Simulation Modelling Practice and Theory 108 (2021) 102245

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

journal homepage: www.elsevier.com/locate/simpat

AutoScaleSim: A simulation toolkit for auto-scaling Web applications in clouds

Mohammad S. Aslanpour [a,c,*], Adel N. Toosi [a], Javid Taheri [b], Raj Gaire [c]

[a] Department of Software Systems and Cybersecurity, Faculty of Information Technology, Monash University, Australia
[b] Department of Mathematics and Computer Science, Karlstad University, Sweden
[c] CSIRO DATA61, Australia

ARTICLE INFO

Keywords:
Simulation
Cloud computing
Auto-scaling
Elasticity
Resource provisioning
Web application

ABSTRACT

Auto-scaling of Web applications is an extensively investigated issue in cloud computing. To evaluate auto-scaling mechanisms, the cloud community is facing considerable challenges on either real cloud platforms or custom test-beds. Challenges include – but not limited to – deployment impediments, the complexity of setting parameters, and most importantly, the cost of hosting and testing Web applications on a massive scale. Hence, simulation is presently one of the most popular evaluation solutions to overcome these obstacles. Existing simulators, however, fail to provide support for hosting, deploying and subsequently auto-scaling of Web applications. In this paper, we introduce AutoScaleSim, which extends the existing CloudSim simulator, to support auto-scaling of Web applications in cloud environments in a customizable, extendable and scalable manner. Using AutoScaleSim, the cloud community can freely implement/evaluate policies for all four phases of auto-scaling mechanisms, that is, Monitoring, Analysis, Planning and Execution. AutoScaleSim can also be used for evaluating load balancing algorithms similarly. We conducted a set of experiments to validate and carefully evaluate the performance of AutoScaleSim in a real cloud platform, with a wide range of performance metrics.

1. Introduction

Cloud computing is currently playing a major role in the efficiency improvement of businesses, industries and governments [1,2]. Such stakeholders are enthusiastic to take advantage of scalability and pay-per-use pricing model of cloud. They tend to host their applications in clouds by renting virtual machines (VMs), instead of buying physical servers with substantial costs [3]. Web applications, such as online stores and online games, must responds to variable request rates across different times of the day and week [4]. The adaptation of a Web application to its incoming load can be solved using elasticity feature of cloud resources, whereby auto-scaling mechanisms come into play.

Auto-scaling mechanisms act on dynamically increasing or decreasing the number of active VMs to adapt to the Web application resource needs [5,6]. Given the growing popularity of VMs offered by CPs, the majority of APs are now demanding efficient mechanisms for auto-scaling of resources, so that they can maintain an application's adaptation while guaranteeing its Quality of Service (QoS) [4]. To this end, one of the most technical challenges is how to implement and evaluate the performance of the designed auto-scaling mechanisms [7–9].

* Corresponding author at: Department of Software Systems and Cybersecurity, Faculty of Information Technology, Monash University, Australia.
E-mail addresses: mohammad.aslanpour@monash.edu (M.S. Aslanpour), adel.n.toosi@monash.edu (A.N. Toosi), javid.taheri@kau.se (J. Taheri), gaire@data61.csiro.au (R. Gaire).

https://doi.org/10.1016/j.simpat.2020.102245
Received 29 June 2020; Received in revised form 6 November 2020; Accepted 8 December 2020
Available online 2 January 2021
1569-190X/

Evaluating the efficiency of auto-scaling mechanisms in production environments in clouds faces huge challenges. Challenges include – but not limited to – (1) deployment impediments, (2) the complexity of setting parameters, and most importantly, (3) the cost of hosting and testing Web applications on a massive scale [8]. For APs to evaluate their auto-scaling mechanisms, they have to first deploy the application on the servers which can be a time-consuming and tricky process. After the deployment, they have to configure the monitoring system, as well as the load balancing, and overcome the complexity of setting parameters. Setting parameters for components such as the monitoring intervals, CPU thresholds, response time thresholds and cooldown time, to name a few, becomes complex as their performance can influence others. This complexity demands sufficient time for data profiling as well [10]. In addition, due to the associated cost of cloud resources, the evaluation of mechanisms in cloud environments can become extensively costly. Therefore, the number of tests should be significantly lowered before production. Given such limitations, researchers cannot evaluate auto-scaling mechanisms for large-scale scenarios with hundreds or thousands of hosting servers.

As alternative approaches, custom test-beds and bench-marking frameworks come into the picture. In a custom test-bed, the focus is on installing a virtualization software that will manage both auto-scaling and load balancing. OpenStack is one of the most relevant tools as a test-bed for customized experiments. With less deployment effort, there exist auto-scaler benchmarking frameworks (e.g., BUNGEE [11]) that allow to evaluate auto-scaling mechanisms as well. APs could utilize such test-beds and frameworks to avoid wasting money, but the first two problems (i.e., deployment impediments and complexity of setting parameters) remain unsolved [5]. Motivated by these limitations, simulation-based studies have gained considerable attention in the community to evaluate auto-scaling systems [12].

Simulation not only reduces the cost of deployments and experiments, but also eliminates the time consumed on the real-world implementation and evaluation of auto-scaling mechanisms [7]. Simulators are highly configurable and allow users to gather information about system states or performance metrics with much less complexity. On the other hand, simulated environments are still an abstraction of physical machine clusters. Hence, the reliability of their results highly depends on the abstraction level considered by developers, and most critically, the correctness and effectiveness of the simulator itself [9,13].

To the best of our knowledge, existing simulators fail to provide an environment for implementing and/or evaluating auto-scaling mechanisms for Web applications in clouds. The well-known and widely-used simulator, CloudSim [14], is playing a central role for resource management purposes and many extensions of which exist. Extensions for realization of a simulator for workflow scheduling by WorkflowSim [12] and for network reconfiguration by CloudSimSDN [15] are of its examples. However, the auto-scaling as a major field of resource management still lacks a proprietary simulation framework.

In this paper, we introduce a novel auto-scaling simulator, named “AutoScaleSim”, to study auto-scaling mechanisms for web applications in cloud environments. AutoScaleSim answers the scalability and load management issues that are considered lacking in existing simulators. The design and implementation of a simulator which addresses such issues poses significant technical challenges. These challenges mainly include (1) identifying and classifying key components of auto-scaling mechanisms which cloud providers and research community are expecting in the real-world and (2) developing the simulator in a customizable, extendable and scalable manner. The former is fulfilled by employing MAPE-K (i.e., Monitoring, Analyzing,Planning and Execution) concept which is a widely-used approach for self-adaptive mechanisms in cloud, fog and edge computing [4,5,16–20]. The latter is fulfilled by employing OOP (i.e., Object Oriented Programming) principles. AutoScaleSim also formulate and measure more than twenty low-level and high-level performance metrics (e.g., response time, CPU utilization and cost) which makes it further easy to evaluate the performance of implemented solutions. The original CloudSim [14] core logic is used to simulate the basic compute elements that composes the cloud infrastructure.

The **key contributions** of this work are as follows:

- AutoScaleSim employs the widely-used MAPE-K concept and OOP principles to realize auto-scaling and load management simulations for Web applications in clouds,
- AutoScaleSim thoroughly models the performance of auto-scaling mechanisms under the test by formulating and measuring a wide range of performance metrics,
- AutoScaleSim provides a customizable, extendible and scalable simulation environment suitable for future research, and
- AutoScaleSim’s performance is validated through a real-world implementation in an OpenStack cloud environment.

The rest of the article is organized as follows. Section 2 reviews related works. Section 3 and 4 illustrate the architecture design and implementation of AutoScaleSim, followed by the simulation process in an algorithmic view in Section 5. In Section 6, AutoScaleSim performance is validated and evaluated through varied use cases in both a real cloud platform (an OpenStack cluster) and the simulator, respectively. Observed results and limitations are discussed in Section 7. Lastly, we draw our conclusions in Section 8.

2. Related work

Several simulators covering different areas of cloud resource management have been developed. In this section, we briefly survey simulators which focus on resource management in cloud computing.

SPECI [21] simulates the performance and behavior of data centers in clouds. The SPECI fails to provide SLA, cost and VM start-up delay supports, regardless of the fact that its main functionality lies only in the low-level view of the computing system, that is the infrastructure.

CloudSim [14], pioneering work and the core of many cloud simulators, is a discrete event-based cloud simulator which enables two main entities as data center and broker to send and receive events. Despite its advantages, CloudSim has several limitations,

Table 1
A comparison study of simulators related to resource and task management in cloud.

Simulator	Platform package	SLA support	Cost support	Simulation duration	VM start-up delay	MAPE support	Considered performance metrics	Real cloud validation
CloudSim [14]	SimJava	No	Yes	Seconds	No	N/A	Low-level	No
Network-CloudSim [22]	CloudSim	No	Yes	Seconds	No	N/A	High-level	Yes
PICS [23]	N/A	Yes	Yes	Minutes	Dynamic	Monitor	Both	Yes
CloudSimSDN [15]	CloudSim	Yes	No	Seconds	No	Monitor	High-level	No
ICARO [24]	N/A	No	No	N/A	No	Monitor	Both	No
PEAS [25]	N/A	No	No	Years	Yes	Analyze&Plan	Low-level	Yes
ElasticSim [28]	CloudSim	No	Yes	Seconds	No	Monitor&Plan	High-level	No
SPECI [21]	SimKit	No	No	Seconds	No	Analyze	Low-level	No
WorkflowSim [12]	CloudSim	No	Yes	Seconds	No	Analyze	High-level	No
Container-CloudSim [6]	CloudSim	Yes	No	Seconds	Static	Monitor&Plan	Low-level	No
AutoScaleSim	**CloudSim**	**Yes**	**Yes**	**Month**	**Hybrid**	**Full**	**Both**	**Yes**

making researchers attempt to add new functionalities to repurpose it for different circumstances. The lack of support for auto-scaling mechanisms, the SLA and cost, are the main reasons for CloudSim not being appropriate for auto-scaling simulations.

NetworkCloudSim [22] is an extension of CloudSim, mainly applicable for modeling scheduling mechanisms. This simulator differs from CloudSim and SPECI when it comes to consider performance metrics by measuring high-level metrics such as response time as opposed to low-level ones such as CPU utilization. However, it fails to support SLA and dynamic auto-scaling as well as the VM start-up delay.

WorkflowSim [12], another CloudSim-based simulator, targets simulating and optimizing scientific workflows, not Web applications. WorkflowSim performs one-off scheduling and does not require to perform continuous simulation because workflow is executed akin to batch workload, whereas AutoScaleSim workload is transactional.

PICS [23] provides a framework for simulating auto-scaling mechanisms in public clouds considering both cost and SLA parameters. PICS simulates cloud resource management from end-user's perspective, but it mainly simulates workflow, not transactional workloads. PICS does not allow for long-term simulations and only can simulate in minutes, while auto-scaling mechanisms demand long lasting experiments to produce a reliable evaluation.

CloudSimSDN [15] proposes a framework for dynamic reconfiguration of networks by extending CloudSim. It also measures the performance of network elements and hosts capacity. Generally, CloudSimSDN is a toolkit for cloud provider side of resource management, not application providers' side, although it attempts to use multi-tier Web application as its case study.

ICARO [24] simulator is a useful tool to predict the pattern of workloads and to apply the obtained knowledge to resource allocation mechanisms from cloud providers' view. However, ICARO, does not provide mechanisms for application providers, brokers, or the target stakeholders like auto-scaling developers.

PEAS [25] simulator bases the performance evaluation on the scenario theory which manages to fairly measure probabilistic guarantees of the mechanism under test. With a design focus on analyzing and planning, similar to our work, PEAS is also validated through real cloud platforms. While PEAS precisely measures low-level metrics such as over- and under-provisioning, it neglects high-level metrics such as the number of failed requests as well as SLA and cost supports in evaluations in the interest of being generic.

ElasticSim [26] is a CloudSim-based toolkit for simulating workflows which supports auto-scaling mechanisms. This simulator does not cover SLA supports, and just considers high-level metrics. Also, it does not include start-up delay of a VM in cloud, which can seriously affect the simulation results [27].

ContainerCloudSim [6] is another extension to CloudSim for modeling containerized cloud environments. Other than VMs, it is beneficial to modeling resource management mechanisms in containers. Contrary to AutoScaleSim, it neither considers a dynamic delay for start-up time of each VM, nor is a tool for assessing Web applications — although start-up delay for containers is not as significant as that of VMs.

Table 1 summarizes the features of all aforementioned simulators compared to AutoScaleSim. The existing simulators do not support modeling and simulation of (1) auto-scaling of Web applications, (2) long-term simulation, (3) SLA and cost constraints, (4) hybrid VM start-up delay (dynamic and static), (5) full implementation of MAPE-K control loops, and (6) both low-level and high-level performance metrics in a cloud environment, simultaneously. Moreover, the performance of the majority of existing simulators are not validated in a real cloud platform. In comparison, AutoScaleSim has been designed to provide a single simulation tool that fulfills these gaps.

3. Architecture design

The architecture of AutoScaleSim, illustrated in Fig. 1, comprises three main entities: End-user, Application Provider, and Cloud Provider. The rationale for this design is to identify the position of auto-scaling mechanisms within cloud computing environments and to identify the precise activities within the scaling flow. Each layer is implemented by an entity object. The original CloudSim core that is an event-based simulator is used to compose the basic architecture of the Cloud Provider layer and to manage the transmitting messages between entities. We discuss the details of our architecture in the following sub-sections.

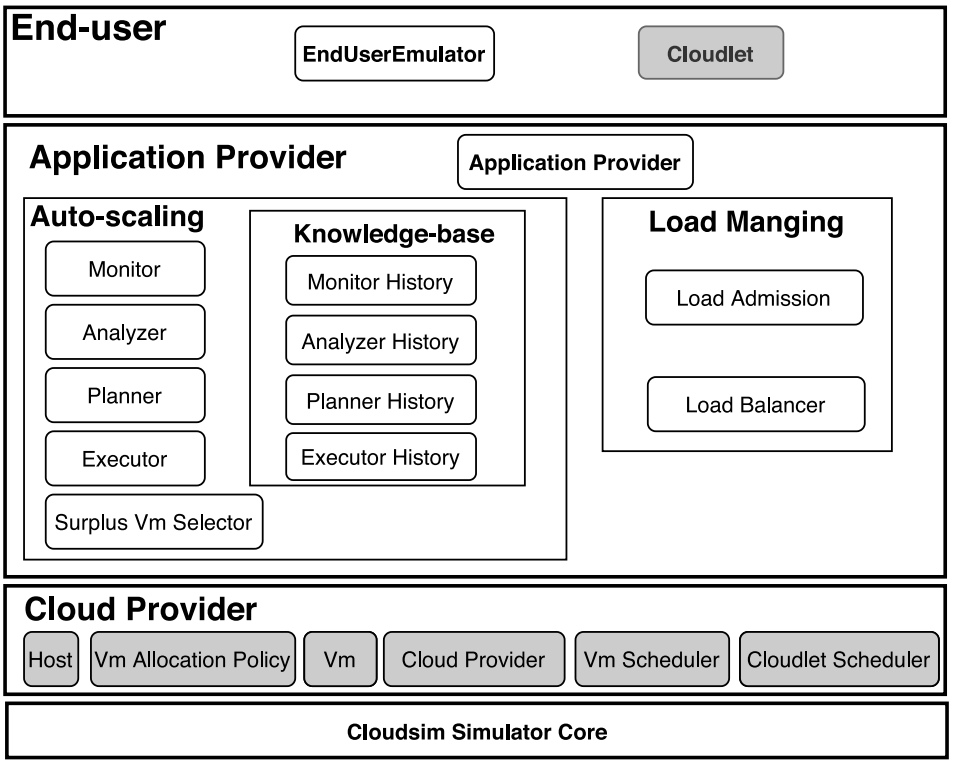

Fig. 1. The architecture of AutoScaleSim. The shaded objects exist in CloudSim by default.

3.1. End-user entity

The End-user entity is responsible for communicating with the Application provider. In the real world, each end-user creates a session by opening a website for searching, ordering, viewing, and more. Web requests can be emulated based on the real traces collected from real web servers, whereby the End-user entity continuously sends requests to the Application Provider (AP) entity over time.

3.2. Application provider entity

The Application Provider entity simultaneously performs two main activities: auto-scaling and load managing.

3.2.1. Auto-scaling

As depicted in Fig. 1, the auto-scaling system derives from the well-known concept of MAPE-K introduced by IBM. The auto-scaling process is started by running the monitor to collect both the application-level (e.g., incoming load) and resource-level (e.g., CPU utilization) observations; the analyzer then attempts to enrich the observations in a reactive or proactive manner; the planner takes the analyzed observations to make a scaling decision; the executor performs the decision by making a connection to the cloud provider through APIs. The surplus VM selection component is activated when the executor aims at performing an scale-down decision and needs to pick a VM as the surplus; lastly, all the four phases save their data into the knowledge-base component in their corresponding history record. Note that in case of dealing with auto-scaling services provided by cloud providers, it is probable that the monitoring, analyzing and executing functionalities will be handled by the cloud provider and the user is only able to tune the planner's parameters and configurations.

3.2.2. Load manager

The load managing component is responsible for receiving the incoming load from the end-users (i.e., load admission), distributing them between available VMs (i.e., load balancing) and returning back the corresponding response to the end-users. This component is directly in connection with the end-users and is a main source of application-level parameters for the monitoring phase of the auto-scaling.

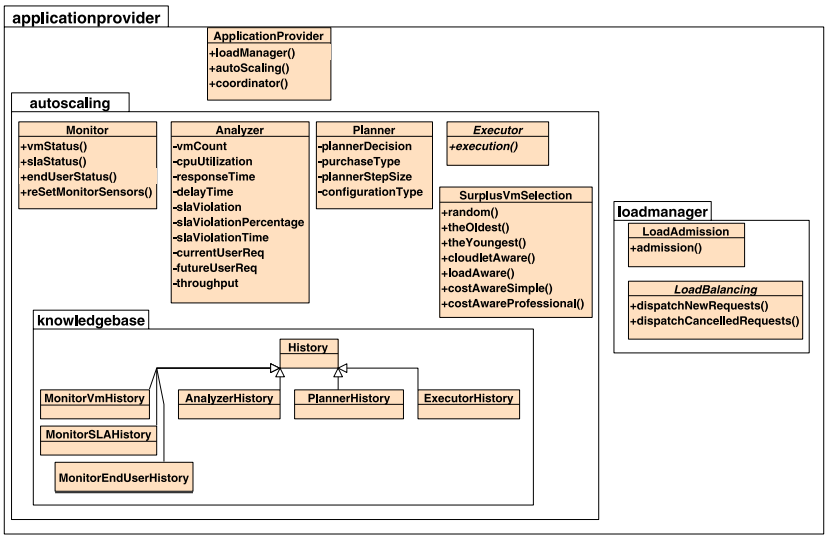

Fig. 2. Class diagram for the Application Provider package.

3.3. Cloud provider entity

The cloud provider entity provides the underlying resources to the AP. It is hosting the data centers and with the help of the virtualization techniques can offer VMs in a pay-as-you-go pricing model. The Amazon EC2 is often the reference pricing model for the research community [16,29]. Regardless of the pricing, the administration of the resources including resource allocation, scheduling, placement and so on are of main responsibilities of the cloud provider.

4. Implementation

The detailed implementation of AutoScaleSim for the three layers is elaborated in this section.

4.1. End-user entity

The End-user entity continuously extracts and sends requests, in the form of cloudlets, to the AP entity through events over time. The modified cloudlet object of CloudSim is used to model the web application requests, where it captures processing requirements (such as million instructions per requests and data size) and requests' distribution available in the real datasets. Application supported in the present version of AutoScaleSim are web applications, where there is no dependencies between requests and it reflects the request/response nature of the transactional web services. Regarding the datasets, the widely used datasets of NASA and Wikipedia [30] are utilized to emulate the Web requests. Using AutoScaleSim, performing long experiments (e.g., 4 weeks), in just few seconds is realized. The simulation duration is customizable by the *SIMULATION_LIMIT* variable in the *ExperimentalSetup* class. Another variable is *CLOUDLET_LENGTH* representing the MIPS (Million Instruction Per Second) each cloudlet has to be executed by VMs. This variable presently receives a fix value for NASA dataset (e.g., 5000) because of unavailability of actual length. However, we ran real experiments in OpenStack platform and captured the requests length for each HTTP request in the Wikipedia datasets. Hence, when using the Wikipedia datasets to generate the workload, AutScaleSim assigns the actual length to each cloudlet after being normalized to an acceptable rate. Overall, *SIMULATION_LIMIT* and *CLOUDLET_LENGTH* along with *PES_NUMBER* and the dataset file can be customized for the end-user entity in the *ExperimentalSetup* class.

4.2. Application provider entity

The Application Provider entity simultaneously performs two main activities: resource auto-scaling and load managing, each of which is implemented in a distinct package of codes (see Fig. 2). To give a basic idea, this entity acts as a broker and has two major methods for managing resources (*autoScaling*) and loads (*loadManager*). It works with the *coordinator* that is in charge of controlling the simulation process.

4.2.1. Auto-scaling

When running AutoScaleSim, the *ApplicationProvider* class acts on initializing the auto-scaling mechanism by creating a static instance for each phase of MAPE (Monitor, Analyze, Planner and Executor classes) and subsequently each class instantiates its associated history class to record the data. In run-time, the *autoScaling* method of *ApplicationProvider* class calls all four phases, which are implemented in *autoscaling* package, and is responsible for controlling this process. For extendibility, each phase is implemented in an isolated class, which will be elaborated in the following paragraphs.

Monitoring: the main method of the Monitor class is *doMonitoring* which is called to gather the essential information from the cloud environment. The gathered data are categorized into VM-, SLA- and end-user-related monitoring data. There are three separate methods for collecting monitored data in Monitor class: (1) *vmStatus*, (2) *slaStatus*, and (3) *endUserStatus*. These methods store the gathered data into a new instance of *monitorVmHistory*, *monitorSLAHistory*, and *monitorEndUserHistory*, respectively, by extending the History class, which plays the K role in MAPE-K.

Precisely, the *vmStatus* method collects low-level parameters such as CPU utilization, CPU load, the number of VMs (initializing, running, and quarantined), running cloudlets, VMs' configuration type (micro, small, medium, or large), VMs' purchasing type (reserved, on-demand, or spot), and throughput using *cpuUtil*, *cpuLoad*, *vms*, *initialingVMs*, *runningVMs*, *quarantinedVMs*, *runningCloudlet*, *vmsConfig*, *vmsPurchase*, and *throughput* variables, respectively. It is noticeable that the only deemed resource in CloudSim to evaluate is CPU. This is because CPU is the most demanded resource for many applications. However, related classes for assessing other resources (i.e., memory, disk space, and network/bandwidth) are provided in AutoScaleSim. The *slaStatus* method collects sensors' high-level parameters as response time, delay time, SLA violation (numbers, percent, and seconds), canceled cloudlets, finished cloudlets, and failed cloudlets using *avgResponseTime*, *avgDelayTime*, *slavNumbers*, *slavPercent*, *slavSeconds*, *cloudletCancelled*, *cloudletFinished*, and *cloudletFailed* variables, respectively. The *endUserStatus* method collects sensor's high-level parameters as the number and length of users' request using *requests* and *requestsLength* variables, respectively. Having collected and stored all monitored data, *doMonitoring* method acts on resetting the sensors' value by calling the *resetMonitorSensors* method.

Analyzing: The Analyzer class is ran to make the monitored parameters further accurate. The Analyzer, by calling its *doAnalysis* method, (1) gets monitored data, (2) analyzes them, and then (3) stores the analyzed parameters' value into the *analyzerHistory* class. The *doAnalysis* method can analyze tens of parameters. Each parameter is analyzed by calling a corresponding method; for example, to analyze CPU utilization, the *ANLZ_CPUUtil*() method is called. Within the analyzing methods, there is a *Switch Case* to decide on which algorithm should be used for analyzing of that parameter. The implemented algorithms are simple (just picking the latest monitored data for this parameter), moving average, weighted moving average, weighted moving average by Fibonacci numbers, and Single Exponential Smoothing. At the beginning of the experiment, in the *ExperimentalSetup* class, the analyzing algorithm for each parameter (ten by default) is defined.

When all of ten monitoring parameters in *doAnalyzing* method gathered their values, a new instance of *AnalyzerHistory* class is created to receive such values and produce a new record of analyzer history to be used by a planner.

While leveraging machine learning methods to analyze parameters is extremely effective [31], a widely used package, *neuroph* [32], is embedded in AutoScaleSim that enables developing artificial neural networks. Such solutions have been implemented in the earlier versions of this simulator, where an RBF (Radial Basis Function) neural network was used to analyze request patterns [17].

Note that there are two analyzer-related variables in *ExperimentalSetup* class that must pick their values. Firstly, the integer variable *timeWindow* denoting how many recorded history items should be considered when using time-series modalities like moving average and weighted moving averaged method. Secondly, the double type variable *sESAlpha* indicating the value of alpha index when using Single Exponential Smoothing method, which can be between 0–1.

Planning: The Planner receives the analyzed parameters from the analyzer part of the knowledge-base (i.e., *AnalyzerHistory*), and tries to decide about the application adaptation. Planner takes a detailed decision by calculating an appropriate value for its fourfold parameters: (1) scaling decision using *plannerDecision*, (2) the step-size using *plannerStepSize*, (3) the type of renting using *purchaseType*, and (4) the configuration type of the needed VM using *configurationType* variables (see Fig. 2).

Precisely, the *plannerDecision* value can be *PLANNER_DO_NOTHING*, *PLANNER_SCALING_UP*, or *PLANNER_SCALING_DOWN*; the scaling type is horizontal. The *plannerStepSize* by default equals to one, meaning that one VM will be added/removed by this decision. The *purchaseType* can be *VM_PURCHASE_ON_DEMAND*, *VM_PURCHASE_RESERVED*, or *VM_PURCHASE_SPOT*; the current version of AutoScaleSim considers the on-demand pricing model, on which researchers focus mostly [5]. The *configurationType* can be *VM_CONFIG_T2MICRO*, *VM_CONFIG_T2SMALL*, *VM_CONFIG_T2MEDIUM*, or *VM_CONFIG_T2LARGE*; note, such VMs configuration was selected based on *T2* Instances type provided by Amazon EC2 [33].

Since there are diverse approaches to planning, the abstract class, (i.e. the *Planner*), enables researchers to perform extendibility. Approaches include: rule-based, analytical modeling, and reinforcement learning. To make the auto-scaling mechanism easy to run, we needed to implement the planner at least once, so a rule-based planner, which has gained credence in the literature [4], was developed. A Rule-based planner acts as a rule-engine which can accommodate multiple rules such as CPU-related or response time-related. This rule-engine can perform independently in case appropriate input and outputs are provided. Technically, the planner in the current version of AutoScaleSim is ran by calling *doPlanning* method of *PlannerRuleBase* class, overriding the one in the *Planner* class. The *doPlanning* method functionality is to (1) get analyzed parameters, (2) set aforementioned planner's variables, (3) create a *PlannerHistory* instance to store such variables, and (4) add the instance as a new record to the *PlannerHistory* list. The main action is setting the planner's parameters. Firstly, the pre-defined rule is selected in this method using a Switch Case. To make AutoScaleSim customizable, disparate widely-used rules are already implemented in *PlannerRuleBase* class as; they are *RESOURCE_AWARE* [29], *SLA_AWARE* [16], *HYBRID* [17], *UT–1Al*, *UT_2Al*, *LAT_1Al*, and *LAT_2Al* [34]. One should indicate the preferred rule by attributing a corresponding value to the *ScalingRule* variable in the *ExperimentalSetup* class. The scale-up

and scale-down thresholds for CPU utilization and delay time are also set by attributing preferred values to *cpuScaleUpThreshold*, *cpuScaleDownThreshold*, *delayTimeScaleUpThreshold*, and *delayTimeScaleDownThreshold*, respectively. CPU utilization threshold is a number between 0–100 and delay time is a number in seconds or millisecond.

Executing: The *Executor* class is ran to execute the taken decision. Executor class receives the planner's decision by retrieving the planner's latest stored history and attempts to execute the decision.

To begin with, the *execution* method of the *Executor* class is first called. The method overall functionality is to get planner's output, set executor's output, and save the output into the associated history (the *ExecutorHistory* class). The method finally returns a list of actions to the *ApplicationProvider* class to be performed. This delegation of authority is due to the fact that requesting a VM from the cloud provider in CloudSim demands cross-entity communication which is applicable only through entities. In the real world, this action is done using APIs.

To provide customizability, executor-specific variables are defined in the *ExperimentalSetup* class to customize executor's functionality. The variables include *executorType*, *surplusVMSelectionPolicy*, *COOLDOWN*, *maxOnDemandVm*, and *minOnDemandVm*. The *executorType* denotes the executor type (*simple* or *Suprex*) to be utilized. The *surplusVMSelectionPolicy* variable comes to play when the *execution* method aims to release a surplus VM. In this situation, the *surplusVMSelectionPolicy* acts on selecting a VM as the surplus. A diversity of policies has been developed in AutoScaleSim for this policy; they are *Random*, *theOldest* [35], *theYoungest* [16], *CloudletAware*, *LoadAware* [17], *CostAware_Simple* [17], and *CostAware_Professional* [27]. To implement a new algorithm for surplus VM selection, first, the code should be implemented in the *SurplusVMSelection* class as a new method, then its name as a new item of *SurplusVMSelectionPolicy* enum variable should be defined in the *ExperimentalSetup* class and also the policy method of *SurplusVMSelection* class. The *COOLDOWN* variable denotes the number of minutes the auto-scaler must wait after a scaling decision. The Cool-down time, or the calm period, is an approach to undermining overhead and to oscillation mitigation [5]. The latest variables are *minOnDemandVm* and *maxOnDemandVm*; they are limiting the minimum and maximum number of VMs that the executor is allowed to release/rent from cloud provider, respectively. For instance, if *maxOnDemandVm* value equals 10, it means that if the number of rented VMs at the time is 10 (currently under the authority), the scale-up executions are denied until a VM is released. This variable is to preserve auto-scaler from destructive actions, resulting in inessential resource provisioning and wastage of cost. The *maxOnDemandVm*'s value have to be in accordance with defined datacenters' capacity in the *ExdperimentalSetup* class as well, so if quite a few VMs are required, new datacenter entities can be added.

4.2.2. Load manager

The *loadManager* in the *ApplicationProvider* class is responsible for receiving, admitting and dispatching both new and canceled Web requests (see Fig. 2). To fulfill these duties, there is a package of codes, named *loadManager*, containing the *LoadAdmission* class for admitting newly received requests and the *LoadBalancing* class for dispatching. The cloudlet cancellation might occur when a VM containing cloudlets has already been released by an auto-scaling mechanism, thereby moving the cloudlets back to the *LoadBalancing* class for re-dispatching. The extendable abstract of the *LoadBalancing* class is presently extended by a round-robin load balancer as *LoadBalancingRoundRobin*. New load balancing algorithms can be implemented as well, simply by overriding the *dispatchingNewRequests* and the *dispatchingCanceledRequests* methods.

4.3. Cloud provider entity

The *Cloud Provider* entity provides the APs with the requested resources in the form of VMs in a pay-as-you-go pricing model. The proposed pricing model in AutoScaleSim similar to what Amazon EC2 offers, that is, billing cycles which are derived from the used hours. The partial hour usage is also considered as one hour. This entity, furthermore, manages its data centers, employs scheduling policies in VMs and also allocates resources to the requested VMs. Overall, the *Cloud Provider* entity is the original *DataCenter* entity of CloudSim, augmented by features within the VM class indicating the operation status, that is, whether it is in *requested*, *started*, *quarantined* or *destroyed* condition), lifetime and cost status.

Technically, the process of allocating resources to the VM, launching the VM, and deploying the application on the VM is rather time-consuming and is called the Start-up Delay. This delay for a VM lasts from the time it has been requested by an AP to the time it is being fully instantiated. According to various studies [36,37], this delay is inherent in the cloud and is prone to affect the performance of applications; hence, it should be inevitably taken into account in simulation; an aspect that is currently either neglected [38] or deemed static [6] in all other available solutions. Unlike others, AutoScaleSim supports both static and dynamic delay for initialing of a VM. The dynamic delay time is determined by the factors such as VM size, time of the day, a datacenter's location, and the number of requested VMs [37].

5. Simulation process in AutoScaleSim

The modeling and simulation process performed in AutoScaleSim for running and auto-scaling Web applications in cloud is elaborated in this section, followed by the directions for extending AutoScaleSim

Fig. 3 shows that the simulation process starts with renting initial VMs for hosting the application. Afterwards, the end-user starts submitting emulated web requests to the application and the VMs are responsible for processing the requests. The Load Manager method of the AP class continuously simulates this process. In parallel with load management, the auto-scaling mechanism regularly performs resource reconfiguration. Finally, once the simulation is finished, obtained results are printed in the terminal.

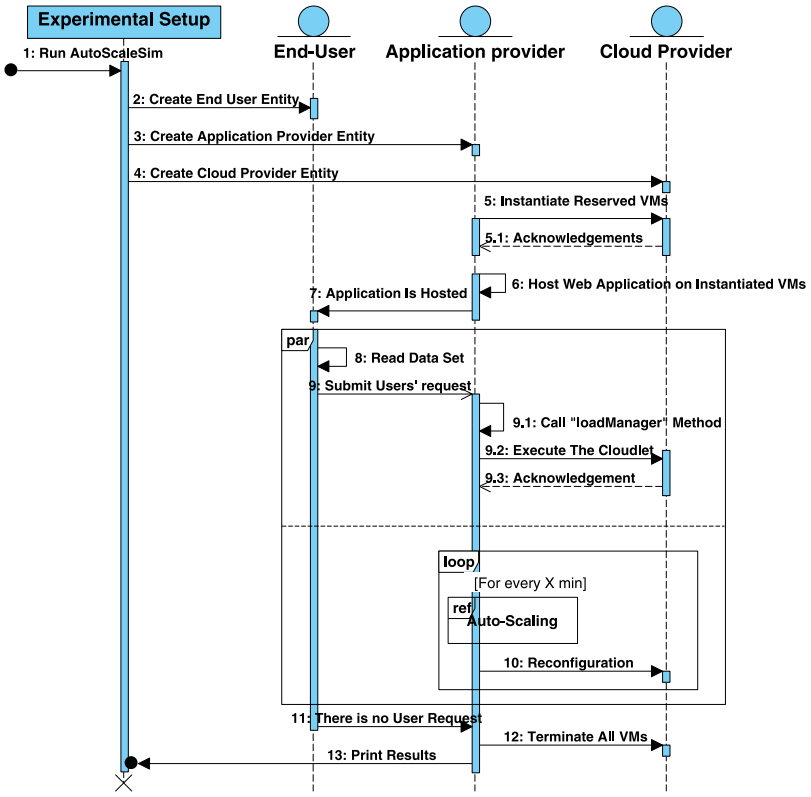

Fig. 3. The modeling and simulation process performed in AutoScaleSim for running and auto-scaling Web applications in cloud.

Taking a close look at the auto-scaling process (see Fig. 4), the monitor starts getting information from cloud parameters such as VMs and load balancers. We assume that the cloud provider provides APs with the low-level access to parameters such as CPU utilization, VMs' lifetime, and the number of running VMs. The collected parameters are added to monitor's history records. Then, it triggers analyzing the effective parameters to help the planner to make decisions. To this end, the analyzer needs monitored parameters (either the last stored item or a set of them). Afterwards, the analyzer adds the analyzed results as a new record into its history record. Next, the planner gets that result, makes a decision (scale-down, scale-up or do nothing) and adds its decision to the corresponding history. Lastly, the executor performs the execution of planner's decisions. The executor sends the decision to the application provider to be executed as this action requires cross-entity communication.

5.1. Algorithmic perspective

The process of auto-scaling is demonstrated in Algorithm 1, representing the pseudo-code for *autoscaling* method of *ApplicationProvider* class which plays the central role in AutoScaleSim. Briefly, this method continuously sends and receives events to manage the execution of each phase of MAPE and coordinates continuity of the simulation. Each event, containing parameters (such as *receiver*, *delay*, *tag*, and *data*) calls a specific phase of the auto-scaling. Having received the event, the *data* (line 1) denotes which phase of the auto-scaling must be evoked through a Switch Case statement (line 2–36).

The first receiver is *VmsSynchronization* where the latest VMs' status reports are updated (line 3); here, the latest status of all VMs is sent to the *ApplicationProvider* by calling the *vmsSynchronization* method, before triggering the monitoring, making the monitoring parameters accurate. Then, monitored parameters are obtained by calling the *doMonitoring* of the *Monitor* class, if the event *data* refers to monitor (line 5). The cool-down time is also decreased by 1 min (line 6). The next action is conditional; if the next auto-scaling epoch is to be done at this time (line 7), the auto-scaling method sends an event to itself (line 8), calling analyzer. To negate the auto-scaling overhead, it is defined that monitoring is done every one minute, while full auto-scaling is done every X minute; the X equals the value for customizable variable *scalingInterval*. If auto-scaling epoch is not required, then if the leveraged executor is *SUPREX* (line 9), the *QuarantinedVMsUpdater* part of *SUPREX* must be called (line 10) to check the latest status of quarantined VMs (for more information refer to the work [27]). Also, there is a case in calling this part to perceive whether any quarantined VM must be released (line 28–34). Finally, if the executor is *SIMPLE*, it calls the *coordinator* (line 12).

To run full auto-scaling, if it turns to the analyzing phase (line 14–16), the *doAnalysis* method of *Analyzer* is called, and then, an event calling the planner is created. If the *receiver* is planner, the planner case is ran (line 17–19), where the *doPlanning* method

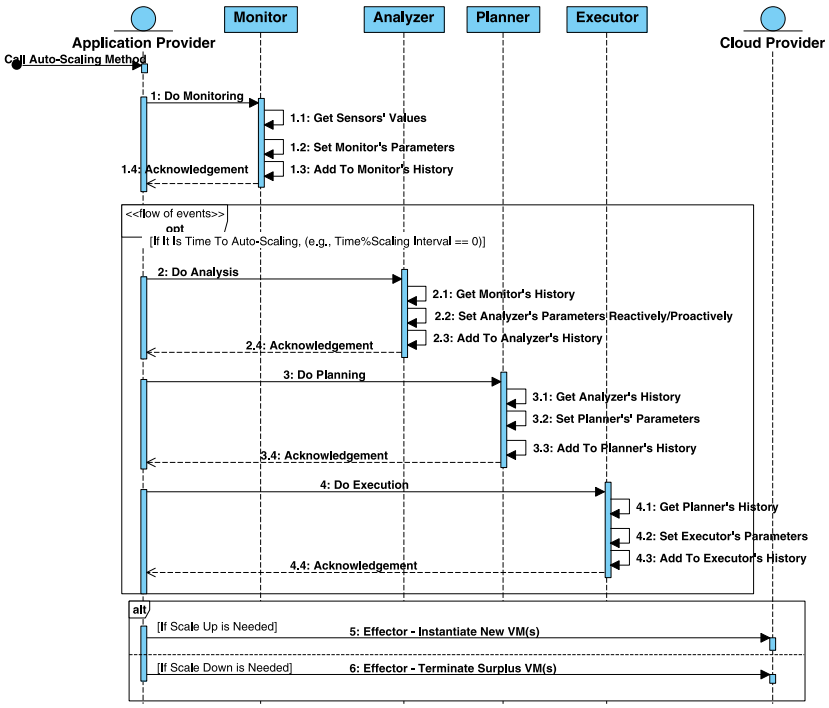

Fig. 4. The modeling process of auto-scaling mechanisms in AutoScaleSim.

of *Planner* class is called and followed by an event to create and send the executor. For case in which the executor is run (line 20–27), at first, the *execution* method of the *executor* is called. Afterwards, as per the prepared scaling action by execution, either the effector performing scale-up action, or the effector performing scale-down action is called. There are two independent if-then conditions because it is possible that none of those actions are required. Then, an event calling the *coordinator* is created and sent to the *autoscaling* method itself. The last case (line 35) calls the *coordinator* method to examine the continuously of the simulation and to schedule the next round of performing auto-scaling. This continuous process is stimulated until the *coordinator* perceive that the *SIMULATION_LIMIT* is met and acts on terminating the simulation.

5.2. Extending AutoScaleSim

Extending a particular phase (monitoring, analyzing, planning or execution) or the whole mechanism is feasible in AutoScaleSim. In the following the instruction for doing so is provided.

Monitor class is extendable, either to add some parameters into the existing methods (e.g., SD for CPU load) or to add a new method (e.g., *NewStatus*). In case of a parameter, it should be added into the corresponding monitor history class enabling the analyzer to read. In case of adding a particular method such as *EnergyStatus*, adding a history class such as *MonitoringEnergyHistory* is required to save the records of energy-related monitored parameters.

The Analyzer class is extendable in terms of both analyzing parameters and algorithms. To extend parameters: (1) a variable is defined in *doAnalysis* method as *newParameter* to be considered when running the analyzer phase. This parameter should be implemented by adding a method into the analyzer class (e.g., *ANLZ_newParamter*). This method uses a Switch Case to indicate the appropriate analyzing solutions for the newly added parameters. Solutions such as instant analyzing, moving average and WMA are already available in AutoScaleSim. The solution selection is set using *ExperimentalSetup* Class using the *analysisMethod* array. Hence, the newly implemented parameter should be included in the array as a new item. On the other hand, to implement new algorithms to analyze CPU utilization, for example, simply a new case to the Switch Case statement of *ANLZ_CPUUtil* needs to be added.

Planner is extendible in terms of implementing new rules and even new planner classes. To implement a rule: (1) a new method (e.g., *rule_NEW*) containing a new algorithm is coded; (2) a new case is added to the Switch Case statement in *doPlanning* method; and (3) the case's value must be added to the *ScalingRule* variable in the *ExperimentalSetup* class to be in the loop. To implement a new standalone planner, the inherited sub-class must override the *doPlanning* method, so that it can assign a value to the fourfold planner's parameters. If new parameters are to be implemented, the parameters must be defined in the *PlannerHistory* class as well.

Executor can either host new parameters *simple* and *Suprex* oe be extended to a new one. For the latter, a new executor class is to inherit the abstract executor class. The inherited class needs to (1) override the execution method, (2) add the name of the executor to *ExecutorType* variable in *ExperimentalSetup* class and (3) add new executor-related parameters to the associated history class, if added.

Algorithm 1: The Auto-scaling process from algorithmic perspective: *autoscaling* method

```
Input: A SimEvent object, ev (events contain receiver ID, delay, tag, and data)
receiver ← ev.data
switch receiver do
    case VmsSynchronization do Call vmsSynchronization method; break;
    case Monitor do
        Call Monitor.doMonitoring method; break;
        Executor.remainedCoolDownTime − = 1 minute;
        if clock % (scalinginterval × 1 minute) == 0 then
            Send an event to autoScaling method with the data 'Analyzer';
        else if executorType == SUPREX then
            Send an event to autoScaling method with the data 'QuarantinedVMsUpdater';
        else
            Send an event to autoscaling method with the data 'Coordinator'; break;
        end
    case Analyzer do
        Call Analyzer.doAnalysis method;
        Send an event to autoScaling method with the data 'Planner'; break;
    case Planner do
        Call Planner.doPlanning method; break;
        Send an event to autoscaling method with the data 'Executor'; break;
    case Executor do
        Call Executor.execution method and return the decision;
        if decision is Scale-up then
            Call effectorScaleUp method and hand in the list of required VM(s) to be provisioned;
        if decision is Scale-down then
            Call effectorScaleDown method;
            Wait until the acknowledge of releasing the VM(s) is received;
        Send an event to autoscaling method with the data 'Coordinator'; break;
    case QuarantinedVMsUpdater do
        Call Executor. quarantinedVMsUpdater method;
        if there is/are VM(s) to be released then
            Call effectorScaleDown method and wait until receiving acknowledge
        else
            Send an event to autoscaling method with the data 'Coordinator'; break;
        end
    case Coordinator do Call coordinator method to check the continuation of simulation;
end
```

6. Validation and performance evaluation

The aim of this section is to validate and evaluate the performance of AutoScaleSim. First, the experimental setup is elaborated in details. Then, the performance metrics provided by the simulator are explained. The validation of the simulator is performed using extensive use cases utilizing the Wikipedia traces, by drawing an analogy with our real cloud experiments using an OpenStack cluster in the Validation Section. Finally, a set of use cases to evaluate the effectiveness and correctness of AutoScaleSim utilizing the NASA web traces are studied in the Performance Evaluation Section.

6.1. Experimental setup

The *ExperimentalSetup* class in the main package of AutoScaleSim is where the developers can assign specific values for experiment's parameters to customize their simulation. Parameters are related to three entities: *End-User*, *Application Provider* and *Cloud Provider* and are listed in Table 2. The high degree of customizability is obvious with the wide range of applicable values for parameters (the Proposed Values column in Table 2) mainly due to the diverse algorithms already developed in AutoScaleSim.

To set up End-User entity, firstly, the simulation period can be set by the *SIMULATION_LIMIT* parameter, indicating how many days/hours the simulation of requests to the website will last. For example, if the NASA Web traces is selected, the simulation of up to 1 month is applicable, see Fig. 5. The NASA dataset contains the logs regarding access to NASA Web servers in July 1995, experiencing variable requests across the week. Another dataset is the 4-hour Wikipedia access traces (see Fig. 6) collected in September 19th 2007, having by far more intensity. The Wikipedia traces contain the data required for each HTTP request, modeled as a cloudlet, according to which we assign the corresponding *CLOUDLET_LENGTH*. The number of instructions and required processing elements per HTTP request is denoted by setting the *CLOUDLET_LENGTH* and *PES_NUMBER*, respectively. The data size is also involved when we model the network delay time for each cloudlet. We assume that network delay is linearly correlated to that of the data size. Hence, a cloudlet's response time is a matter of data size and network delay. The NASA traces lack the data size, so we set the *CLOUDLET_LENGTH* parameter as a constant value. In a real environment, Web requests to a Website have a timeout time, so *cloudletTimeout* parameter is designed to play this role.

The parameters related to Application Provider entity are twofold: (1) load balancing and (2) auto-scaling. The load balancing (*loadBalancing* class) is implemented based on a Round Robin [39] manner by default and task scheduling method is TimeShared [38].

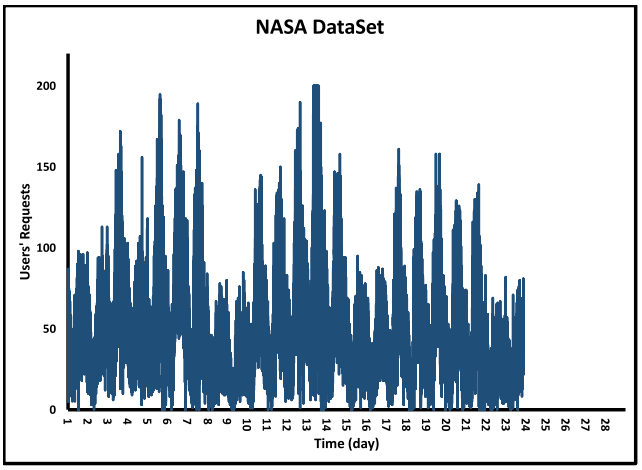

Fig. 5. The NASA dataset used to emulate the users' requests to the Web application, the requests from 1st to 28th of July, 1998. The first day is Saturday.

Turning to the auto-scaling parameters for each phase of MAPE-K, first of all, the number of initial VMs is indicated by *initialVMs*. The scaling interval (*scalingInterval*) can be set on the scale of minutes. According to various studies [16,37], it is more efficient if we assign it between 2 to 15 min to help agile alleviation of application maladaptation. Despite that, the monitor is collecting observed data every one minute, enabling fine-grained analyzing. The *slaContractOnDelayTime* parameter is also the acceptable delay time for running a cloudlet, meaning that if the delay time for a cloudlet went above this limitation — this is called an SLA violation.

Analyzing phase needs to know which type of analysis (*analysisMethod*) be carried out (simple or complex). In case Single Exponential Smoothing algorithm is selected, the alpha parameter for this method can be set using *sESAlpha*. In time-series analyzing methods, the number of monitored items involving for parameters (e.g., CPU utilization) can be set using the *timeWindow* variable.

The developed planner is Rule-based [4], so the rule can be set by the *scalingRule* variable. The upper and lower thresholds for parameters, such as CPU utilization, involved in the decision are also available to be set. If the RESOURCE_AWARE rule is selected, the planner decision is derived from CPU utilization thresholds. If SLA_AWARE rule is selected, the decision is made according to delay time thresholds, and if HYBRID rule [27] is selected, the planner considers both.

Finally, the execution phase has some vital parameters to be set. The type of executor (*executorType*) which can be SIMPLE [17] or *SUPREX* [27], the surplus selection policy (*surplusVMSelectionPolicy*) for which there are a variety of algorithms in AutoScaleSim, the cool-down time [36] (*COOLDOWN*), and the limitation to the number of VMs (*maxOnDemandVm* and *minOnDemandVm*) that the auto-scaling mechanism is authorized to provision and deprovision.

The Cloud Provider is inherited from CloudSim with the following upgrades: realizing the start-up delay time for VMs (*startUpDelayType*) to model a static or dynamic start-up delay, considering vital factors such as OS type, time of the day, the number of VMs, VMs configuration type, and servers' geographical location. The BASIC_STARTUP_DELAY parameter plays two roles: (1) the VM start-up delay in static mode and (2) the basic amount of delay in dynamic mode, which is increased by mentioned factors.

Finally, the results are exported into CSV files by the *report* parameter through *M_VM*, *M_SLA*, *M_USER*, *ANALYZER*, *PLANNER* and *EXECUTOR* values indicating the monitored VMs, monitored SLA, monitored end-users, analyzed, planned and executed activities, respectively. Besides that, an audit of the auto-scaling mechanism is continuously printed on the console by the *LogAutoScaler*.

6.2. Performance metrics

AutoScaleSim collects several metrics, categorized in Fig. 7, to evaluate major objectives: scalability, elasticity and efficiency [40]. Scalability refers to the responsiveness of the auto-scaler (e.g., proportionally adding resources) to the increasing workload; elasticity is a matter of time and frequency which refers to the time to make the application adapted again and the extent at which the mechanism is able to avoid resource under- and over-provisioning; and efficiency is a matter of effective utilization of resources which refers to the amount of resources which are actually utilized or wasted [41]. The following metrics measures the above-mentioned objectives:

CPU Utilization is calculated as the average and standard deviation of all VMs' utilization [27]. Note that higher average CPU utilizations do not always lead to more efficient auto-scaling; because, a high level of CPU utilization might be a repercussion of load accumulation, stemming from inefficient load or resource management [27].

VM Load is a measurement for the number of processes using or waiting for the CPU cores at a single point in time.

VM Lifetime, a measure of efficiency, shows how many minutes, on average, the rented VMs are running [38]. The higher, the more efficient the auto-scaling mechanism will be.

Max. Run VM a measure of efficiency, shows the maximum number of running VMs at the maximum load. The lower, the more efficient and fluctuation-proof it is; although in some cases this resistance may even usher in SLA violation [27].

Table 2
The parameters to be set at the ExperimentalSetup class for running AutoScaleSim.

Entity	Parameter	Proposed Value(s)	Default Value (NASA/Wikipedia)	Comments
End-user	SIMULATION_LIMIT	$[1–28] \times 1440$ (NASA) $[1-4] \times 60$ (Wikipedia)	$7 \times 1440/4 \times 60$	Minute(s)
	CLOUDLET_LENGTH	[2500–10000]	5000/variable	Million Instruction
	PES_NUMBER	[1–2]	2/1	No more than VMs' core(s)
	datasetType	NASA or Wikipedia	NASA/Wikipedia	
	cloudletTimeout	[30–50]	30/50	second
Application Provider	loadBalancing	LoadBalancing RoundRobin	LoadBalancing RoundRobin	
	cloudletSchedulerName	Timeshared	Timeshared	
	initialVMs	[1–5]	2/1	
	configurationType	micro, small, medium, large	medium/small	
	scalingInterval	[1–60]	10/2	minute
	slaContractOnDelayTime	[0.2–2]	1.0	second
	analysisMethod	SIMPLE, COMPLEX_MA, COMPLEX_WMA, COMPLEX_WMAfibo, COMPLEX_SES	SIMPLE	
	sESAlpha	[0.1–1] per analyzing parameter	0.2/0.1	for analyzing parameters by SES
	timeWindow	[1–60]	Equal scalingInterval/5	data effective in analyzing
	scalingRule	RESOURCE_AWARE, SLA_AWARE, HYBRID	SLA_AWARE	
	cpuScaleUpThreshold	[50–100]	70	percentage
	cpuScaleDownThreshold	[0–50]	40	percentage
	delayTimeScaleDownThreshold	[0–0.5]	$0.2/0.130 \times second$	
	ExecutorType	SIMPLE or Suprex	SIMPLE	
	surplusVMSelectionPolicy	Random, theYoungest, theOldest, CloudletAware, LoadAware, CostAware_Simple, CostAware_Professional	theOldest	
	COOLDOWN	[0-scalingInterval]	0	minute
	maxOnDemandVm	[0–40]	40/10	
	minOnDemandVm	[1–5]	1	
Cloud Provider	startUpDelayType	Static or Dynamic	Static	
	BASIC_STARTUP_DELAY	[0-15]	5/1	minute
Results	reports	M_VM, M_SLA, M_User, ANALYZER, PLANNER, EXECUTOR	–	prints reports to a CSV file

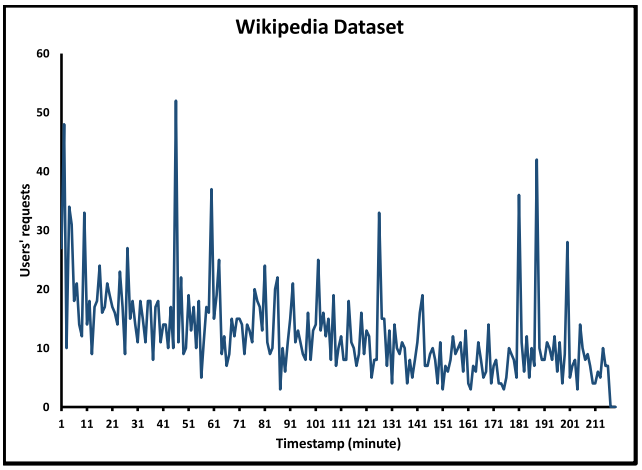

Fig. 6. The Wikipedia dataset used to emulate the users' requests to the Web application, the requests from 17th September in 2007.

Max. Run On-Demand VM is similar to Max. Run VM metric with the difference that it only considers On-Demand VMs.

Delay Time is the discrepancy between the desired and actual response time for all processed cloudlets. The less delay time, the less violation of QoS requirements.

Response Time is the time taken from the submission of a cloudlet to its completion. The average of this time for all cloudlets is computed.

Canceled Cloudlet(s) calculates the aggregate number of cloudlets canceled during the runtime and are resubmitted to the load balancer for execution.

Failed Cloudlet(s) calculates the total cloudlets reached their timeout and subsequently returned to the user with no answer.

SLA Violation, as a measure of elasticity, determines both the aggregated times (in seconds and hours) and the percentage of violation in meeting the SLA requirements in terms of response time [27,42]. A request execution is deemed as SLA violation if its execution time reaches a time higher than the expected time specified by the SLA, e.g. 1 s.

Users' requests metric counts the aggregate of received user requests during the simulation.

Analyzed CPU Util. is the average CPU utilization produced by analyzer. To evaluate the accuracy of the developed analyzer, this value can be compared to the actual CPU Utilization.

Analyzed Delay Time is the average delay time estimated by analyzer during the simulation, which can be compared to the actual delay time. Scale Up Decisions shows the sum of scale up decisions made by planner. The more the number of scale up decisions, the more provisioned resource and, hence, renting cost. Scale Down Decisions is the total number of scale-down decisions made by planner.

Contradictory Scaling Decisions is the time the planner makes a scale up decision and subsequently a scale down decision is performed, or vice versa, in two subsequent scaling epochs [4,27].

Time to Adaptation, or Mean Time To Quality Repair [40], is the average time taken from receiving the scaling decision to the time that the delay time goes below expected value e.g., 1 Second. The lower time to adaptation, the timelier reactions to variability in workload.

Provisioned VMs is a measure of efficiency, is the total rented VMs, provisioned by executor. The less provisioning, the less auto-scaling overload and the more efficiency.

De-Provisioned VMs is the total released VMs, de-provisioned by executor.

Over-/Under-provisioned VMs is the quantitative normalized metric to measure the scalability of the auto-scaler. It shows how many times the auto-scaler over-provisioned or under-provisioned the underlying resources according to the incoming workload in percentage [43]. The closer to zero, the better.

Scale Up/Down Precision is the qualitative normalized metric to measure the elasticity of the auto-scaler. It indicates the amount of surplus resources imposed by scale up and scale down actions [44]. The resources are measured at VM scale as recommended by the work.

Contradictory Scaling Actions is to measure how many times the scale up and scale down actions are performed in two subsequent scaling intervals throughout the simulation [27].

Cost of Renting VMs is the cost of renting VMs based on an hourly-based billing offered by Amazon EC2. Similar to Amazon EC2, the partial using (i.e., less that one hour) is deemed as one hour.

SLA Violation Penalty (cost), a measure of inability to maintain scalability and elasticity, is the criterion to consider the SLA violations incurred by the poor performance of the auto-scaling mechanism [17]. To measure the scalability and elasticity, the sum of the delay time in seconds in excess of the SLA contract occurred for all cloudlets is firstly calculated; then the aggregate seconds is changed into hour units. After rounding-up the number, the SLA penalty cost is calculated by multiplying the obtained number by the renting cost for a VM.

Total Cost is the aggregate of renting and SLA Violation cost, which can be a measure of efficiency. As the most comprehensive metric, it measures efficiency (i.e., efficient provisioned resources through Cost of renting VMs) and scalability and elasticity (i.e., the penalties imposed for inability in maintaining both).

6.3. Validation

The main concern about the reliability of a simulator's performance is that if the test results are aligned with the experiments in real environments. To address the validation concern, we deployed a Web application on a cloud platform using OpenStack in the DisNet laboratory at Monash University (Australia). By running experiments under rather the same conditions, we attempt to draw an analogy with our simulator. In the end-user layer, the Wikibench tool is used to send real traces of Wikipedia HTTP requests to the front-end load balancer. A Linux-based VM holding the traces and having Wikibench installed (*version 3.2*) is acting as the client. An HAProxy server (*version 1.6*) installed on another VM acts as the load balancer. In the application layer, there are Web servers (*Apache 2.0*) having the Mediawiki application (*version 1.30*) installed which are receiving HTTP requests, processing, and returning them back to the end-user. The data required for the requests are populated in an *m1.xlarge* VM (8 vCPUs, 16GB of RAM and 180GB of HDD) as the database server. The data includes roughly 7 million English Wikipedia pages imported into a MYSQL server. While the HTTP requests are sent to the Web servers, the auto-scaler component (co-located with the load balancer in the same VM) dynamically adds/removes Web servers. This auto-scaler is a Java program and its development is also based on the MAPE-K concept, code available on GitHub.[1] We already prepared a snapshot of the Web server having Mediawiki version 3.0 and LAMP server (Linux, Apache, MYSQL and PHP) installed by which the auto-scaler launches new Web servers.

[1] https://github.com/aslanpour/auto-scaling-in-openstack.

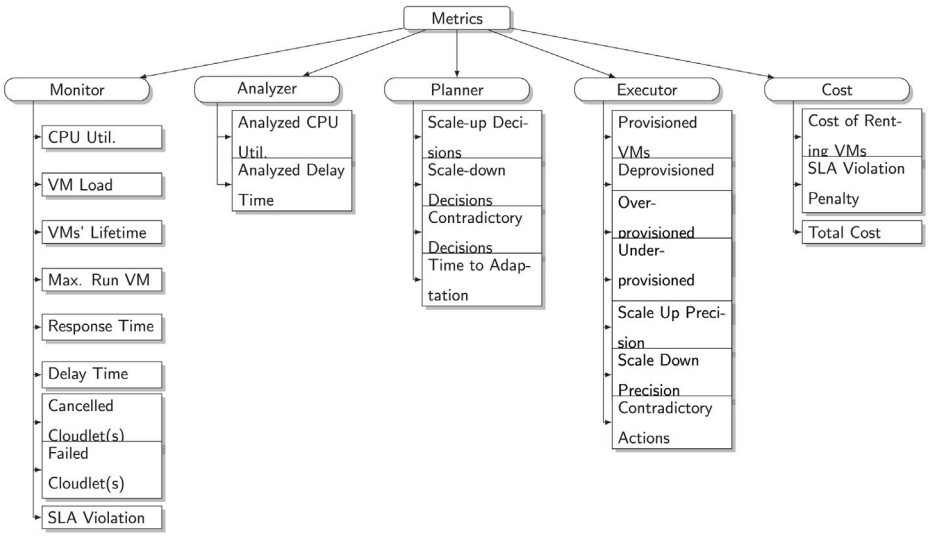

Fig. 7. Performance Metrics collected by AutoScaleSim.

We target investigating the tail latency [45], which is presently a hot topic in cloud optimization and is seldom investigated in the context of auto-scaling. Tail latency is a measure of finding outliers and the occurrence of very long latencies. The tail can be measured using percentiles, particularly high percentiles such as 90th and 95th in the distribution of response time. Having performed careful analysis of possible tail sources, we set a critical tail-related question, among others, to be investigated: Do the scaling interval (Scaling Interval validation), analysis method (Analysis method validation) and threshold tuning (Threshold tuning validation) have any impact on the tail latency? The same experiments are carried out in the simulator and real cloud to see if the simulator behaves similar to a real platform when a particular parameter is manipulated. The Wikipedia traces are utilized as the workload for both simulated and real platform experiments.

6.3.1. Scaling interval validation

The scaling interval can technically be tuned in three different ways: short-term (2 min), mid-term (4 min) and long-term (8 min). The comparison results show that, as expected, once the scaling interval becomes longer, the average and tail of latency go up. This pattern was seen harmoniously in the experimental results for the tests in the both platform and simulator (see Fig. 8). This stems from the delayed reaction of the auto-scaler to the workload variability, especially when it is growing. Assume that the incoming workload is experiencing a flash crowd. If the auto-scaler adopts the long-term scaling interval, it is highly likely that current users' requests would have to wait for the scale up decisions more. Hence, both the simulator and the real cloud platform indicate that the longer the scaling interval, the longer the tail latency will be.

Technically, we purposely employed a small number of VMs to host the application at the beginning of this experiment. By this, auto-scaler is quickly faced with request surge and has to make more scaling decisions in the rest of the experiment. Given such settings, shorter SIs are expected to quickly respond to the resource demand while longer SIs fail to do so. We observed that accumulation of requests in both simulator and real testbed linearly influences the CPU performance and response time (as parameters involved in scaling decisions). However, in the real environments, upon requests' accumulation, the VMs are getting saturated, CPU utilization goes up, memory utilization increases, more resource contention for MySql read/writes are appeared and subsequently some HTTP requests are kept in the queue for a long time. Some of these underlying issues are simplified or are not completely modeled or simulated in AutoScaleSim. For example, AutoScaleSim discards to simulate memory usage and resource contention which is inherited from CloudSim. This is why the same phenomenon (i.e., accumulation of requests) has different impact on the simulator. Limitations of AutoScaleSim and possible solutions are further discussed in Section 7.

6.3.2. Analysis method validation

The comparison results for evaluating instant and predictive analysis methods are shown in Fig. 9. Single Exponential Smoothing algorithm is used as the predictive approach and the parameter to be analyzed is the delay time. The predictive approach performs better, that is, leads to a lower latency. This is because the occurrence of temporary short-term fluctuation in the Web workloads is highly probable, so considering only the present observation, without taking recent changes into account, would make the auto-scaler impulsive. Both the simulator and the real platform managed to take advantage of the predictive method to manage the workload, proportionally. The same pattern again was achieved, with rare outliers, for the auto-scalers in both the simulator and the real platform. With accurate performance, the simulator is positively affected by delay time prediction which ended up in not only the average, but also the tail optimization. This improvement demonstrates the synergy between the simulation and the real platform experimental results.

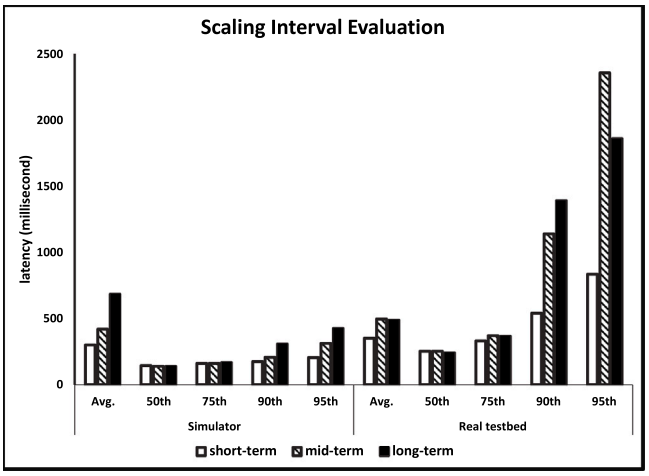

Fig. 8. The comparison of scaling intervals in simulator and real testbed.

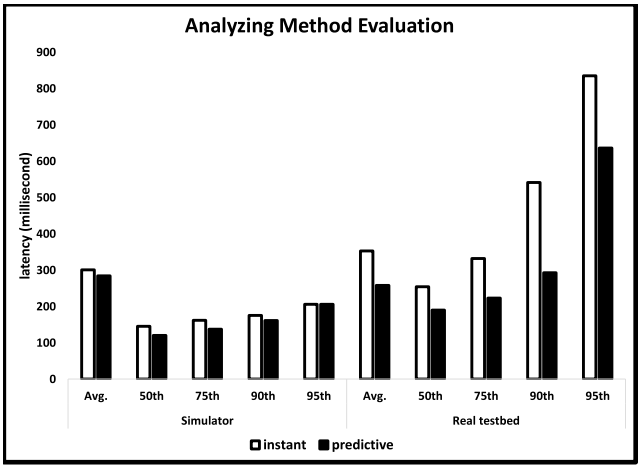

Fig. 9. The comparison of analysis methods in simulator and real testbed.

6.3.3. *Threshold tuning validation*

The threshold tuning has been a matter of concern for rule-based auto-scalers which has been studied in [46], but not in terms of the impact on tail latency. Three strategies can be adopted to reach a pair of thresholds and are categorized as: Loose, Moderate and Tight. For instance, considering CPU utilization upper 70% as the scale-up threshold would be a loose tuning strategy while considering 90% would be a tight and 80% a moderate. We conduct experiments in both the simulator and the real platform and then draw an analogy between their results. The parameter for scaling decisions is the delay time. Technically, it is expected that loose thresholds react to the workload variation promptly, by which the VMs would sustain low load accumulation and latency.

As results show in Fig. 10, tantamount to the real platform, the simulator is correctly reporting lower average and tail latency when the Loose thresholds are used. In contrast, the Tight tuning is performing worse than others in the real platform, as this strategy is reacting to the workload variation with hesitation. The same happened in the simulator as a result of applying the Tight tuning, that is, the worse performance than other strategies (see Fig. 10). Both experiments reported similar position for the Moderate strategy. The higher level of latencies for the real platform has the root to the load accumulation at the beginning of the test which then is alleviated through operating system scheduling, which is missing in simulation environments as explained in Section 6.3.1.

To give an estimation of performance precision of AutoScaleSim, the latency estimation by simulator is compared to the observed latency in the real experiments point by point, which is demonstrated in Fig. 11. This figure shows that AutoScaleSim is able to capture the real world systems behavior. Despite the variability in the beginning (which is explained in Section 7 in more details), the simulator is able to show a reasonably close performance to that of real testbed. The proximity of latency estimations in the second half of the results confirms that the simulator is able to adapt itself to the incoming workload and produce acceptable estimations for evaluated parameters, e.g., the latency. The simulator performance measured by MAPE (Mean Absolute Percentage Error) confirms the accuracy of 81% which reasonable for such variable environments. This accuracy will become even smaller when

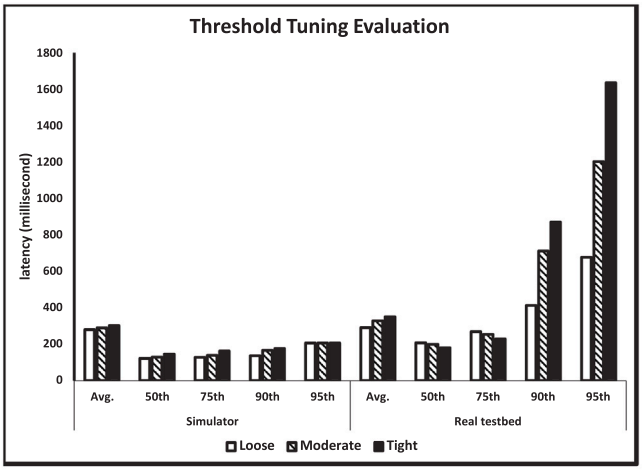

Fig. 10. The estimation performance of the simulator when compared with the real testbed threshold tuning analysis.

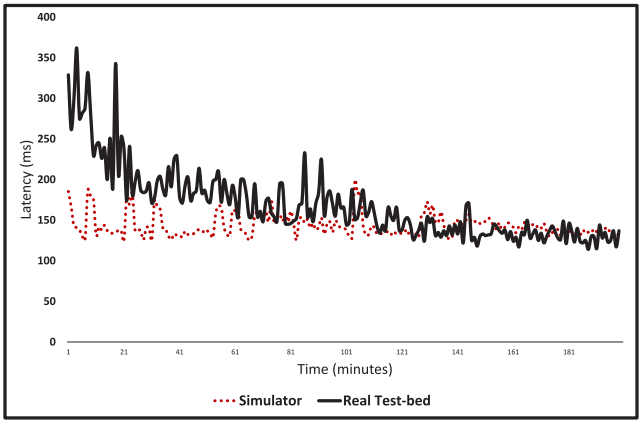

Fig. 11. The latency estimation accuracy by the simulator, AutoScaleSim.

the resource abstraction level in the simulator becomes more comprehensive. Thus, it is assumed that the experimental results in simulated environment represent the behavior of the system not the actual values.

6.4. Evaluation

Three use cases are presented here to prove the effectiveness and correctness of AutoScaleSim. The aim is to demonstrate the way researchers can use and customize AutoScaleSim. AutoScaleSim is ran by default value of each experimental parameter, shown in 'Default Value' column in Table 2. Then three predominant use cases from the literature are tested. Within each use case, the value for parameters of one particular component of AutoScaleSim is modified and others remain unchanged. The result of running each use case is compared with the default scenario in AutoScaleSim to analyze their impact and to prove the performance of AutoScaleSim.

Note that there is a technical affinity between the default test settings and that of Amazon EC2 Auto-scaling Service.[2] In the analysis phase, the *SIMPLE* analysis is selected. The rule-based planner is selected to use only delay time parameter to make its decision about resources (i.e., *SLA_AWARE* rule). The executor benefits from the simple executor (i.e., *SIMPLE* executor). In case of executing scale-down decisions, the oldest VM as the surplus VM (i.e., *theOldest* policy) comes to play. The results obtained by running this default scenario is shown in Table 3, which is then compared to the three use cases.

[2] https://aws.amazon.com/autoscaling.

```
private void rule_SLAAware(double delayTime){
    if(delayTime > delayTimeMax)
        setPlannerDecision (AutoScaleSimTags.PLANNER_SCALING_UP);
    else if (delayTime < delayTimeMin)
        setPlannerDecision (AutoScaleSimTags.PLANNER_SCALING_DOWN);
}

private void rule_HYBRID(double cpuUtil, double delayTime){
    if(cpuUtil > cpuScaleUpThreshold && delayTime > delayTimeMax)
        setPlannerDecision (AutoScaleSimTags.PLANNER_SCALING_UP);
    else if (cpuUtil < cpuScaleDownThreshold && delayTime < delayTimeMin)
        setPlannerDecision (AutoScaleSimTags.PLANNER_SCALING_DOWN);
}
```

Pseudocode 1: The pseudocode for SLA_AWARE and HYBRID rules in PlannerRuleBased class

6.4.1. Evaluating the analyzer

In this test, we improve the analyzer phase by analyzing the corresponding parameter (i.e., delay time) using a predictive and times-series-based algorithm called Weighted Moving Average-Fibonacci [16], rather than the *SIMPLE* used in the default test. The value for all parameters, therefore, coincides with what is shown in 'Default Value' column in Table 2, except *analysisMethod* whose value is set to *COMPLEX_WMAfibo*. It is expected that by using a predictive analyzer, the accuracy of decisions improves. Looking at the results in Table 3, the performance of the auto-scaling mechanism significantly increased for the majority of metrics. The most apparent improve is perceived in SLA-related metrics, where response time and also SLA violation is declined from 3.99 to 2.75 s and from 29.62% to 18.14%, respectively. Moreover, the lower the delay time was, the smaller the number of failed cloudlets. Since a predictive analyzer adopted, the decisions are provident and therefore the time to adaptation metric reduced significantly. Regarding the cost, it is clear that, despite an increase in the renting cost, ascribing to the considerable number of scaling actions, the cost of SLA violation lessened from $11.24 to $3.92. This improvement highlights how forecasting methods can be effective in QoS. However, to reduce the scaling actions, it seems that it is planner phase that should be improved, so that does not make quick decisions. In terms of CPU utilization and load, the average values reduced compared to the default test. Whereas a lower percentage of utilization and load can denote reduction in the performance, this can also be interpreted as the ability of the mechanism in alleviating the accumulation of load on VMs. Hence, such metrics seem irrational to be interpreted individually, and should be perceived as one among a group.

6.4.2. Evaluating the planner

In the default test, the *scalingRule* was set to *SLA_AWARE*, whereas in this use case the *HYBRID* rule, a more advanced planner, is utilized. This rule posits both SLA and CPU utilization to make the scaling decision, see Pseudocode 1. The *HYBRID* rule follows a more restricted decision-making: considering both CPU utilization and delay time.

Results in Table 3, ascribes improvement in scaling decisions to doubling parameters affecting decision making, from only delay time to both delay time and CPU utilization. In comparison with the default test, the incidence of scaling decisions declined from 306 to 177 and from 304 to 175 for scale up and scale-down decisions, respectively. The Max Used VM metric calculated 7 running VMs, whereas it was 8 for the default test. Consequently, the cost of renting VMs reduced notably from 29.84 to 26.16, at around 12% reduction. The lower provisioning of VMs has another impact that can be perceived in the increased CPU utilization, where lower VMs served the incoming workload, whereby the average CPU utilization increased by 4%. A repercussion of lower provisioning of VMs, however, can be perceived in the SLA situation, where delay time and subsequently SLA violation went up, stemming from not provisioning a new VM when either delay time or CPU utilization threshold is reached, not both. As a drawback of this rule, the time to adaptation is increased. The evaluation and results demonstrate that AutoScaleSim is able to manage real workloads and react to planner strategies reasonably.

It is crucial that what range of thresholds is set for scaling parameters such as delay time and CPU utilization. For instance, instead of thresholds 70% and 40% for, respectively, scale up and down decisions, one can use more restricted values at 80% and 30% to reduce the number of decisions. If rule-based planners do not satisfy needs, one can extend Planner class and implement other types of planners such as fuzzy inference, application profiling, analytical modeling, machine learning and hybrid approaches [4].

6.4.3. Evaluating the load balancing

Load balancing is another support provided by AutoScaleSim. Here a comparison between the baseline algorithm, i.e., Round Robin (Default column in Table 3), and an alternative, i.e., Random Selection Policy [4] (Load Balancing Evaluation column in Table 3) is made. The aim of this evaluation is to demonstrate how the simulator can accommodate different load balancing solutions and present corresponding functions. The Random load balancer might send incoming workload to the overloaded VMs since it is unaware of the load already distributed between VMs. On the other hand a Round Robin algorithm is expected to distribute the load between resources more evenly.

Table 3
The experimental results collected for the first three use cases in Evaluation Section.

Metric	Default (uses default values for parameters)	Analyzer evaluation	Planner evaluation	Load balancing evaluation
CPU Util. (%)—Avg (SD)	35.8 (36.33)	28.69 (32.35)	39.84 (38.42)	12.15 (14.30)
CPU Load (%)—Avg. (SD)	110 (276)	49 (114)	138 (319)	19.18 (45)
Throughout (%)	100.67	100.22	100.88	100.044
VMs' Lifetime (min.)	120	184.09	192.54	738.48
Max. Used VM	8	8	7	21
Response Time (sec.)—Avg (SD)	3.99 (4.46)	2.75 (1.76)	4.47 (5.03)	2.64 (0.87)
Delay Time (sec.)—Avg (SD)	1.99 (4.46)	0.75 (1.76)	2.48 (5.03)	0.64 (0.83)
Canceled Cloudlet	68	46	26	5
Failed Cloudlet	9264	890	12306	236
SLA Violation (%)	29.62	18.14	33.63	20.71
Analyzed CPU Util. (%)	36.25	29.2	40.75	12.26
Analyzed Delay Time (sec.)	1.6	0.82	2.23	0.62
Scale-up Decisions	306	222	177	139
Scale-down Decisions	304	220	175	133
Contradictory Decisions	289	98	100	29
Time to Adaptation (min.)	2.57	1.48	4.07	1.55
Provisioned VMs	306	222	177	139
De-provisioned VMs	304	220	175	133
Over-provisioning %	97.67	99.71	96.54	99.96
Under-provisioning %	2.21	0.27	3.19	0.04
Scale Up Precision/Penalty (Vms)	0.007	0.001	0.01	0
Scale Down Precision/Penalty (Vms)	1.79	2.21	1.56	8.46
Contradictory Actions	279	98	100	29
Renting Cost ($)	29.84	31.12	26.16	71.56
SLA Penalty Cost ($)	11.24	3.92	14	2.48
Total Cost ($)	41.08	35.04	40.16	74.04

Results in Table 3 show that the simulator is reasonably responding to the algorithms' functionalities. For instance, distributing uneven workload among VMs by Random policy resulted in over-provisioning of VMs by the auto-scaler. Our careful analysis revealed that this tendency to provision more VMs at the beginning resulted in multiple scale up decisions whereby the number of provisioned VMs reached at a high level of 21 (which never happened for Round Robin algorithm). This over-provisioning is highlighted by VMs' Lifetime metric at 738 min which is far more than other experiments and by the slight increase in the over-provisioning metric. The significance of this issue for the Random policy is revealed when the Scale Down Precision/Penalty is recorded as almost 9 VMs which means on average there have been 9 additional provisioned VMs during the experiment. This number was less than 2 for the Round Robing policy. It is expected that the Random policy imposes more renting cost, which is the case by renting cost of $71.56 in comparison to only $29.84 for Round Robin policy. However, over-provisioning is expected to have user-side benefits such as lower response time. This was also fulfilled by response time of 2.64 s which is lower than that of not only the Round Robin policy but also other evaluated scenarios. As a summary, AutoScaleSim is able to accommodate load balancing policies for web applications in conjunction with auto-scaling policies.

7. Discussion and limitations

To validate AutoScaleSim, similar experiments are conducted in simulation and real environments under real workloads (i.e., Wikipedia). Observations showed that the workload management in both environments is performed proportionally. For instance, a multi-objective planner managed to reduce the provisioning overhead in both. Performance metrics such as latency and cost faced the same pattern despite severe variability in the workload. This indicates the careful abstraction of the Web application and the auto-scaler in AutoScaleSim. Multiple further use cases were investigated in AutoScaleSim to evaluate the accuracy and correctness. The investigations demonstrated that AutoScaleSim directly responses to customizations, showing its reliability for implementing and evaluating novel auto-scaling mechanisms whose development in real platforms are complex and costly.

Although, the "exact" same results are not achieved with the simulator, they comply with the objectives of auto-scaling approaches of the web applications in cloud, and generally speaking, followed the same trends. The differences can be explained as follows: Firstly, the difference is insignificant in many cases, meaning that the overall pattern for parameters under the test gave the same implication, not negatively influencing the evaluations' results. Secondly, this is inevitable since in the real platform there are latencies for the communication between tiers of the Web application which are failed to be captured by the simulator. Thirdly, we noticed that results of simulation resembles the reality more accurately when the platform is free of resource contention for any of its underlying resources (CPUs, Memory, Disk and Network). Therefore, one must pay extra attention to the simulated results when the utilization of available resources is high (e.g., network congestion). Lastly, in the real platform, there are optimization mechanisms like caching which are applicable to alleviate the flash crowd situations while the simulator lacks such complimentary

add-ons. We emphasis that the main goal of simulators such as AutoScaleSim is to provide an approximate replication of a real system operations through abstractions or simplifications.

In modern clouds, auto-scaling is becoming more of a provider issue than users' by the emergence of Function-as-a-Service (FaaS) or so-called Serverless technology. Moving from low-level services (e.g., VMs), which leave the management tasks to users, to such modern cloud services, the auto-scaling evaluations demand different developments such as estimating the service behaviour, e.g., the cost and QoS, under different workload. AutoScaleSim is highly customizable as it follows OOP principles and the well-positioned MAPE-K loop. Therefore it can be extended to model the auto-scaling of more advanced cloud services such as FaaS.

It is essential to note that virtualization of resources is no longer limited to hypervisor-based machines and containers and unikernels are exist. In simple cases, container provisioning is viable in AutoScaleSim by omitting the delay in start-up of VMs and considering the resource as a container. However, such level of abstraction would be unrealistic and demands technical investigations if more complex scenarios such as replica management and container orchestration are required to be investigated.

Currently, relevant research on auto-scalers tend more to benefit from hybrid and more dynamic solutions. Examples of such developments include: (1) combination of vertical and horizontal scaling (i.e., two-dimensional), (2) provisioning hybrid resources (e.g., VM and container), (3) combined resources (e.g., nested containers inside a VM), (4) event-driven decision making (not fixed scaling intervals), (5) fine-grained function scaling for microservices (rather than course-grained VM or container scalings), (6) hybrid analyzing (reactive and proactive together), (7) dynamic step-size for resource provisioning (e.g., more than one instance provisioning/de-provisioning per decision), and (8) multi-objective planning. AutoScaleSim is open to such consideration as it modularizes the components of an auto-scaler. For instance, a dynamic step-size approach is implemented in the planner class without the need to put a significant modification in other components of the auto-scaler. Also, AutoScaleSim is intented to model transactional and CPU-bound web applications with the support for continuous simulation of workload allocation. To employ other types of Web applications (e.g., data-bound or memory-bound), the *End-user* entity can be modified.

The present version of AutoScaleSim measures the cost according to an hourly billing cycle which is common in the literature [4] while the competitive cloud market and modern services introduce other billing models, for example, yearly-based billing for AWS Reserved Instances; hourly-based billing for AWS Dedicated Instances and Saving Plans; finer-grained per minute billing for some of Google Cloud Services and per second billing in AWS Fargate and AWS On-demand Instances. In addition, by the advent of Serverless the billing models has been revolutionized, where the allocated resource is not the only effective parameter, but the number of requests is essential as well. The billing becomes further complicated when combined services (i.e., serverful and serverless) will come to play. Therefore, AutoScaleSim might require further extensions or modifications to support other billing models.

Finally, simulators consider an abstraction of the underlying resources such as VMs, hence there always exist outliers and mismatches between simulator and real platform's results. Such mismatches have been observed in existing simulators such as NetworkCloudSim [22] and PICS [23] which are validated through real cloud platforms as well. AutoScaleSim is not an exception, and demands further consideration of technical issues to become more realistic. Our validation investigations show that consideration of resource contention, which is highly difficult to simulate, significantly affects the outcomes (e.g., scaling interval evaluation).

Overall, we believe that adding more detailed modeling of abstractions such (1) resource contention, (2) novel virtualization technologies such as containers and unikernels, and (3) supporting applications having different request's pattern can make AutoScaleSim more applicable and accurate and the incorporation of such limitations are interesting areas for future research.

8. Conclusions

We introduced a simulation toolkit called AutoScaleSim, to model auto-scaling of Web applications hosted in clouds, one of the most investigated issues in the field of resource management. AutoScaleSim was proposed and implemented as an extension to the CloudSim simulator. It provides an extendable, scalable, customizable, and accurate simulator for investigating auto-scaling mechanism in clouds. It also uses realistic workloads such as NASA and Wikipedia. We deployed a real cloud platform and used realistic workloads to carry out experiments for the careful validation of AutoScaleSim. We also discussed the way researcher/developers can customize and extend AutoScaleSim in details.

For future work, we plan to enrich features of AutoScaleSim to support both vertical and horizontal scaling. Providing support for containers and unikernels provisioning is an interesting area of development. We also make the simulator further realistic by improving the networking and caching abstraction level covering other factors such as inter-tier communication delay and caching.

References

[1] M.S. Aslanpour, S.S. Gill, A.N. Toosi, Performance evaluation metrics for cloud, fog and edge computing: A review, taxonomy, benchmarks and standards for future research, Internet of Things (2020) 100273, http://dx.doi.org/10.1016/j.iot.2020.100273.
[2] B. Varghese, R. Buyya, Next generation cloud computing: New trends and research directions, Future Gener. Comput. Syst. 79 (2018) 849–861.
[3] T. Chen, R. Bahsoon, Self-adaptive and online qos modeling for cloud-based software services, IEEE Trans. Softw. Eng. 43 (5) (2017) 453–475.
[4] C. Qu, R.N. Calheiros, R. Buyya, Auto-scaling web applications in clouds: A taxonomy and survey, ACM Comput. Surv. 51 (4) (2018) 73.
[5] T. Lorido-Botran, J. Miguel-Alonso, J.A. Lozano, A review of auto-scaling techniques for elastic applications in cloud environments, J. Grid Comput. 12 (4) (2014) 559–592.
[6] S.F. Piraghaj, A.V. Dastjerdi, R.N. Calheiros, R. Buyya, ContainerCloudSim: An environment for modeling and simulation of containers in cloud data centers, Softw. - Pract. Exp. 47 (4) (2017) 505–521.
[7] A. Bashar, Modeling and simulation frameworks for cloud computing environment: A critical evaluation, Int. Conf. Cloud Comput. Serv. Sci. (2014) 1–6.
[8] F. Fakhfakh, H.H. Kacem, A.H. Kacem, Simulation tools for cloud computing: A survey and comparative study, in: Computer and Information Science (ICIS), 2017 IEEE/ACIS 16th International Conference on, IEEE, 2017, pp. 221–226.

[9] G. Sakellari, G. Loukas, A survey of mathematical models, simulation approaches and testbeds used for research in cloud computing, Simul. Model. Pract. Theory 39 (2013) 92–103.
[10] A. Beloglazov, R. Buyya, Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers, Concurr. Comput. Pract. Exp. 24 (13) (2012) 1397–1420.
[11] N.R. Herbst, S. Kounev, A. Weber, H. Groenda, BUNGEE: an elasticity benchmark for self-adaptive IaaS cloud environments, in: 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, IEEE, 2015, pp. 46–56.
[12] W. Chen, E. Deelman, Workflowsim: A toolkit for simulating scientific workflows in distributed environments, in: E-Science (E-Science), 2012 IEEE 8th International Conference on, IEEE, 2012, pp. 1–8.
[13] C. Wang, J. Chen, B.B. Zhou, A.Y. Zomaya, Just satisfactory resource provisioning for parallel applications in the cloud, in: Services (SERVICES), 2012 IEEE Eighth World Congress on, IEEE, 2012, pp. 285–292.
[14] R.N. Calheiros, R. Ranjan, A. Beloglazov, C.A.F. De Rose, R. Buyya, CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, Softw. - Pract. Exp. 41 (1) (2011) 23–50.
[15] J. Son, A.V. Dastjerdi, R.N. Calheiros, X. Ji, Y. Yoon, R. Buyya, Cloudsimsdn: Modeling and simulation of software-defined cloud data centers, in: Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on, IEEE, 2015, pp. 475–484.
[16] M.S. Aslanpour, S.E. Dashti, Proactive auto-scaling algorithm (PASA) for cloud application, Int. J. Grid High Perform. Comput. 9 (3) (2017) 1–16, http://dx.doi.org/10.4018/IJGHPC.2017070101.
[17] M. Aslanpour, S. Dashti, M. Ghobaei-Arani, A. Rahmanian, Resource provisioning for cloud applications: a 3-D, provident and flexible approach, J. Supercomput. 74 (12) (2017) 6470–6501, http://dx.doi.org/10.1007/s11227-017-2156-x.
[18] M. Ghobaei-Arani, S. Jabbehdari, M.A. Pourmina, An autonomic resource provisioning approach for service-based cloud applications: A hybrid approach, Future Gener. Comput. Syst.
[19] E.F. Coutinho, F.R. de Carvalho Sousa, P.A.L. Rego, D.G. Gomes, J.N. de Souza, Elasticity in cloud computing: a survey, Ann. Telecommun. 70 (7–8) (2015) 289–309.
[20] M.S. Aslanpour, M. Ghobaei-Arani, M. Heydari, N. Mahmoudi, LARPA: A learning automata-based resource provisioning approach for massively multiplayer online games in cloud environments, Int. J. Commun. Syst. (2019) e4090, http://dx.doi.org/10.1002/dac.4090.
[21] I. Sriram, SPECI, a simulation tool exploring cloud-scale data centres, in: IEEE International Conference on Cloud Computing, Springer, 2009, pp. 381–392.
[22] S.K. Garg, R. Buyya, Networkcloudsim: Modelling parallel applications in cloud simulations, in: Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on, IEEE, 2011, pp. 105–113.
[23] I.K. Kim, W. Wang, M. Humphrey, Pics: A public iaas cloud simulator, in: Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on, IEEE, 2015, pp. 211–220.
[24] C. Badii, P. Bellini, I. Bruno, D. Cenni, R. Mariucci, P. Nesi, ICARO Cloud simulator exploiting knowledge base, Simul. Model. Pract. Theory 62 (2016) 1–13.
[25] A.V. Papadopoulos, A. Ali-Eldin, K. Årzén, J. Tordsson, W. Elmroth, PEAS: A performance evaluation framework for auto-scaling strategies in cloud applications, ACM Trans. Model. Perform. Eval. Comput. Syst. 1 (4) (2016) 1–31.
[26] B.-L. Cai, R.-Q. Zhang, X.-B. Zhou, L.-P. Zhao, K.-Q. Li, Experience availability: tail-latency oriented availability in software-defined cloud computing, J. Comput. Sci. Tech. 32 (2) (2017) 250–257.
[27] M. Aslanpour, M. Ghobaei-Arani, A. Nadjaran Toosi, Auto-scaling web applications in clouds: A cost-aware approach, J. Netw. Comput. Appl. 95 (2017) 26–41, http://dx.doi.org/10.1016/j.jnca.2017.07.012.
[28] Z. Cai, Q. Li, X. Li, Elasticsim: A toolkit for simulating workflows with cloud resource runtime auto-scaling and stochastic task execution times, J. Grid Comput. 15 (2) (2017) 257–272.
[29] M.S. Aslanpour, S.E. Dashti, SLA-Aware resource allocation for application service providers in the cloud, in: 2016 2nd International Conference on Web Research, ICWR 2016, IEEE, Tehran, 2016, pp. 31–42, http://dx.doi.org/10.1109/ICWR.2016.7498443.
[30] E.-J. van Baaren, Wikibench: A Distributed, Wikipedia Based Web Application Benchmark (Master's thesis), VU, University Amsterdam.
[31] T. Chen, R. Bahsoon, X. Yao, A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems, ACM Comput. Surv. 51 (3) (2018) 61.
[32] Z. Sevarac, Neuroph-Java neural network framework, Retrieved in January. https://github.com/neuroph/neuroph-master.
[33] Amazon, Amazon EC2 instance types, 2018, https://aws.amazon.com/ec2/instance-types/.
[34] E. Casalicchio, L. Silvestri, Mechanisms for SLA provisioning in cloud-based service providers, Comput. Netw. 57 (3) (2013) 795–810.
[35] Amazon, Controlling Which Instances Auto Scaling Terminates During Scale In, http://docs.aws.amazon.com/autoscaling/latest/userguide/as-instance-termination.html.
[36] M. Beltrán, Automatic provisioning of multi-tier applications in cloud computing environments, J. Supercomput. 71 (6) (2015) 2221–2250.
[37] M. Mao, M. Humphrey, A performance study on the vm startup time in the cloud, in: Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on, IEEE, 2012, pp. 423–430.
[38] R.N. Calheiros, R. Ranjan, R. Buyya, Virtual machine provisioning based on analytical performance and QoS in cloud computing environments, in: Parallel Processing (ICPP), 2011 International Conference on, IEEE, 2011, pp. 295–304.
[39] A.N. Toosi, C. Qu, M.D. de Assunção, R. Buyya, Renewable-aware geographical load balancing of web applications for sustainable data centers, J. Netw. Comput. Appl.
[40] M. Becker, S. Lehrig, S. Becker, Systematically deriving quality metrics for cloud computing systems, in: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, 2015, pp. 169–174.
[41] S. Lehrig, H. Eikerling, S. Becker, Scalability, elasticity, and efficiency in cloud computing: A systematic literature review of definitions and metrics, in: Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures, ACM, 2015, pp. 83–92.
[42] R. Almeida, F. Sousa, S. Lifschitz, J. Machado, On defining metrics for elasticity of cloud databases, 2013, pp. 1–12.
[43] S. Islam, K. Lee, A. Fekete, A. Liu, How a consumer can measure elasticity for cloud platforms, in: Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering, 2012, pp. 85–96.
[44] N. Herbst, S. Kounev, R. Reussner, Elasticity in cloud computing: What it is, and what it is not, in: 10th International Conference on Autonomic Computing (ICAC 13), 2013, pp. 23–27.
[45] J. Dean, L.A. Barroso, The tail at scale, Commun. ACM 56 (2) (2013) 74–80.
[46] F. Al-Haidari, M. Sqalli, K. Salah, Impact of cpu utilization thresholds and scaling size on autoscaling cloud resources, in: 2013 IEEE 5th International Conference on Cloud Computing Technology and Science, Vol. 2, IEEE, 2013, pp. 256–261.