy in the selection of surplus VM

```
1.  Input:       On-Demand Vm List $(PL + SL)$
2.  Output:      surplusVm
3.  Parameters:  minLoad ←MAX_VALUE, vmRemainedLoad, length, ranTime, remainedServiceTime
                 cloudletList ←new ArrayList <Cloudlet> ()
4.  for vm in onDemandVmList do
5.      vmRemainedLoad ←0;
6.      cloudletList ←vm.getScheduler().getCloudletList();
7.      for cloudlet in cloudletList do
8.          remainedServiceTime ← 0
9.          length ←cloudlet.getLength()× cloudlet.getNumberOfPes();
10.         ranTime ←Clock - cloudlet.getFirstSubmissionTime();
11.         remainedServiceTime ←length - (ranTime × (vm.getMips()× vm.getNumberOfPes() );
12.         if vm.getNumberOfPes() > cloudlet.getNumberOfPes() then
13.             remainedServiceTime ←length - (ranTime × (vm.getMips()× cloudlet.getNumberOfPes() ) );
14.         end if
15.         vmRemainedLoad ← vmRemainedLoad + remainedServiceTime;
16.     end for
17.     if vmRemainedLoad < minLoad then
18.         minLoad ←vmRemainedLoad;
19.         surplusVm ←vm;
20.     end if
21. end for
22. return surplusVm;
```

5. Performance evaluation

This section explains experiments conducted to evaluate the performance of Suprex in CloudSim Simulator [47]. The experiment scenario is as follows: the AP rents a limited number of VMs from the CP for hosting a Web application. Afterward, the users start sending their requests to the application. Meanwhile, the scaling mechanism automatically prevents resource under-provisioning and over-provisioning by employing on-demand VMs. The purpose of the mechanism is cost saving and maintaining end user satisfaction.

In each experiment, one of the following executors is used: Default (common in research [43]), Professional (improved), and Suprex (proposed). Other parameters (parameters related to monitoring, analysis and planning phases) stay fixed so that the performance of the executors is evaluated fairly.

5.1. Experiment setup

Each experiment is comprised of end user, AP, and CP, the detail of which shall be presented in the following subsections (see Fig. 2).

5.1.1. End user entity

We used NASA workload [48] as the emulator of Web users requests to the AP (see Fig. 7). The NASA workload is very common for the performance evaluation of web application auto-scaling [25, 49-52]. This workload represents realistic load variations over time that makes the results and conclusions more realistic and reliable to be used in real environments. The workload comprises 100960 user requests sent to NASA Web servers during a day. Extreme fluctuations in this workload can trigger a realistic experiment for the scaling mechanism. Research shows that the pattern of user request arrivals to many Websites complies with the similar pattern of this workload. In other words, the majority of websites experience reduced incoming load during early hours of the day, increased incoming load with the start of office hours, significant fluctuations at the end of office hours, and a decreasing pattern during the closing hours of the night [53, 54].

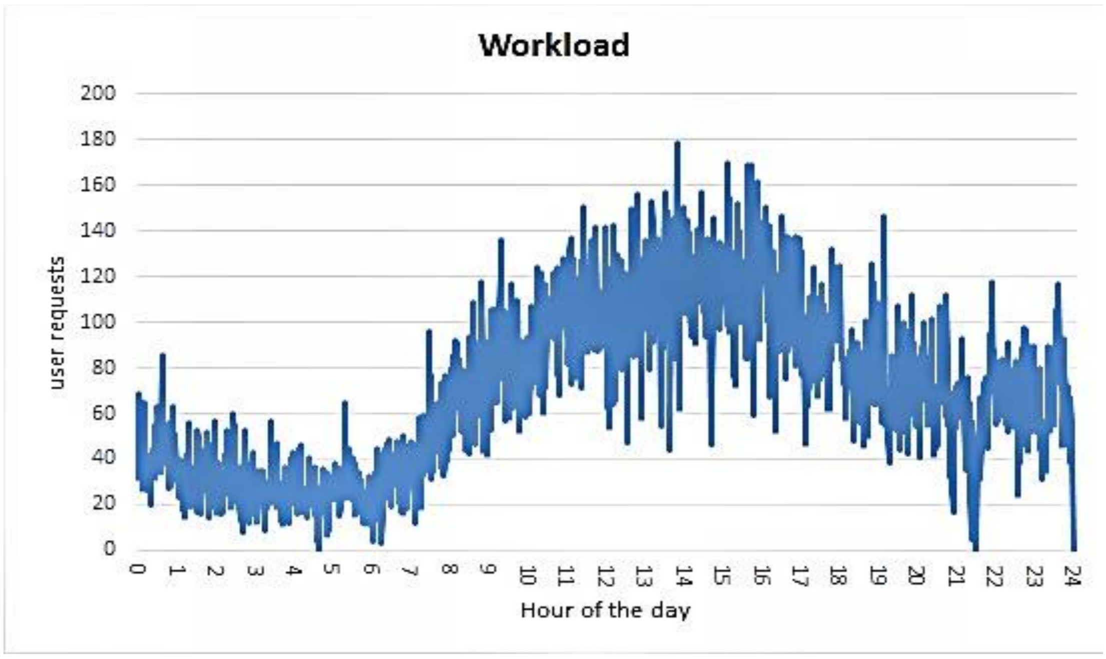


Fig. 7: User requests to NASA Kennedy Space Center WWW server in Florida on July 6th, 1995

5.1.2. AP entity

In this system, the AP hosts the application by renting two reserved machines with the size of *t2.large*. Then, with the help of the auto-scaling mechanism, on-demand VMs with the size of *t2.medium* are used to control incoming load fluctuations (more details on VM configuration in [43]). A number of parameters can be seen in Table 3.

Note that the delayed startup of on-demand VMs is a fundamental issue in research based on cloud simulation environment. In simulations, some researchers determine this delay as a fixed number [43, 55] or even better as normal distribution [23]. Nonetheless, an interesting study by Mao and Humphrey [45] has shown that this delay depends on factors such as VM size, VM request time (time of day), etc. In Equation (9), we study this issue with more details. In Equation (9) where *BASE* is a fixed minimum amount (equivalent to 5 minutes) which is added to the total of delay dependent on *VMSIZE* and *TimeOfTheDay* in accordance with Fig. 8 to calculate the delay in VM startup. According to Fig. 8, the larger the VM size, the longer the startup delay is. The VM startup is also affected by the time of the day the VM is requested as the rise in the number of VM requests by CP clients in working hours will increase the delay. It should be noted other factors such as servers' geographical location and step sizes of scale-up and scale-down operations [4] can also be added to this equation which are omitted here for the sake of brevity. We also consider the delay for releasing a VM to be zero since it is negligible [23, 45].

$$VM\ Delay = BASE + VMConfig + TimeOfTheDay \tag{9}$$

		VM Config			
		micro	**small**	**medium**	**large**
Time of The Day (hour)	**0-6**	**0**	**1**	**2**	**3**
	7-12	**1**	**2**	**3**	**4**
	13-18	**2**	**3**	**4**	**5**
	19-24	**1**	**2**	**3**	**4**

Fig. 8: The effect of VM size and time of VM request on the delay in VM startup

5.1.3. CP entity

The CP entity potentially exists in CloudSim, but features need to be added to VM class for cost management and issuance of resource rental bills. Task scheduling and resource scheduling were both time-shared (similar to [43]).

Table 3: Values determined for experiment parameters

Parameter	**Value**
Workload	NASA traces
Resource Utilization Thresholds	$U^{low\_thr} = 20\%$ and $U^{high\_thr} = 80\%$
Response Time Thresholds	$RT^{low\_thr} = 200ms$ and $RT^{high\_thr} = 1000ms$
Scaling Interval	Δt = 10 min
Desired Response Time	*DRT* = 1000*ms* = 1*s*
Load Balancing Policy	Round-Robin
Configuration of VMs	t2.medium and t2. Large (More information about the configuration in [43]
Maximum On-demand VM Limitation	*maxVM* = 10 VM
Task and Resources Scheduling policy	Time-Shared

5.2. Performance criteria

Evaluation criteria are presented in eight categories which are as follows:

- The **cost** of renting on-demand machines as the most important criteria. According to CPs' policy [24], the cost is calculated by rounding up the rented hours of all VMs. Lines 5-7 of Algorithm 5 show how this calculation is performed.
- **Response time**: Based on studies [43], the desired Web application response time of 1 second was agreed upon (*DRT* = 1s). This is obtained by calculating the average of all observed M^{rt}. It is desirable to see that the mechanism's performance fulfils this agreement. The standard deviation (SD) of response time is also measured; the lower SD shows timely decisions and prevention of serious delays in responses.
- **The rate of SLA violation**: the total length of time (in hours) of delayed response to user request, which is calculated according to (8).
- **Resource utilization**: Mean percentage of the utilization of resources used obtained by calculating the average of all M^{u}. The SD of resource utilization is also measured; the lower SD indicates that the VMs are not surprised by the load [43].
- **Oscillation mitigation or mechanism overload control** which is evaluated by measuring the number of VMs provisioned and de-provisioned by the executor. Fewer decisions are indicative of higher accuracy.
- **Time to adaptation**: If the planner discovers disturbance of desirable conditions for resource utilization and response time, it recognizes the application as maladapted. In this situation, the executor considered more efficient if it makes the scale-up decision in a way that the time for the re-establishment of application adaptability is minimized; in other words, the one which establishes desirable conditions for resource utilization and response time. Therefore, this metric shows how long it takes on average (in seconds) to re-establish adaptability after each scale-up decision in the mechanisms**.**
- **VM Minutes**: The average length of time VMs are rented by the AP, i.e., the period of time from demand to release of a VM. The higher amount of this criterion is indicative of the stability in resource provisioning [41].
- **Contradictory Scaling Actions (CSA)**: Provision and release of a VM in a short period of time (for example, in two consecutive scaling intervals), or vice versa, which results in increased cost and SLA violation [4]. Efficient executors must be able to avoid CSA.

5.3. Experimental results and discussion

5.3.1. Experimental results

According to Table 4, there are three scenarios for experiments; the parameters of the three phases of monitoring, analysis, and planning are fixed in all three experiments but different executors are used in the scaling mechanism. The experiments were carried out as follows: The first experiment used the Default executor for the execution of scaling decisions. This executor makes decisions by adapting unaware LUFD [43] or FUFD [44] policies in the selection of the surplus VM for the execution of scale-down decisions. A Professional executor was used in the second experiment to observe the effectiveness of aware selection of the surplus VM in scaling decisions; the Professional one utilizes two heuristic *cost-aware* and *load-aware* policies. The proposed executor Suprex was used in the third experiment. In this experiment, instead of immediately releasing the selected surplus VM, the executor sends it to quarantined status to be restored if the need arises.

Table 4: The scenario of discussed experiments

Scenario	**Auto-scaling parameters (based on MAPE loop phases)**				
	Monitoring	**Analysis**	**Planning**	**Execution**	
				Type	**Feature**
Scenario 1	Algorithm 2	Hybrid (Algorithm 3)	Rule-based (Algorithm 4)	Default	LUFD and FUFD surplus VM selection
Scenario 2	Algorithm 2	Hybrid (Algorithm 3)	Rule-based (Algorithm 4)	Professional	*Cost-Aware* and *Load-Aware* surplus VM selection
Scenario 3	Algorithm 2	Hybrid (Algorithm 3)	Rule-based (Algorithm 4)	Suprex (Proposed)	*Cost-Aware* and *Load-Aware* surplus VM selection + VM Quarantining

First we conducted a preliminary study to check if the proposed mechanism yields a desirable overall performance. There are two signs in preliminary results that determined the performance of the mechanism was desirable; (1) the study of the mechanism's throughput indicated that despite severe incoming load fluctuations, the throughput was effectively close to 100% (mean of 100%) (Fig. 9); (2) the comparison between the pattern of incoming user requests and the pattern of VMs provisioned by the mechanism shows that the mechanism carries out VM provisioning appropriately (Fig. 10). From now on, the number of VMs is the total number of reserved and on-demand VMs unless otherwise mentioned.

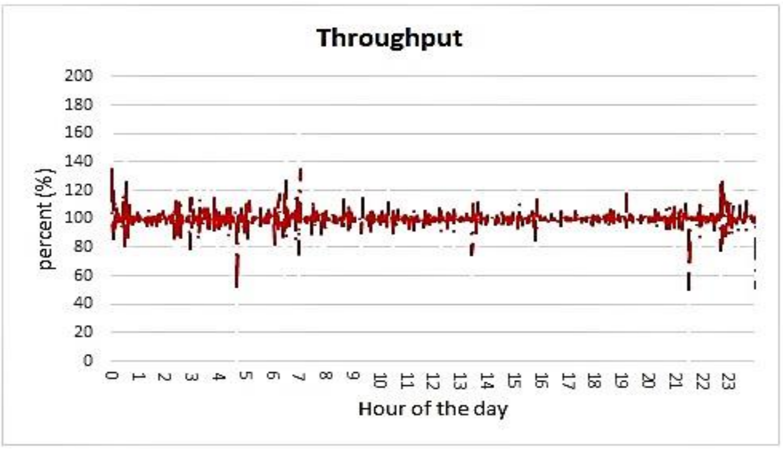


Fig. 9: Throughput in the mechanism equipped with the Default executor

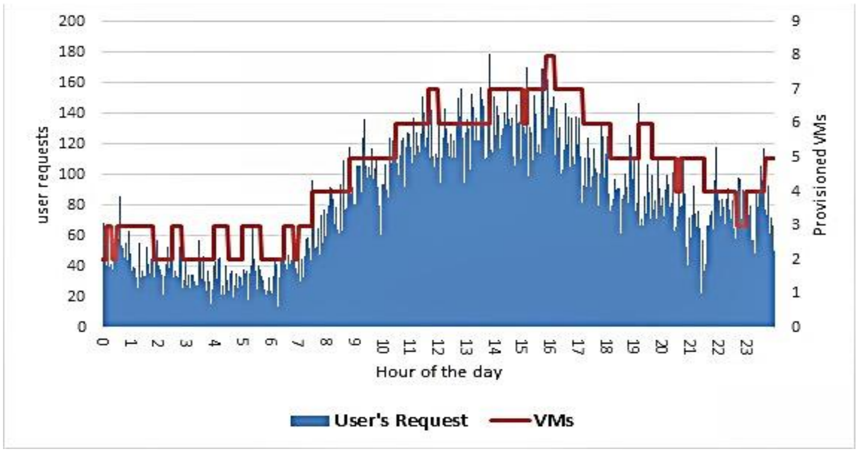


Fig. 10: The appropriateness of the pattern of the incoming load and provisioned VMs in the mechanism equipped with the Default executor

The results obtained from the evaluation of the three executors are as follows. The most important criterion for verifying the effectiveness of executors is their influence on cost. Fig. 11 illustrates that after adding each feature to the executor, there was a considerable cost saving. In addition, Fig. 12 shows that although Default and Professional executors were able to maintain the response time at the desired level, according to the SLA (*DRT* < 1s), Suprex managed to improve this criterion. Response time SD in this experiment reached 1.69 using the Suprex executor, while this figure was 1.96 in other experiments. Improvement in the response time SD by the Suprex executor is indicative of timely decisions and prevention of serious delays in responses. Also, as a result of improved response time, SLA violation decreased by 5% (Fig. 13). On the other hand, the percentage of resource utilization decreased in the mechanism equipped with Suprex executor (Fig. 14). The reason is that with the possibility of restoring quarantined VMs without any delay requests are not accumulated on resources and this technique results in lower utilization in some situations. The other reason for decreased utilization is the fact that quarantined VMs that do not receive any load is included in the calculation of utilization. The SD of resource utilization improved by applying Suprex executor to the mechanism. Improvement in the SD of utilization by applying Suprex reveals that VMs are less often surprised by the incoming load when Suprex is exploited. The comparison between the amount of oscillation mitigation (or scaling overload control) shows the system's stability after applying Suprex (Fig. 15). The reason is that Suprex's rate of provisioning and releasing VMs was 24% less than the Default executor. Finally, the time to adaptation metric for the mechanism taking advantage of the Suprex executor is almost half of the rest (Fig. 16). The reason for Suprex's faster application adaptation is that this executor overcomes the challenge of delayed startup of a new VM. Default and Professional executors increase adaptation time since in their method for the execution of scaling decisions, application adaptation is delayed by the startup of a new VM and its effect in the processing of user requests.

In overall, the Professional executor is only efficient in cost saving but Suprex in addition to cost shows more efficiency in meeting SLA. The reason behind this performance improvement by Suprex is its aware selection techniques and quarantining surplus VMs.

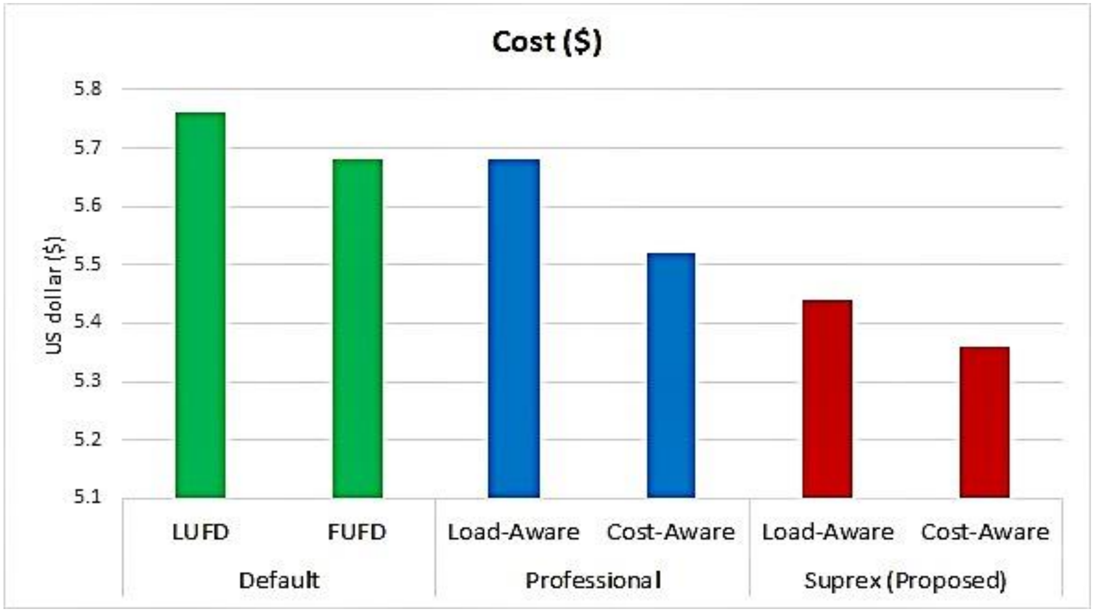


Fig. 11: Comparison between the costs of renting VMs after using each of the executors in the mechanism

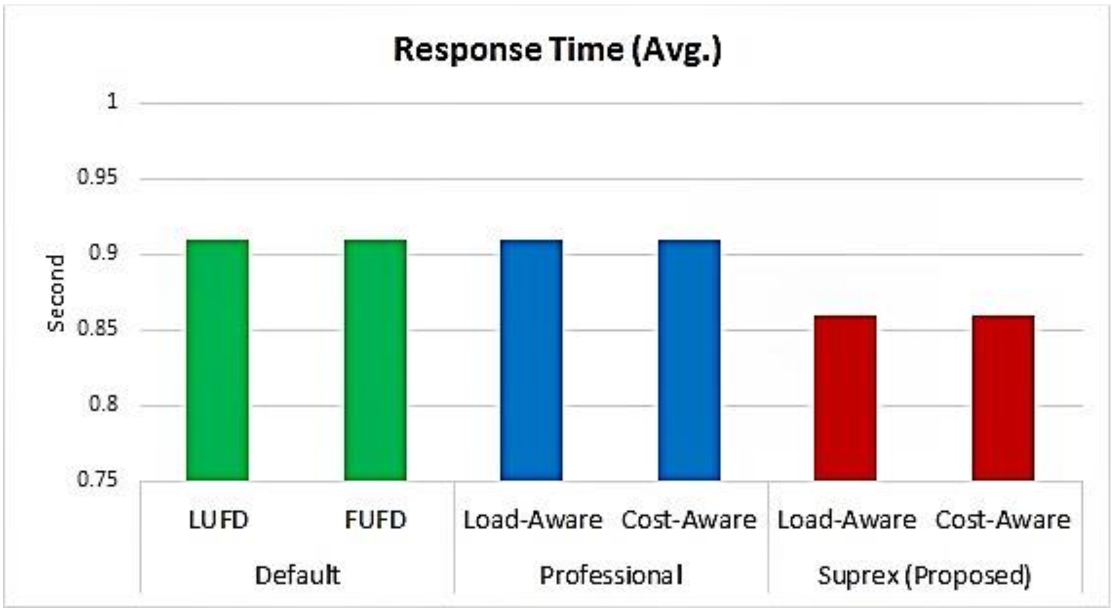


Fig. 12: Comparison between the mean response times after using each of the executors in the mechanism

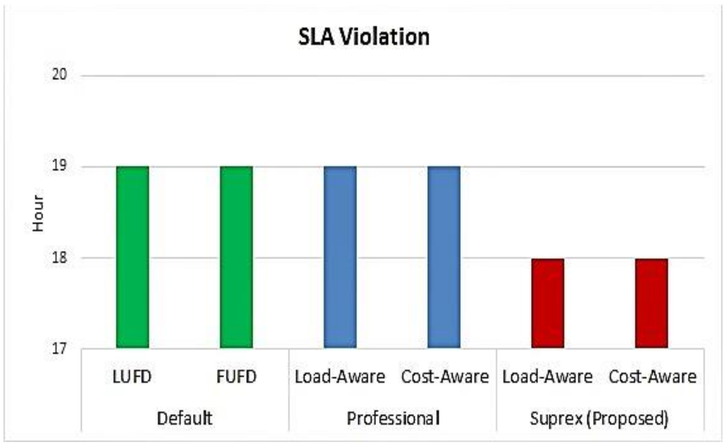


Fig. 13: Comparison between the rates of SLA violation after using each of the executors in the mechanism

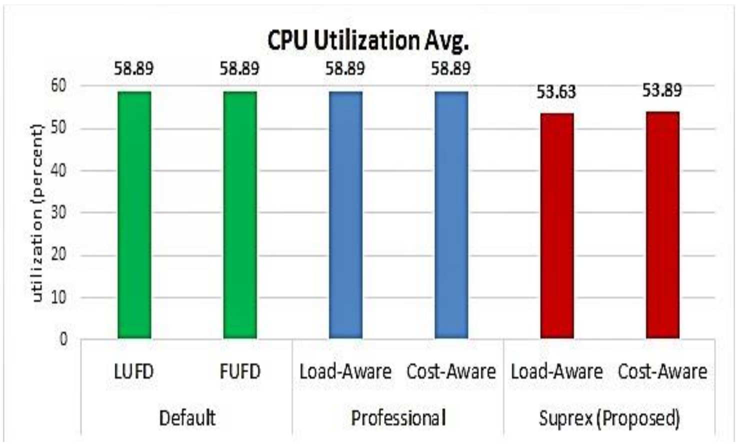


Fig. 14: Comparison between the percentages of resource utilization after using each of the executors in the mechanism

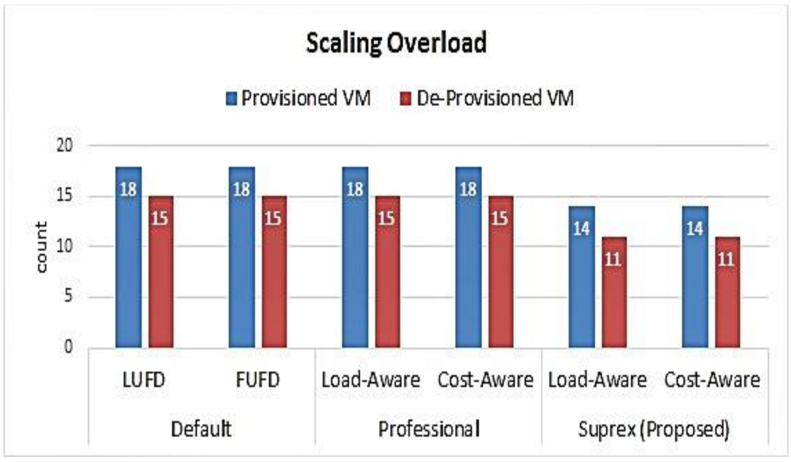


Fig. 15: Comparison between the oscillation mitigation of the mechanism (or overload control) after using each of the executors in the mechanism

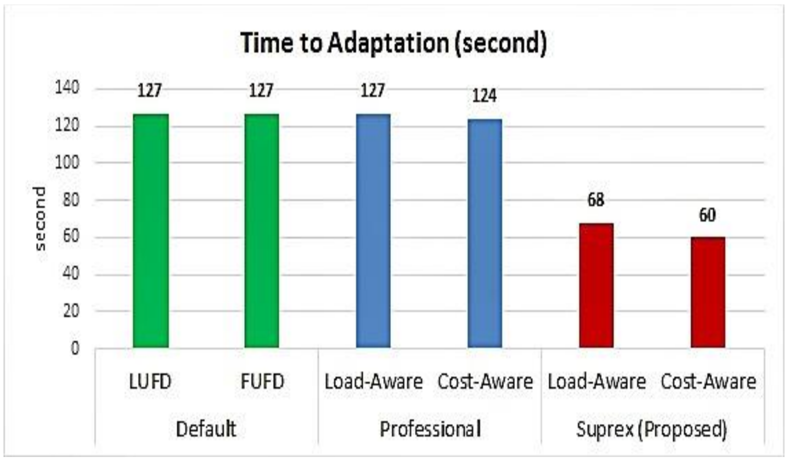


Fig. 16: Comparison of time to adaptation after using each of the executors in the mechanism

5.3.2. Discussion

With a closer look, this section studies how the two features of aware selection and quarantining surplus VMs influence the proposed Suprex executor.

The results show that by adding a *cost-aware* selection of surplus VMs, the Professional executor managed to decrease the imposed cost relative to the Default executor ($5.76 to $5.52). Note that whenever the figure witnessed a fall, a scaling down decision is made and a VM is released. According to Fig. 17, the number and time of these decisions were similar in the experiments. A total of 17 decisions are taken but in only 8 of them, both executors make the same selection (s = same). The reason why selections from the beginning of the simulation until 7 A.M. are similar is that in the selection of surplus on-demand VMs, the executors are only faced with one release option. Moreover, in the same period, they fail to take actions for scale-down decisions twice (no-ac = no action) because there are no on-demand VMs at that time. However, in 6 of the 7 remaining selections, the Professional executor selects the VM with longer elapsed minutes compared to the Default executor (longer *MaxPT*) equipped with the *cost-aware* surplus VM selection policy. In two cases of selection by the Professional executor, the surplus VM with exactly *X* hours of use is selected; this choice prevented the mechanism from the addition of another hour to the billing of the VM. These savings in the time of surplus VM selection result in the reduction of total cost imposed by the Professional executor for renting resources compared to the Default executor.

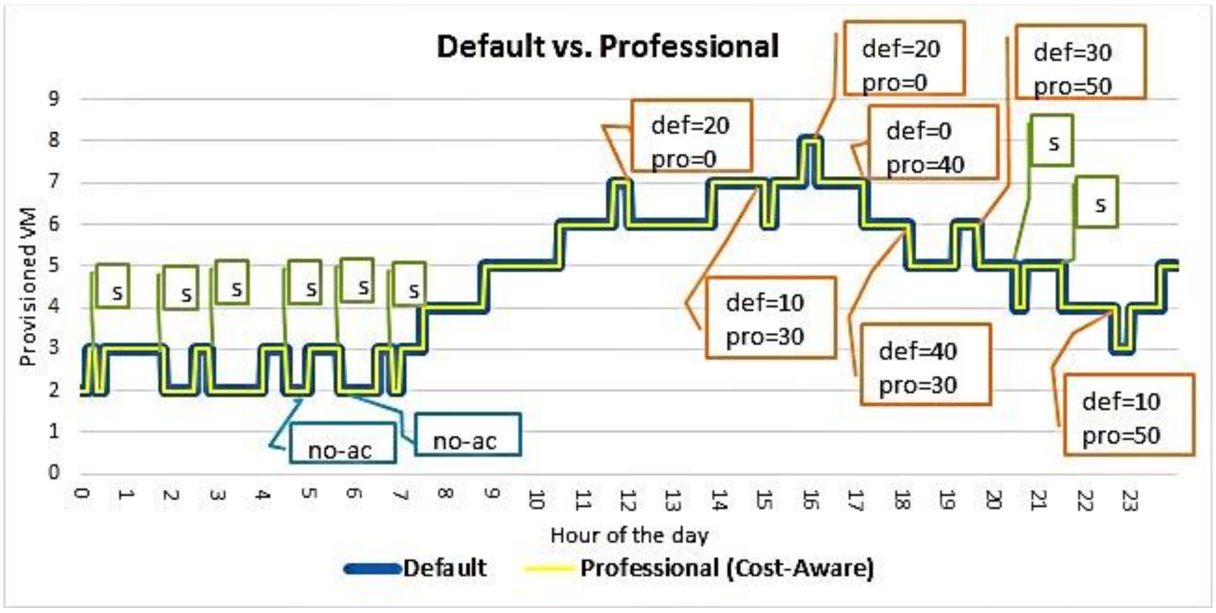


Fig. 17: Comparison between the performance of Default and Professional executors in the selection of the surplus VM, s = same, no-ac = no action, def = default, pro = professional. For example, pro = 40 means the professional executor selected a VM with the 40-minute use of the last hour.

In regards to the technique for quarantining and restoring surplus VMs, the results in Fig. 11 show that the Suprex executor is more effective than the Professional executor in cost reduction (reduction from 5.52 to 5.36). This happens since after the selection of the surplus VM, the Professional executor immediately releases it (see Fig. 18) while after the selection of a surplus VM, Suprex moves the VM to the quarantined status. Yellow circles in Fig. 18 show when Suprex has done so. The green lines connected to the circles show how long the VM remained in quarantined status. Black triangles show the times that the mechanism restored quarantined VMs (5 times) in the execution of scale-up decisions, as a result of which the request of a new VM was avoided. Interestingly, the Suprex executor reduces the cost compared to the Professional executor while a greater number of VMs are rented by the AP. This fact as well as avoidance of provisioning new VMs confirms the mechanism’s stability. Although quarantined VMs are not available to the load balancer, the AP do not have to wait for VM startup, and user requests are responded quickly given the possibility of restoring them in scale-up decisions.

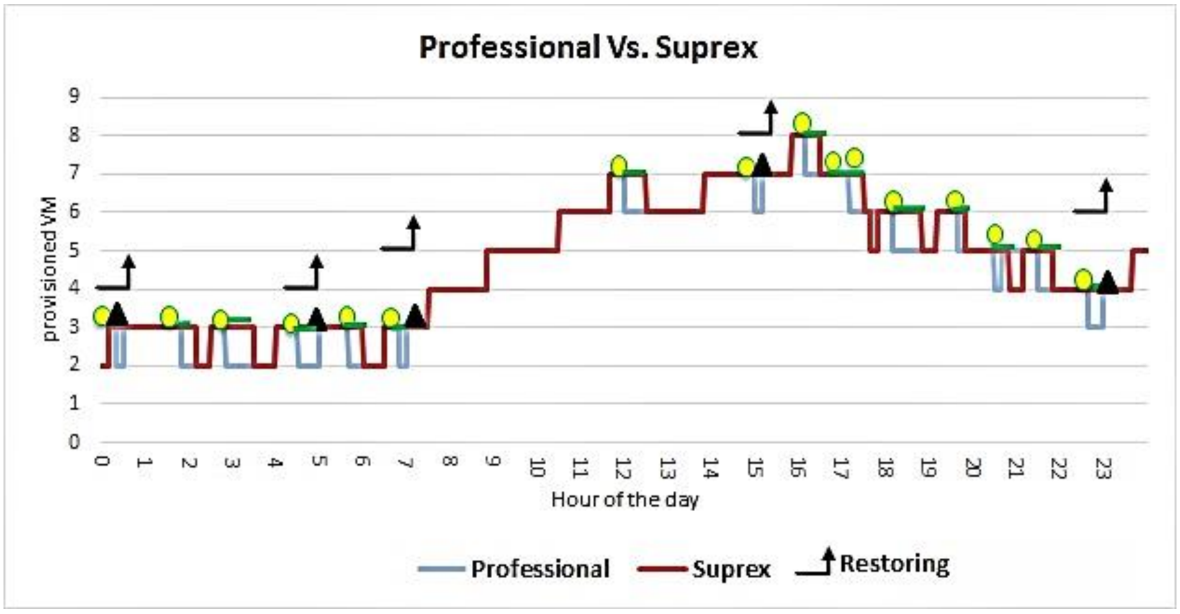


Fig. 18: Comparison between the performance of Professional and Suprex executors in the execution of scale-down and scale-up decisions

Figs. 17 and 18 show that the proposed techniques in the selection of surplus VMs and quarantining them results in more stable VM provisioning. Fig. 19 compares the performance of the mechanism regarding the VM Minutes metric as an indicator of stability for each of the three executors. According to Fig. 19, pattern of changes in the number of VMs for both Default and Professional executors is similar as a result of similar reaction to the planner's decisions. In the mechanism taking advantage of these two executors, the longest duration the number of VMs remained constant was 100 minutes; between hours 08:50 to 10:30, while the same duration was 140 minutes for the Suprex executor. In addition, Suprex had 120-minute (twice) and 110-minute stability. This shows a better performance compared to the Default and Professional executors (see Fig. 19). More importantly, Suprex makes the mechanism more stable exactly when the incoming load has the highest fluctuations while the Default and Professional executors fail to do so due to their inability in controlling contradictory scaling decisions (CSD) of the planner; for example, the period between 13:50 to 15:50 when a peak happens in the incoming load. On average, the VM Minutes metric for Default and Professional executors was 204 while this figure was 287 minutes for the Suprex executor which shows that Suprex's is 40% more effective in keeping the mechanism stable.

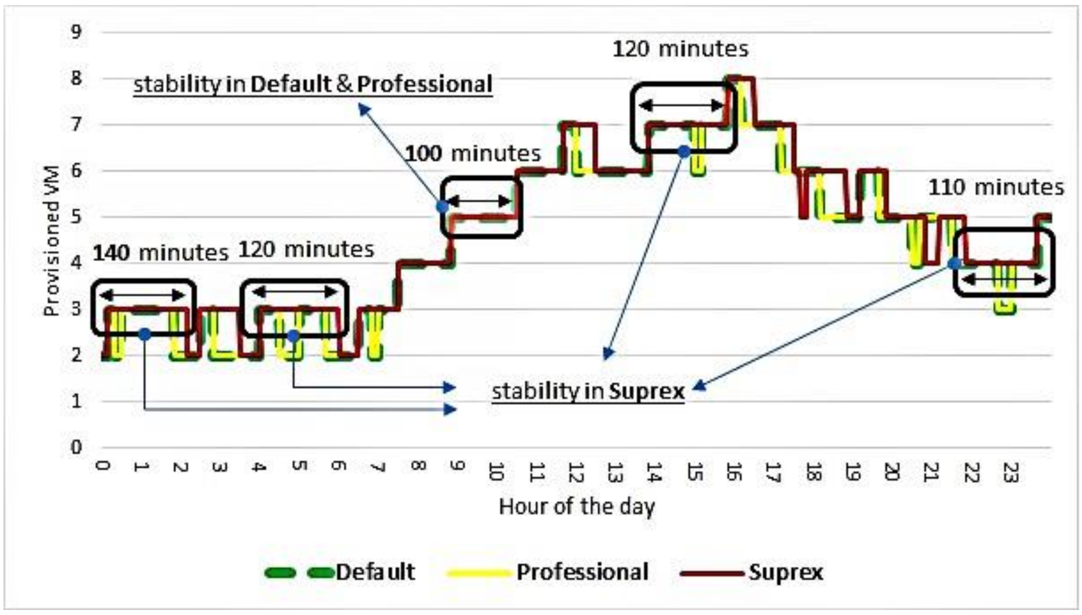


Fig. 19: Comparison between the performance of executors regarding the VM Minutes criteria

Study of the VM Minutes metrics also showed that one of the main reasons for the poor stability is failure in controlling the CSD of the planner (this is when scale-down decision and immediate scale-up decision happen consecutively). Fig. 20 shows the CSD of the planner and the performance of executors in this situation. During each experiment, the planner made 4 contradictory scaling decisions. As shown in Fig. 20, both Default and Professional executors had 4 CSAs where a VM (🡇) is released and immediately a new one (🡅) is provisioned upon the next scaling interval. These contradictory actions never reported when the Suprex executor is exploited; because instead of provisioning a new VM, Suprex covers the scale-up command by restoring a quarantined VM. As a result, excessive costs and violation of the SLA is prevented.

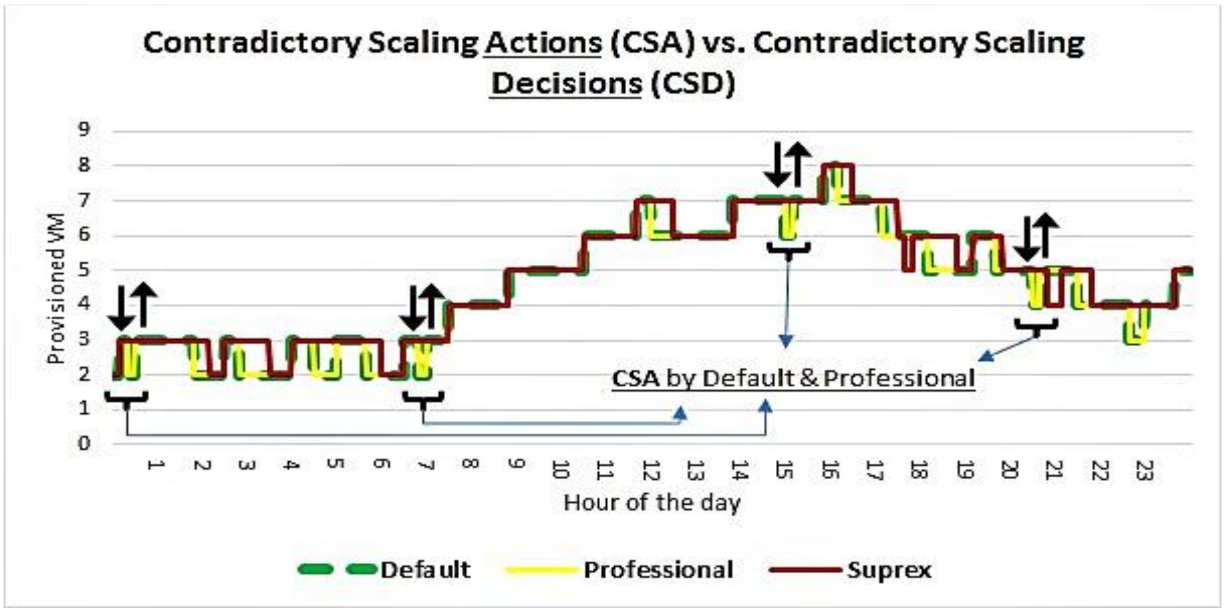


Fig. 20: Comparison between the performance of executors in the face of contradictory scaling decisions by the planner

It is worth mentioning that all these improvements are attained only by applying a different executor in the mechanism, not a completely different scaling mechanism. The fundamental differences between these two techniques are the aware selection and quarantining the surplus VMs. We conclude that the techniques presented in this research can significantly improve the effectiveness of the auto-scaling mechanisms.

6. Conclusion

In this paper, we proposed Suprex, an executor for the cost-aware auto-scaling mechanism. Suprex benefits from two heuristic features: (1) aware selection of surplus VMs during the execution of scale-down commands and (2) quarantining and immediate restore of surplus VMs (as opposed to immediate release). Simulation results show that Suprex can achieve a 7% reduction in resource rental costs for the AP while improving response time by up to 5% and decreasing SLA violation and the mechanism's oscillation overload by 5% and 24%, respectively. Suprex overcomes the challenge of delayed startup for new VMs without the help of cool-down time or vertical scaling. Finally, with respect to the close relationship between Suprex features and the pay-per-use rental model, APs can take advantage of Suprex in their auto-scaling mechanisms.

For future work, we plan to explore the impact of making quarantined VMs accessible by the load balancer to redirect load to them and evaluate Suprex's performance against an executor with the feature of vertical scaling. We also will investigate utilization of Spot instances offered by CPs such as Amazon to save cost. Another future work can be the investigation of using resources from multiple cloud environments in the auto-scaling mechanism.

References

[1] EC2. *Elastic Compute Cloud* Available: http://aws.amazon.com/ec2/

[2] 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, pp. 559-592, 2014.

[3] E. F. Coutinho, F. R. de Carvalho Sousa, P. A. L. Rego, D. G. Gomes, and J. N. de Souza, "Elasticity in cloud computing: a survey," *annals of telecommunications-annales des télécommunications*, vol. 70, pp. 289-309, 2015.

[4] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling Web Applications in Clouds: A Taxonomy and Survey," *arXiv preprint arXiv:1609.09224*, 2016.

[5] M. G. Arani and M. Shamsi, "An Extended Approach for Efficient Data Storage in Cloud Computing Environment," *International Journal of Computer Network and Information Security*, vol. 7, p. 30, 2015.

[6] Y. Shen, H. Chen, L. Shen, C. Mei, and X. Pu, "Cost-Optimized Resource Provision for Cloud Applications," in *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and*

Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS), 2014 IEEE Intl Conf on, 2014, pp. 1060-1067.

[7] M. Amiri and L. Mohammad-Khanli, "Survey on Prediction Models of Applications for Resources Provisioning in Cloud," *Journal of Network and Computer Applications,* 2017.

[8] A. Computing, "An architectural blueprint for autonomic computing," *IBM White Paper,* vol. 31, 2006.

[9] M. Mohamed, M. Amziani, D. Belaïd, S. Tata, and T. Melliti, "An autonomic approach to manage elasticity of business processes in the Cloud," *Future Generation Computer Systems,* 2014.

[10] M. Ghobaei-Arani, S. Jabbehdari, and M. A. Pourmina, "An autonomic approach for resource provisioning of cloud services," *Cluster Computing,* pp. 1-20, 2016.

[11] R. Weingärtner, G. B. Bräscher, and C. B. Westphall, "Cloud resource management: A survey on forecasting and profiling models," *Journal of Network and Computer Applications,* vol. 47, pp. 99-106, 2015.

[12] M. Ghobaei-Arani, M. Shamsi, and A. A. Rahmanian, "An efficient approach for improving virtual machine placement in cloud computing environment," *Journal of Experimental & Theoretical Artificial Intelligence,* pp. 1-23, 2017.

[13] S. Singh and I. Chana, "Resource provisioning and scheduling in clouds: QoS perspective," *The Journal of Supercomputing,* vol. 72, pp. 926-960, 2016.

[14] 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, pp. 155-162, 2012.

[15] J. Huang, C. Li, and J. Yu, "Resource prediction based on double exponential smoothing in cloud computing," in *Consumer Electronics, Communications and Networks (CECNet), 2012 2nd International Conference on*, 2012, pp. 2056-2060.

[16] A. A. Bankole and S. A. Ajila, "Cloud client prediction models for cloud resource provisioning in a multitier web application environment," in *Service Oriented System Engineering (SOSE), 2013 IEEE 7th International Symposium on*, 2013, pp. 156-161.

[17] S. A. Ajila and A. A. Bankole, "Cloud client prediction models using machine learning techniques," in *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*, 2013, pp. 134-142.

[18] N. R. Herbst, N. Huber, S. Kounev, and E. Amrehn, "Self‐adaptive workload classification and forecasting for proactive resource provisioning," *Concurrency and computation: practice and experience,* vol. 26, pp. 2053-2078, 2014.

[19] H. R. Qavami, S. Jamali, M. K. Akbari, and B. Javadi, "Dynamic Resource Provisioning in Cloud Computing: A Heuristic Markovian Approach," in *Cloud Computing,* ed: Springer, 2014, pp. 102-111.

[20] A. G. García, I. B. Espert, and V. H. García, "SLA-driven dynamic cloud resource management," *Future Generation Computer Systems,* vol. 31, pp. 1-11, 2014.

[21] S. Singh and I. Chana, "Q-aware: Quality of service based cloud resource provisioning," *Computers & Electrical Engineering,* vol. 47, pp. 138-160, 2015.

[22] M. Fallah, M. G. Arani, and M. Maeen, "NASLA: Novel Auto Scaling Approach based on Learning Automata for Web Application in Cloud Computing Environment," *International Journal of Computer Application,* vol. 117, pp. 18-23, 2015.

[23] M. D. de Assunção, C. H. Cardonha, M. A. Netto, and R. L. Cunha, "Impact of user patience on auto-scaling resource capacity for cloud services," *Future Generation Computer Systems,* vol. 55, pp. 41-50, 2016.

[24] D. Moldovan, H. L. Truong, and S. Dustdar, "Cost-Aware Scalability of Applications in Public Clouds," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, 2016, pp. 79-88.

[25] J. Li, S. Su, X. Cheng, M. Song, L. Ma, and J. Wang, "Cost-efficient coordinated scheduling for leasing cloud resources on hybrid workloads," *Parallel Computing,* vol. 44, pp. 1-17, 2015.

[26] R. Z. Khan and M. O. Ahmad, "Load Balancing Challenges in Cloud Computing: A Survey," in *Proceedings of the International Conference on Signal, Networks, Computing, and Systems*, 2016, pp. 25-32.

[27] J. Panneerselvam, L. Liu, N. Antonopoulos, and Y. Bo, "Workload analysis for the scope of user demand prediction model evaluations in cloud environments," in *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*, 2014, pp. 883-889.

[28] A. Gholami and M. G. Arani, "A trust model based on quality of service in cloud computing environment," *International Journal of Database Theory and Application,* vol. 8, pp. 161-170, 2015.

[29] E. Casalicchio and L. Silvestri, "Mechanisms for SLA provisioning in cloud-based service providers," *Computer Networks,* vol. 57, pp. 795-810, 2013.

[30] M. Dhingra, "Elasticity in IaaS Cloud, preserving performance SLAs," *Master's thesis, Indian Institute of Science,* 2014.

[31] M. Ghobaei-Arani, S. Jabbehdari, and M. A. Pourmina, "An autonomic resource provisioning approach for service-based cloud applications: A hybrid approach," *Future Generation Computer Systems,* 2017.

[32] A. El Rheddane, "Elasticity in the Cloud," Université Grenoble Alpes, 2015.

[33] M. C. Huebscher and J. A. McCann, "A survey of autonomic computing—degrees, models, and applications," *ACM Computing Surveys (CSUR),* vol. 40, p. 7, 2008.

[34] T. Baker, M. Mackay, M. Randles, and A. Taleb-Bendiab, "Intention-oriented programming support for runtime adaptive autonomic cloud-based applications," *Computers & Electrical Engineering,* vol. 39, pp. 2400-2412, 2013.

[35] V. Andrikopoulos, T. Binz, F. Leymann, and S. Strauch, "How to adapt applications for the cloud environment," *Computing,* vol. 95, pp. 493-535, 2013.

[36] E. Cavalcante, T. Batista, F. Lopes, A. Almeida, A. L. de Moura, N. Rodriguez*, et al.*, "Autonomous adaptation of cloud applications," in *IFIP International Conference on Distributed Applications and Interoperable Systems*, 2013, pp. 175-180.

[37] D. STEBLIUK, "Fine grained adaptation of Cloud applications with containers," 2016.

[38] Amazon. (2015, 12, 12). *Amazon Auto Scaling*. Available: https://aws.amazon.com/autoscaling/

[39] P. D. Kaur and I. Chana, "A resource elasticity framework for QoS-aware execution of cloud applications," *Future Generation Computer Systems,* vol. 37, pp. 14-25, 2014.

[40] A. N. Toosi, C. Qu, M. D. de Assunção, and R. Buyya, "Renewable-aware Geographical Load Balancing of Web Applications for Sustainable Data Centers," *Journal of Network and Computer Applications,* 2017.

[41] M. Beltrán, "Automatic provisioning of multi-tier applications in cloud computing environments," *The Journal of Supercomputing,* vol. 71, pp. 2221-2250, 2015.

[42] G. Moltó, M. Caballer, and C. de Alfonso, "Automatic memory-based vertical elasticity and oversubscription on cloud platforms," *Future Generation Computer Systems,* vol. 56, pp. 1-10, 2016.

[43] M. S. Aslanpour and S. E. Dashti, "SLA-aware resource allocation for application service providers in the cloud," in *2016 Second International Conference on Web Research (ICWR)*, 2016, pp. 31-42.

[44] M. S. Aslanpour and S. E. Dashti, "Proactive Auto-Scaling Algorithm (PASA) for Cloud Application," *International Journal of Grid and High Performance Computing,* vol. 9, pp. 1-16, 2017.

[45] M. Mao and M. Humphrey, "A performance study on the vm startup time in the cloud," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, 2012, pp. 423-430.

[46] S. Vajda, *Fibonacci and Lucas numbers, and the golden section: theory and applications*: Courier Corporation, 2007.

[47] R. Buyya, R. Ranjan, and R. N. Calheiros, "Modeling and simulation of scalable Cloud computing environments and the CloudSim toolkit: Challenges and opportunities," in *High Performance Computing & Simulation, 2009. HPCS'09. International Conference on*, 2009, pp. 1-11.

[48] M. F. Arlitt and C. L. Williamson, "Internet web servers: Workload characterization and performance implications," *IEEE/ACM Transactions on Networking (ToN),* vol. 5, pp. 631-645, 1997.

[49] J. Liu, Y. Zhang, Y. Zhou, D. Zhang, and H. Liu, "Aggressive resource provisioning for ensuring QoS in virtualized environments," *IEEE Transactions on Cloud Computing,* vol. 3, pp. 119-131, 2015.

[50] V. R. Messias, J. C. Estrella, R. Ehlers, M. J. Santana, R. C. Santana, and S. Reiff-Marganiec, "Combining time series prediction models using genetic algorithm to autoscaling Web applications hosted in the cloud infrastructure," *Neural Computing and Applications,* vol. 27, pp. 2383-2406, 2016.

[51] G. Feng and R. Buyya, "Maximum revenue-oriented resource allocation in cloud," *International Journal of Grid and Utility Computing,* vol. 7, pp. 12-21, 2016.

[52] J. Kumar and A. K. Singh, "Dynamic resource scaling in cloud using neural network and black hole algorithm," in *Eco-friendly Computing and Communication Systems (ICECCS), 2016 Fifth International Conference on*, 2016, pp. 63-67.

[53] X. Wang, A. Abraham, and K. A. Smith, "Intelligent web traffic mining and analysis," *Journal of Network and Computer Applications,* vol. 28, pp. 147-165, 2005.

[54] H. Wang, F. Xu, Y. Li, P. Zhang, and D. Jin, "Understanding mobile traffic patterns of large scale cellular towers in urban environment," in *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*, 2015, pp. 225-238.

[55] A.-F. Antonescu and T. Braun, "Simulation of SLA-based VM-scaling algorithms for cloud-distributed applications," *Future Generation Computer Systems,* 2015.