J Grid Computing (2014) 12:559–592
DOI 10.1007/s10723-014-9314-7

A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments

Tania Lorido-Botran · Jose Miguel-Alonso · Jose A. Lozano

Received: 11 March 2013 / Accepted: 15 September 2014 / Published online: 11 October 2014

Abstract Cloud computing environments allow customers to dynamically scale their applications. The key problem is how to lease the right amount of resources, on a pay-as-you-go basis. Application re-dimensioning can be implemented effortlessly, adapting the resources assigned to the application to the incoming user demand. However, the identification of the right amount of resources to lease in order to meet the required Service Level Agreement, while keeping the overall cost low, is not an easy task. Many techniques have been proposed for automating application scaling. We propose a classification of these techniques into five main categories: static threshold-based rules, control theory, reinforcement learning, queuing theory and time series analysis. Then we use this classification to carry out a literature review of proposals for auto-scaling in the cloud.

Keywords Cloud computing · Scalable applications · Auto-scaling · Service level agreement

1 Introduction

Cloud computing is an emerging technology that is becoming increasingly popular because, among other advantages, allows customers to easily deploy elastic applications, greatly simplifying the process of acquiring and releasing resources to a running application, while paying only for the resources actually allocated (pay-per-use or pay-as-you-go model). Elasticity and dynamism are two key concepts of cloud computing. In this context, resources are usually in the form of Virtual Machines (VMs). Clouds can be used by companies for a variety of purposes, from running batch jobs to hosting web applications.

Three main markets are associated to cloud computing:

Infrastructure-as-a-Service (IaaS) designates the provision of information technology and network resources such as processing, storage and bandwidth as well as management middleware.

T. Lorido-Botran · J. Miguel-Alonso · J. A. Lozano
Intelligent Systems Group, University of the Basque Country, UPV/EHU, Paseo Manuel de Lardizabal, 1, 20018 Donostia-San Sebastián, Spain

T. Lorido-Botran (✉) · J. Miguel-Alonso
Department of Computer Architecture and Technology, Donostia-San Sebastian, Spain
e-mail: tania.lorido@ehu.es

J. Miguel-Alonso
e-mail: j.miguel@ehu.es

J. A. Lozano
Department of Computer Science and Artificial Intelligence, Donostia, Gipuzkoa, Spain
e-mail: ja.lozano@ehu.es

Examples are Amazon EC2 [3], RackSpace [20] and Google Compute Engine [13].

Platform-as-a-Service (PaaS) designates programming environments and tools hosted and supported by cloud providers that can be used by consumers to build and deploy applications onto the cloud infrastructure. Examples include Amazon Elastic Beanstalk [5], Google App Engine [10], and Microsoft Windows Azure [18].

Software-as-a-Service (SaaS) designates hosted vendor applications. For example, Google Apps [11], Microsoft Office 365 [17] and Salesforce.com [25].

In this work we focus on the IaaS client's perspective. A typical scenario could be a company that wants to host an application, and for this purpose leases resources from a IaaS provider such as Amazon EC2. From now on the following terms will be used:

Provider. It refers mainly to the IaaS provider, that offers virtually *unlimited* resources in the form of VMs. A PaaS provider may also play this role.

Client. The client is the customer of the IaaS or PaaS service, that uses it for hosting the application. In other words, it is the application owner.

User. This is the end user that accesses the application and generates the workload or demand that drives its behavior.

As we said previously, a key characteristic of cloud computing is *elasticity*. This can be a double-edged sword. While it allows applications to acquire and release resources dynamically, adjusting to changing demands, deciding the *right* amount of resources is not an easy task. It would be desirable to have a system that automatically adjusts the resources to the workload handled by the application. All this with minimum human intervention or, even better, without it. This would be an *auto-scaling system*, the focus of this survey.

Resource scaling can be either horizontal or vertical. In horizontal scaling (*scaling out/in*), the resource unit is the server replica (running on a VM), and new replicas are added or released as needed. In contrast, vertical scaling (*scaling up/down*) consists of changing the resources assigned to an already running VM, for example, increasing (or reducing) the allocated CPU power or the memory. Most common operating systems do not allow on-the-fly (without rebooting) changes on the machine on which it runs (even if it is a VM); for this reason, most cloud providers only offer horizontal scaling.

The auto-scaler must be aware of the economical costs of its decisions, which depend on the pricing scheme used by the provider, to reduce the total expenditure. The pricing model may include types of VMs, unit charge (per minute, per hour), etc. The auto-scaler must also secure the correct functioning of the application, by maintaining an acceptable Quality of Service (QoS). The QoS depends on two types of Service Level Agreement (SLA): the application SLA, which is a contract between the client (application owner) and end users; and the resource SLA, that is agreed by the provider and the client. An example of the former is a certain response time, and of the latter, 99.9 % availability of the infrastructure. Both types of SLA are usually mixed up, as to satisfy the application SLA, it is necessary for the provider to comply with the resource SLA. From now on, this review focuses on the application SLA that must be guaranteed to end users.

Applications hosted in cloud environments can be of very diverse nature: batch jobs, map-reduce tasks, massively multiplayer online role-playing games (MMORPGs), web applications, video streaming services, and a large etcetera. Resource allocation for batch applications is usually denoted as *scheduling* and involves meeting a certain job execution deadline. It has been extensively studied in grid environments (job schedulers) and also explored in cloud environments, but this is not the topic of this review. The scope of applicability of the auto-scaler covers any scalable or *elastic* application composed of a load balancer that receives job requests and dispatches them to any of a collection of replicable servers. Web applications, MMORPGs and video streaming services fit into this scheme. The main contributions of this review are:

- A problem definition of the auto-scaling process and its different phases.
- A classification for auto-scaling techniques, together with a description of each category and its pros/cons.
- A review of the literature about auto-scaling, organized using this classification. However, given the heterogeneity of auto-scaling approaches and testing conditions, it is *not* possible to provide an

assessment of the different proposals, alone or in comparative terms.

- A description, in an Appendix, of the different tools and environments used to implement and test auto-scalers.

Note that the management of the cloud infrastructure is out of the scope of this paper: topics such as VM placement into physical servers, VM migration, energy consumption and related problems are not discussed.

The remainder of this paper is organized as follows. Section 2 describes the general scenario of elastic applications in which to apply auto-scaling techniques. The auto-scaling process is described in Section 3. A classification criterion to organize auto-scaling proposals is introduced in Section 4. Section 5, the core of this work, contains a review of the literature on auto-scaling in the cloud. Section 6 completes this paper with some conclusions extracted from the review and an outline of future work. Furthermore, the Appendix provides details about the experimental platforms, workload generation mechanisms and application benchmarks used by different authors to test their proposal of auto-scaling algorithms.

2 Scenario: Elastic Applications

An *elastic* application has the capability of being scaled (horizontally or vertically) to make it adjust to the input, variable workload. Applications of this class are normally based on a load balancer (a dispatcher) and a collection of identical servers. In a cluster environment, those would be physical servers but, in cloud environments, servers are hosted in VMs and, therefore, the terms servers and VMs can be used interchangeably. An auto-scaler is in charge of making decisions about scaling actions, without the intervention of a human manager.

Although several applications can be considered elastic (e.g. MMORPG or video streaming servers), most of the literature is focused on web applications. These typically include a business-logic tier, containing the application logic, and a persistence or data base tier. The literature pays most of its attention to the business-logic tier, that can be easily scaled. There are some works that study auto-scaling at the persistence tier, although replicating distributed databases brings out additional issues. Our description of auto-scaling techniques is as neutral as possible, without focusing on any particular application class or tier. However, given the literature bias towards web applications, it is unavoidable to see it reflected in this survey.

Let us consider an elastic application deployed over a pool of n VMs. VMs may have the same or different assigned amount of resources (e.g. 1GB of memory, 2 CPUs,...), but each VM has its own unique identifier (it could be its IP address). The requests received by the load balancer may come from real end users or from other application. We will assume that the execution time of a request may vary between milliseconds and minutes; long-running tasks lasting hours will not be considered in the auto-scaling problem, as they rather belong to a scheduling problem.

The load balancer will receive all the incoming requests and forward them to one of the servers in the pool. It may be implemented in different ways. These are a few examples: a specific, hardware/software combination (an Internet appliance), a part of the Domain Name System, a service offered by the IaaS provider, or even an ad-hoc part of the client's application. Whichever the chosen technology, it is assumed that the load balancer has updated information about the VMs to use (the active ones): it will stop immediately sending requests to removed VMs, and it will start sending workload to newly added VMs. It is also assumed that each request will be assigned to a single VM, that will run it until completing the task associated to it. Several load balancing policies could be used, for example, random, round-robin or least-connection. In the case of heterogeneous collections of VMs, workload dispatching should be proportional to the processing power of the VMs. The revision of techniques to implement load balancers, distribute workload among VMs, control the admission of new requests, or redirect requests between VMs fall outside the scope of this survey.

Next section is devoted to the auto-scaling process: its objectives and the phases of the process.

3 Auto-Scaling Process: the MAPE Loop

The aim of the auto-scaler is to dynamically adapt the resources assigned to the elastic applications, depending on the input workload. This auto-scaler may be either an ad-hoc implementation for a particular application, or a generic service offered by the IaaS provider. In any case, the system should be able to find a trade-off between meeting the SLA of the application and minimizing the cost of renting cloud resources. Any auto-scaler faces several problems:

Under-provisioning: The application does not have enough resources to process all the incoming requests within the temporal limits imposed by the SLA. More resources are needed, but it takes some time from the moment they are requested until they are available. In case of sudden traffic bursts, an under-provisioned application may lead to many SLA violations, even to system congestion. It will take some time until the application returns to its normal operational state.

Over-provisioning: In this case the application has more resources than those needed to comply with the SLA. This is correct from the point of view of SLA, but the client is paying an unnecessary cost if the VMs are idle or lightly loaded. A certain degree of over-provisioning could be desirable in order to cope with small workload fluctuations. In general, neither a manual provision of resources, nor the one carried out by an auto-scaler, aims to keep the VMs operating at 100 % of its capacity.

Oscillation: A combination of both undesirable effects. Oscillation occurs when scaling actions are carried out too quickly, before being able to see the impact on each scaling action on the application. Keeping a capacity buffer (aiming to keep the VMs at, say, 80 % instead of 100 %), or using a cooldown period (e.g. [73]) are common approaches to avoid oscillation.

The auto-scaling process matches the MAPE loop of autonomous systems [68, 82], which consists of four phases: Monitoring (M), Analysis (A), Planning (P) and Execution (E). First, a monitoring system gathers information about the system and application state. The auto-scaler encompasses the analysis and planning phases: it uses the retrieved information to make estimations of future resource utilization and needs (analysis), and then plans a suitable resource modification action, e.g., removing a VM or adding some extra memory. The provider will then execute the actions as requested by the auto-scaler. The phases of the MAPE loop are described in more detail in the forthcoming sections.

3.1 Monitor

An auto-scaling system requires the support of a **monitoring system** providing measurements about user demands, system (application) status and compliance with the expected SLA. Cloud infrastructures provide, through an API, access to information useful for the provider itself (for example, to check the correct functioning of the infrastructure and the level of utilization) and for the client (for example, to check the compliance with the SLA).

Auto-scaling decisions rely on having useful, updated *performance metrics*. The performance of the auto-scaler will depend on the quality of the available metrics, the sampling granularity, and the overheads (and costs) of obtaining the metrics [38]. In the literature, authors have used a variety of metrics as drivers for scaling decisions. An extensive list of those metrics is provided in [59], for both transactional (e.g. e-commerce web sites) and batch workloads (e.g. video transcoding or text mining). They could be easily adapted to other types of applications.

Hardware: CPU utilization per VM, disk access, network interface access, memory usage.

General OS Process: CPU-time per process, page faults, real memory (resident set).

Load balancer: size of request queue length, session rate, number of current sessions, transmitted bytes, number of denied requests, number of errors.

Application server: total thread count, active thread count, used memory, session count, processed requests, pending requests, dropped requests, response time.

Database: number of active threads, number of transactions in a particular state (write, commit, roll-back).

Message queue: average number of jobs in the queue, job's queuing time.

Note that, when working with applications deployed in the cloud, some metrics are obtained from the cloud provider (those related to the acquired resources and the hypervisor managing the VMs), others from the host operating system on which the application is implemented, and yet others from the application itself. In order to reduce the complexity of the monitoring system, sometimes *proxy* metrics are used:

for example, CPU utilization (hypervisor-level) as a proxy of current system workload (application-level).

As stated before, auto-scaling is mostly related to the Analyze and Plan parts of the MAPE loop. For this reason, monitoring will not be further studied in this review. It will be assumed that a good monitoring tool is available, gathering different and updated metrics about system and application current state, with negligible intrusion, and at a suitable granularity (e.g. per second, per minute).

3.2 Analyze

The **analysis** phase consists of processing the metrics gathered directly from the monitoring system, obtaining from them data about current system utilization, and *optionally* predictions of future needs. Some auto-scalers do not perform any kind of prediction, just respond to current system status: they are *reactive*. However, some others use sophisticated techniques to predict future demands in order to arrange resource provisioning with enough anticipation: they are *proactive*. Anticipation is important because there is always a delay from the time when an auto-scaling action is executed (for example, adding a server) until it is effective (for example, it takes several minutes to assign a physical server to deploy a VM, move the VM image to it, boot the operating system and application, and have the server fully operational [80]). Reactive systems might not be able to scale in case of sudden traffic bursts (e.g. special offers or the *Slashdot effect*). Therefore, proactivity might be required in order to deal with fluctuating demands and being able to scale in advance.

Part of the reviewed works focus only on this analysis phase, on the way of processing the metrics to either determine the current state of the application or anticipate future needs.

3.3 Plan

Once the current (or future) state is known (or predicted), the auto-scaler is in charge of **planning** how to scale the resources assigned to the application in order to find a satisfactory trade-off between cost and SLA compliance. Examples of scaling decisions are removing a VM or adding more memory to a particular VM. Decisions will be made considering the data obtained from the analysis phase (or directly from the monitoring system) and the target SLA, as well as other factors related to the cloud infrastructure, including pricing models and VM boot-up time.

This part constitutes the core of any auto-scaling proposal and, therefore, will be thoroughly studied in Sections 4 and 5.

3.4 Execute

This phase consists of actually **executing** the scaling actions decided in the previous step. Conceptually, this is a straightforward phase, implemented through the cloud provider's API. Actual complexities are hidden to the client. Remember that it takes some time from the moment a resource is requested until it is actually available, and that bounds on these delays may be part of the resource SLA.

4 A Classification of Auto-Scaling Techniques

As the body of literature dealing with proposals of auto-scaling systems is large, we have tried to put some order into it, to better understand and compare those proposals. To that extent, we need some classification rules to group works into meaningful sets. To the best of our knowledge, there is no previous work proposing such a classification. The most closely related survey, done by Guitart et al. [61], targets the performance of general Internet applications, deployed over shared or dedicated clusters, relying on methods such as admission control and service differentiation. However, this review focuses on exploiting the elastic nature inherent to cloud systems. Manvi and Shyam [78] gather many references about resource management on IaaS environments, but put little focus on auto-scaling.

The works we have revised to put together this survey are very diverse in regards to the underlying theory or technique used to implement the auto-scaler, including the metrics or models used to decide both when to scale or what modification in resources is necessary. Each auto-scaling approach has been designed with particular goals, focusing on several application architectures or cloud providers offering different scaling capabilities. The most differentiating factor is the final goal/evaluation criteria used for a particular auto-scaler: the prediction accuracy, the compliance with the SLA (defined in many ways such as response time or availability of resources), or the cost of resources. It seems quite obvious that comparing auto-scaling approaches is not that straight-forward. For example, let us consider two proposals, one focused in short-term prediction of workloads with a distinctly diurnal pattern, and another one that tries to comply with the SLA even in the case of sudden bursts of traffic. Both auto-scaling systems may work well for their target systems, but clearly the latter approach is more prone to cause over-provisioning. Still, it would be wrong to assume that this extra cost due to over-provisioning state is enough to prefer one auto-scaling system instead of the other. Then, the target goal of the auto-scaler cannot be used as the classification criterion.

A possible grouping for auto-scalers could be done using their anticipation capabilities, arranging them in two classes: *reactive* and *proactive*. The former implies that the system reacts to changes in the workload only when those changes have been detected, using the last values obtained from the set of monitored variables; consequently, as resource provisioning takes some time, the desired effect may arrive when it is too late. Proactive systems anticipate future demands and make decisions taking them into consideration. We have chosen not to use this as a classification criterion because sometimes it is not clear whether a particular approach is purely reactive or proactive.

We decided to adopt the underlying theory or technique used to build the auto-scaler as our classification criterion. This will help the reader to better understand the basic concepts of a particular group or category, including their advantages and limitations. Most reviewed works can fit in one or more of these five groups (note that some works propose hybridizations of several techniques):

1. Threshold-based rules (rules)
2. Reinforcement learning (RL)
3. Queuing theory (QT)
4. Control theory (CT)
5. Time series analysis (TS)

Commercial cloud providers offer purely reactive auto-scaling using threshold-based rules. The scaling decisions are triggered based on some performance metrics and predefined thresholds. This approach has become rather popular due to its (apparent) simplicity: rule-based auto-scalers are easy to provide as a cloud service, and are also easy to set-up by clients. However, the effectiveness of rules under bursty workloads is questionable.

Time series analysis covers a wide range of methods to detect patterns and predict future values on sequences of data points. The accuracy in the forecast value (e.g. future number of requests or average CPU utilization) will depend on selecting the right technique and setting the parameters correctly, specially the history window and the prediction interval. Time-series analysis is the main enabler of proactive auto-scaling techniques.

There are two auto-scaling methods that rely on modeling the system in order to determine its future resource needs. This is the case of both queuing theory and control theory. Queuing theory has been largely applied to computing systems, in order to find the relationship between the jobs arriving to and leaving from a system. A simple approach consists in modeling each VM (or set of VMs) as a queue of requests in order to estimate different performance metrics such as the response time. A main limitation of QT models is that they are too rigid, and need to be recomputed when there are changes in either the application or the workload.

Control theory also relies on creating a model of the application. The aim is to define a (reactive or proactive) controller to automatically adjust the required resources to the application demands. The nature and performance of a controller highly depends on both the application model and the controller itself. As we will see, many researchers consider that this type of auto-scaling has a great potential, specially when combined with resource prediction.

Finally, the last of our categories contains proposals based on reinforcement learning. Similarly to control theory, RL tries to automate the scaling task, but without using any a-priori knowledge or model of the application. Instead, RL tries to learn the most suitable action for each particular state on-the-fly, with a trial-and-error approach. Although the absence of model and adaptability of the technique might seem appealing for auto-scaling, the truth is that RL suffers from long learning phases. The time required by the method to converge to an optimal policy can be unfeasible long.

As defined in Section 3, the auto-scaling process is mainly related to the analysis and planning phases of the MAPE loop. Some auto-scaling proposals focus on one of these phases, but most of them cover both phases. Queuing theory and time series analysis are useful in the analysis phase in order to estimate the current utilization or future needs of the application. Threshold-based rules, reinforcement learning and control theory can be used in the planning phase to decide the scaling action, and they can be combined with a previous analysis phase involving for example, time series analysis.

5 Review of Auto-Scaling Techniques

In this section we carry out a survey of the literature on auto-scaling systems, following the classification introduced in the previous section. It is organized in as many sub-sections as categories, each one starting with a description of the technique that characterizes the category, including the definition, methodologies, pros and cons. After that, we analyze a collection of papers fitting into the category, also discussing their features and limitations. This information is complemented with a set of tables summarizing the reviewed literature, one table per category. Each table row includes a synopsis of a reviewed auto-scaling paper, which includes:

1. The specific technique or combination of techniques applied
2. Whether an horizontal (H) or vertical (V) scaling is performed
3. The reactive (R) or proactive (P) nature of the approach
4. The performance metric considered (e.g. CPU load, input request rate)
5. The monitoring tool and the granularity or monitoring interval
6. Characteristics of the environment used to test the technique, including
   - The SLA considered for the application
   - The type of workload (either real or synthetic)
   - The experimental platform (simulator, custom testbed or real provider), together with the application benchmark

Note that many table entries contain a “-”. This means that the reviewed paper does not provide enough information to know or infer the corresponding piece of information. Unfortunately, this happens more often than desirable. Also, note that authors have used many different mechanisms to assess the goodness of their auto-scaler, ranging from completely synthetic simulated environments to actual production systems. For more information about testing platforms, the interested reader is referred to the Appendix.

5.1 Threshold-Based Rules

Threshold-based auto-scaling rules or policies are very popular among cloud providers such as Amazon EC2, and third-party tools like RightScale [22]. The simplicity and intuitive nature of these policies make them very appealing to cloud clients. However, setting the corresponding thresholds is a per-application task, and requires a deep understanding of workload trends.

5.1.1 Description of the Technique

From the MAPE loop (see Section 3), rules are purely a decision-making technique (planning phase). The number of VMs or the amount of resources assigned to the target application will vary according to a set of rules, typically two: one for scaling up/out and one for scaling down/in. Rules are structured like these examples:

$$\begin{aligned}&\text{if } x_1 > thrU_1 \text{ and/or } x_2 > thrU_2 \text{ and/or } \ldots\\ &\qquad\text{for } durU \text{ seconds then}\\ &\qquad\quad n = n + s \text{ and}\\ &\qquad\quad \text{do nothing for } inU \text{ seconds}\end{aligned} \tag{1}$$

$$\begin{aligned}&\text{if } x_1 < thrL_1 \text{ and/or } x_2 < thrL_2 \text{ and/or } \ldots\\ &\qquad\text{for } durL \text{ seconds then}\\ &\qquad\quad n = n - s \text{ and}\\ &\qquad\quad \text{do nothing for } inL \text{ seconds}\end{aligned} \tag{2}$$

Each rule has two parts: the condition and the action to be executed when the condition is met. The condition part uses one or more performance metrics x_1, x_2, ..., such as the input request rate, CPU load or average response time. Each performance metric has upper $thrU$ and lower $thrL$ thresholds. If the condition is met for a given time ($durU$ or $durL$), then the corresponding action will be triggered. For horizontal scaling, the application manager should define a fixed amount s of VMs to be acquired or released, while for vertical scaling s refers to an increase or decrease of resources such as CPU or RAM. After executing an action, the auto-scaler inhibits itself for a small *cooldown* period defined by inU or inL.

The best way to understand threshold-based rules is by means of an example: *add 2 small instances when the average CPU usage is above 70 % for more than 5 minutes, and then, do nothing for 10 minutes*.

5.1.2 Review of Proposals

Threshold-based rules constitute an easy to deploy and use mechanism to manage the amount of resources assigned to an application hosted in a cloud platform, dynamically adapting those resources to the input demand (e.g. [48, 54, 63, 64, 72, 81]). However, creating the rules requires an effort from the application manager (the client), who needs to select the suitable performance metric or logical combination of metrics, and also the values of several parameters, mainly thresholds. The experiments carried out in [72] show that application-specific metrics (e.g. the average waiting time in queue), obtain better performance that system-specific metrics (e.g. CPU load). Application managers also need to set the corresponding upper (e.g. 70 %) and lower (e.g. 30 %) thresholds for the performance variable (e.g. CPU load). Thresholds are the key for the correct working of the rules. In particular, Dutreilh et al. [54] remark that thresholds need to be carefully tuned in order to avoid oscillations in the system (e.g. in the number of VMs or in the amount of CPU assigned). To prevent this problem, it is advisable to set an *inertia*, *cooldown* or *calm* period, a time during which no scaling decisions can be committed, once an scaling action has been carried out.

Most works and cloud providers use only two thresholds per performance metric. However, in [64] a set of four thresholds and two durations are considered: *ThrU*, the upper threshold; *ThrbU*, which is slightly below the upper threshold; *ThrL*, the lower threshold; *ThroL*, which is slightly above the lower threshold. Used in combination, it is possible to determine the trend of the performance metric (e.g. trending up or down), and then perform finer auto-scaling decisions.

Conditions in the rules are usually based on a single, or at most two performance metrics, being the most popular the average CPU load of the VMs, the response time, or the input request rate. Both Dutreilh et al. [54] and Han et al. [63] use the average response time of the application. On the contrary, Hasan et al. [64] prefer using performance metrics from multiple domains (compute, storage and network) or even a correlation of several of them.

Note that in most cases the rules use the metrics directly as obtained from the monitor, thus acting in a purely reactive way. However, it is possible to carry out a previous analysis of the monitored data in order to predict the future (expected) behavior of the system and execute rules in a proactive way [71]. This topic will be discussed later, in the section devoted to time series analysis.

RightScale's auto-scaling algorithm [23] proposes combining regular reactive rules with a voting process. If a majority of the VMs agree on that they should scale up/out or down/in, that action is taken; otherwise, no action is planned. Each VM votes to scale up/out or down/in based on a set of rules evaluated individually. After each scaling action, RightScale recommends a 15 minute-period of calm time because new machines generally take between 5 to 10 minutes to be operational. This auto-scaling technique has been adopted by several authors ([51, 59, 73, 96]). Chieu et al. [51] initially proposed a set of reactive rules based on the number of active sessions, but this work was extended in [52] following the RightScale approach: if all VMs have active sessions above the given upper threshold, a new VM is provisioned; if there are VMs with active sessions below a given lower threshold and with at least one VM that has no active session, the idle one will be shut down.

As RightScale's voting system is based on rules, it inherits their main disadvantage: the technique is highly dependent on manager-defined threshold values, and also, on the characteristics of the input workload. This was the conclusion reached by Kupferman et al. [73] after comparing RightScale with other algorithms. Simmons et al. [96] try to overcome these problems with a strategy-tree, a tool that evaluates the deployed policy set, and switches among alternative strategies over time, in a hierarchical manner. Authors created three different scaling policies, customized to different input workloads, and the strategy-tree would switch among them based on the workload trend (analyzed with a regression-based technique).

In order to save costs, some researchers [48, 83] came up with an idea called *smart kill*. Many IaaS providers charge partial utilization hours as full hours. Therefore, it is advisable not to terminate a VM before the hour is over, even if the load is low. Apart from

Table 1 Summary of the reviewed literature about threshold-based rules

Ref	Auto-scaling Techniques	H/V	R/P	Metric	Monitoring	SLA	Workloads	Experimental Platform
[63]	Rules	Both	R	CPU, memory, I/O	Custom tool. 1 minute	Response time	Synthetic. Browsing and ordering behavior of customers.	Custom testbed (called IC Cloud) + TPC
[72]	Rules	H	R	Average waiting time in queue, CPU load	Custom tool.	–	Synthetic	Public cloud. FutureGrid, Eucalyptus India cluster
[64]	Rules	Both	R	CPU load, response time, network link load, jitter and delay.	–	–	Only algorithm is described, no experimentation is carried out.	
[48]	Rules + QT	H	P	Request rate	Amazon Cloud-Watch. 1–5 minutes	Response time	Real. Wikipedia traces	Real provider. Amazon EC2 + Httperf + MediaWiki
[52]	RightScale + MA to performance metric	H	R	Number of active sessions	Custom tool	–	Synthetic. Different number of HTTP clients	Custom testbed. Xen + custom collaborative web application
[73]	RightScale + TS: LR and AR(1)	H	R/P	Request rate, CPU load	Simulated.	–	Synthetic. Three traffic patterns: weekly oscillation, large spike and random	Custom simulator, tuned after some real experiments.
[59]	RightScale	H	R	CPU load	Amazon CloudWatch	–	Real. World Cup 98	Real provider. Amazon EC2 + RightScale (PaaS) + a simple web application
[96]	RightScale + Strategy-tree	H	R	Number of sessions, CPU idle	Custom tool. 4 minutes.	–	Real. World Cup 98	Real provider. Amazon EC2 + RightScale (PaaS) + a simple web application.
[81]	Rules	V	R	CPU load, memory, bandwidth, storage	Simulated.	–	Synthetic	Custom simulator, plus Java rule engine Drools
[77]	Rules	V	R	CPU load	Simulated. 1 minute	Response time	Real. ClarkNet	Custom simulator

Table rows are as follow. (1) The reference to the reviewed paper. (2) A short description of the proposed technique. (3) The type of auto-scaling: horizontal (H) or vertical (V). (4) The reactive (R) and/or proactive (P) nature of the proposal. (5) The performance metric or metrics driving auto-scaling. (6) The monitoring tool used to gather the metrics. The remaining three fields are related to the environment in which the technique is tested. (7) The metric used to verify SLA compliance. (8) The workload applied to the application managed by the auto-scaler. (9) The platform on which the technique is tested reducing costs, smart kill may also improve system performance: in case of an oscillating input workload, the costs of continuously shutting down and starting up VMs are avoided, improving boot-up delays and reducing SLA violations.

The popularity of rules as auto-scaling method is probably due to their simplicity and the fact that they are easy to understand for clients. However, this kind of technique shows two main problems: its reactive nature and the difficulty of selecting the correct set of performance metrics and the corresponding thresholds. The effectiveness of those thresholds is highly dependent on the workload changes, and may require frequent tuning. In order to solve the problem of static thresholds, Lorido-Botran et al. [77] introduce the concept of dynamic thresholds: initial values are set-up, but they are automatically modified as a consequence of the observed SLA violations. Meta-rules are included to define how the threshold used in the scaling rules may change to better adapt to the workload.

In conclusion, Rules can be used to easily automate the auto-scaling of a particular application without much effort, specially in the case of applications with quite regular, predictable patterns. However, in case of bursty workloads the client should consider a more advanced and powerful auto-scaling system from the rest of categories.

The main characteristics of the auto-scaling proposals based on rules and reviewed in this section are summarized in Table 1.

5.2 Reinforcement Learning

Reinforcement Learning (RL) [98] is a type of automatic decision-making approach that has been used by several authors to implement auto-scalers. Without any *a priori* knowledge, RL techniques are able to determine the best scaling action to take for every application state, given the input workload.

5.2.1 Description of the Technique

Reinforcement learning [98] focuses on learning through direct interaction between an agent (e.g. the auto-scaler) and its environment (e.g. the application as defined in Section 2). The auto-scaler will learn from experience (trial-and-error method) the best scaling *action* to take, depending on the current *state*, given by the input workload, performance or other set of variables. After executing an action, the auto-scaler gets a response or *reward* (e.g. performance improvement) from the system, about how good that action was. So, the auto-scaler will tend to execute actions that yield a high reward (best actions are *reinforced*). From now on, in this section the general term *agent* will be used, instead of *auto-scaler*.

The objective of the agent is to find a policy π that maps every state s to the best action a the agent should choose. The agent has to maximize the expected *discounted rewards* obtained in the long run:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_0^\infty \gamma^k r_{t+k+1} \qquad (3)$$

where r_{t+1} is the reward obtained at time $t+1$, and $gamma$ is the discount factor.

The policy is based on a value function $Q(s, a)$, usually called the Q-value function. Every $Q(s, a)$ value estimates the future cumulative rewards by executing an action a in a state s. In other words, it represents the *goodness* of executing action a when in state s. The Q-value function can be defined as:

$$Q(s, a) = E_\pi \left\{ \sum_{k=0}^\infty \gamma^k r_{t+k+1} | s_t = s, a_t = a \right\} \qquad (4)$$

There are many RL algorithms that can be used in order to obtain the Q-value function, but Q-learning is the most used in the literature. Typically, the $Q(s, a)$ values are stored in a lookup table, that maps all system states s to their best action a, and the corresponding Q-value. The Q-learning algorithm is sketched in Algorithm 1.

Algorithm 1 Q-learning basic steps

```
1: Initialize the $Q$-values table, $Q(s,a)$.
2: Observe the current state, $s$.
3: loop {infinitely}
4:     Choose an action, $a$, for state $s$ based on one of the action selection policies, such as $\varepsilon$-greedy.
5:     Execute the action, and observe the reward, $r$, as well as the new state, $s'$.
6:     Update the $Q$-value for the state using the observed reward and the maximum reward possible for the next state. The resulting update formula is:

           $Q(s,a) = Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$        (5)

7:     Set the state $s$ to the new state $s'$.
```

Step 4 of the algorithm involves choosing an action for a given state. Among the existing action selection policies, ϵ-greedy is often the one of choice. Most of the time (with probability $1 - \epsilon$), the action with the best reward will be executed ($\arg_a \max Q(s, a)$); a random action will be chosen with a low probability ϵ, in order to explore non-visited actions. Once the action is executed, the corresponding Q-value is updated (step 6) with the obtained reward r and the maximum reward possible for the next state $\max_{a'} Q(s', a')$. The parameter γ is the discount factor that adjusts the importance given to future rewards. The update formula also includes a parameter α, that determines the learning rate. It can be the same for every state-action pair, or can be adjusted based on the number of times each state has been visited.

The policy learned by the agent is just the action that maximizes the Q-value in each state. Watkins and Dayan [104] proved that the discrete case of Q-learning converges to an optimal policy under certain conditions, and this is independent of the initial values of the Q-table. If each pair (s, a) is visited an infinite number of times, then the lookup table converges to a unique set of values $Q(s, a) = Q^*(s, a)$, which defines a stationary deterministic optimal policy. Note that the auto-scaling process is a continuing task, not stationary. For this reason, the Q-learning policy has to be learned infinitely and adapted to the workload changes.

An algorithm very similar to Q-learning is SARSA [98]. In contrast to Q-learning, it does not use the maximum reward for the next state $\max_{a'} Q(s', a')$ to update the $Q(s, a)$ value. Instead, the same action selection policy (check Step 4 in Algorithm 1) is applied to the new state s', in order to determine an action a'. Then, the corresponding $Q(s', a')$ value is used to update the current $Q(s, a)$, as shown in the following formula:

$$Q(s, a) = Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)] \quad (6)$$

The effect of a scaling decision takes some time to have an impact on the application, and the reward comes after a delay. Given that RL makes a decision with current information (application state) about a future reward (e.g. response time), a proactive nature is assumed for all RL approaches. The method includes two phases of the MAPE process: analyze and plan. First, information about application and rewards are collected in a lookup table (or any other structure) for later use (analyze phase). Then, in the planning phase, this information is used to decide the best scaling action.

5.2.2 Review of Proposals

Three basic elements have to be defined in order to apply RL to auto-scaling: the action set A, the state space S, and the reward function R. The first two highly depend on the type of scaling, either horizontal or vertical, whereas the reward function usually takes into account the cost of the acquired resources (renting VMs, bandwidth, ...) and the penalty cost for SLA violations. In case of horizontal scaling, the state is mostly defined in terms of the input workload and the number of VMs. For example, Tesauro et al. [99] propose using (w, u_{t-1}, u_t), where w is the total number of user requests observed per time period; u_t and u_{t-1} are the number of VMs allocated to the application in the current time step, and the previous time step, respectively; Dutreilh et al. [55] considered the (w, u, p) state definition, where p is the performance in terms of average response time to requests, bounded by a value P_{max} chosen from experimental observations. The set of actions for horizontal scaling are usually three: add a new VM, remove an existing VM, and do nothing.

In contrast, the state definition for vertical scaling takes into account the amount of resources assigned to each VM (CPU and memory, mostly). In particular, Rao et al. [92, 105] considered the following state definition: $(mem_1, time_1, vcpu_1, \ldots, mem_u, time_u, vcpu_u)$, where mem_i, $time_i$ and $vcpu_i$ are the i^{th} VM's memory size, scheduler credit and the number of virtual CPUs, respectively. For each of the three variables, possible operations can be either increase, decrease or no-operation (i.e. maintain the previous value). The actions for the RL task are defined as the combinations of the operations on each variable. Given that the variables are continuous, operations are discretized: mem is reconfigured in blocks of 256MB; scheduler changes ($time$) in 256 credits; and $vcpu$ in 1 unit. The same authors propose in [93] a distributed approach, in which a RL agent is learned per VM. In this case, the state is configured as $(CPU, bandwidth, memory, swap)$.

Even though Q-learning is the most extended algorithm in auto-scaling, there are some exceptions: e.g. Tesauro et al. [99] use the SARSA approach, explained before. Some of the articles referenced in this section ([45, 92, 93, 105]) do not specify which is the specific RL algorithm applied in the experiments, but the problem definition and update function resembles those of SARSA. Both Q-learning and SARSA present several problems, that have been addressed in a number of ways [42, 54, 55, 92, 99]:

- Bad initial performance and large training time. The performance obtained during live on-line training may be unacceptably poor, both initially and during a long training period, before a good enough solution is found. The algorithm needs to explore the different states and actions.
- Large state-space. This is often called the *curse of dimensionality problem*. The number of states grows exponentially with the number of state variables, which leads to scalability problems. In the simplest form, a lookup table is used to store a separate value for every possible state-action pair. As the size of such a table increases, any access to the table will take a longer time, delaying table updates and action selection.
- Exploration actions and adaptability to environment changes. Even assuming that an optimal policy is found, the environment conditions (e.g. workload pattern) may change and the policy has to be re-adapted. For this purpose, the RL algorithm executes a certain amount of exploration actions, believed to be suboptimal. This could lead to undesired bad performance, but it is essential in order to adapt the current policy. Following this method, RL algorithms are able to cope with relatively smooth changes in the behavior of the application, but not to sudden burst in the input workload.

The problem of the bad performance in the early steps has been addressed in a number of ways. Its main cause is the lack of initial information and the random nature of the exploration policy. For this reason, Dutreilh et al. [54] used a custom heuristic to guide the state space exploration, and the learning process lasted for 60 hours. The same authors [55] further propose an initial approximation of the Q-function that updates the value for all states at each iteration, and also speeds up the convergence to the optimal policy. Reducing this training time can also be addressed with a policy that visits several states at each step [92] or using parallel learning agents [42]. In the latter, each agent does not need to visit every state and action; instead, it can learn the value of non-visited states from neighboring agents. A radically different approach to avoid the poor early performance of on-line training consists of using an alternative model (e.g. a queuing model) to control the system, whilst the RL model is trained off-line on collected data [99].

Some authors have proposed methods to address the curse of dimensionality issue, reducing the state space. Bu et al. [45] use a Simplex optimization that selects the *promising* states that would return a high reward. A parallel approach would also help coping with large state spaces. Barrett et al. [42] create an agent per VM, that keeps its own, small lookup table. Using a lookup table for representing the Q-function is not efficient, and other nonlinear function approximators can be utilized, such as neural networks, CMACs (Cerebellar Model Articulation Controllers), regression trees, support vector machines, wavelets and regression splines. For example, neural networks [92, 99] take the state-action pairs as input and output the approximated Q-value. They are also able to predict the value for non-visited states, and deal with continuous state spaces. Rao et al. [93] combine the parallel approach (an agent per VM) with a CMAC [37, 93] to represent the Q function. Authors found that updates of the CMAC-based Q table only need 6.5 milliseconds, in comparison with the 50-second update time in a neural network.

It is also worth mentioning the usefulness of RL in other tasks that can be tightly linked to the auto-scaling problem. For example, application parameter configuration [45, 105]. Xu et al. [105] use an ANN-based RL agent to tune parameters directly related to the application and VM performance, such as MaxClients, Keepalive timeout, MinSpareServers, MaxThreads and Session timeout (these are examples of tunable parameters from Tomcat or Apache applications).

RL can be used in combination with other methods such as control theory. Following a radically different approach, Zhu et al. [108] combine a PID controller with an RL agent in charge of estimating the derivative term (this is further explained in Section 5.4). The controller guides the parameter adaptation of applications (e.g. image size) in order to meet the SLA. Then, virtual resources, CPU and memory, are dynamically provisioned according to the change in the adaptive parameters.

Table 2 Summary of the reviewed literature about reinforcement learning

Ref	Auto-scaling techniques	H/V	R/P	Metric	Monitoring	SLA	Workloads	Experimental platform
[54]	Rules + RL	H	P	Request rate, response time, number of VMs	Custom tool. 20 seconds	Response time	Synthetic. Made up of 5 sinusoidal oscillations	Custom testbed + RUBiS
[55]	RL	H	P	Number of user requests, number of VMs, response time	–	Response time, cost	Synthetic. With sinusoidal pattern	Custom testbed. Olio application + Custom decision agent (VirtRL)
[42]	RL	H	P	Number of user requests, number of VMs, time	Simulated	Response time, cost	Synthetic. Based on Poisson distribution	Custom simulator (Matlab)
[99]	RL(+ANN) − SARSA + Queuing model	H	P	Arrival rate, previous number of VMs	–	Response time	Synthetic. Poisson distribution (open-loop), different number of users and exponentially distributed think times (closed-loop)	Custom testbed (shared data cen ter). Trade3 application (a realistic simulation of an electronic trading platform)
[92]	RL(+ANN)	V	P	CPU and memory usage	Custom tool	Throughput, response time	Synthetic: 3 workload mixes (browsing, shopping and ordering)	Custom testbed. Xen + 3 applications (TPC-C, TPC-W, SpecWeb)
[105]	RL(+ANN)	V	P	CPU and memory usage	Custom tool	Response time	Synthetic: 3 workload mixes (browsing, shopping and ordering)	Custom testbed. Xen + 3 applica tions (TPC-C, TPC-W, SpecWeb)
[93]	RL(+CMAC)	V	P	CPU, I/O, memory, swap	Custom tool (scripts)	Response time,	Real. ClarkNet trace throughput	Custom testbed. Xen + 3 applica tions (TPC-C, TPC-W, SpecWeb)
[45]	RL(+Simplex)	V	P	CPU, memory - application parameters	Custom tool	Response time, throughput	Synthetic. Different number of clients	Custom testbed. Xen + 2 applica tions (TPC-C, TPC-W)
[108]	CT − PID controller + RL + ARMAX model + SVM regression	V	P	Application adaptive parameters CPU and memory	–	Application-related benefit function	Synthetic	Custom testbed. Xen + 2 real applications (Great Lake Forecasting System, Volume Rendering)

Before finishing this section, it is important to remark the interesting capability of RL algorithms to capture the best management policy for a target scenario, without any *a-priori* knowledge. The client does not need to define a particular model for the application; instead, it is learned online and adapted if the conditions of the application, workload or system change. In our opinion, RL techniques could be a promising approach to solve the auto-scaling task of general applications, but the field is not mature enough to satisfy the requirements of a real production scenario. In this open research field, efforts should be addressed towards providing an adequate adaptability to sudden bursts in the input workload, and also to deal with continuous state spaces and actions.

Table 2 shows a summary of the articles reviewed in this section.

5.3 Queuing Theory

Classical queuing theory has been extensively used to model Internet applications and traditional servers. Queuing theory can be used for the analysis phase of the auto-scaling task in elastic applications (see Section 3), i.e., estimating performance metrics such as the queue length or the average waiting time for requests. This section describes the main characteristics of a queuing model and how they can be applied to scalable scenarios.

5.3.1 Description of the Technique

Queuing theory makes reference to the mathematical study of waiting lines, or queues. The basic structure of a model is depicted in Fig. 1. Requests arrive to the system at a mean arrival rate λ, and are enqueued until they are processed. As the figure shows, one or more servers may be available in the model, that will attend requests at a mean service rate μ.

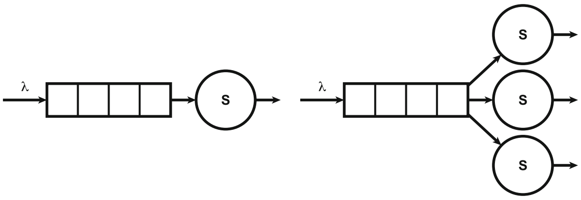

Fig. 1 A simple queuing model with one server (*left*) and multiple servers (*right*)

Kendall's notation is the standard system used to describe and classify queuing models. A queue is denoted by $A/B/C/K/N/D$. This is the meaning of each element in the notation:

A: *Inter-arrival time distribution.*

B: *Service time distribution.*

C: *Number of servers.*

K: *System capacity* or *queue length*. It refers to the maximum number of customers allowed in the system including those in service. When the system is fully occupied, further arrivals are rejected.

N: *Calling population*. The size of the population from which the customers come. If the requests come from an infinite population of customers, the queuing model is *open*, whereas a *closed* model is based on a finite population of customers.

D: *Service discipline* or *priority order*. The service discipline or priority order in which jobs in the queue are served. The most typical one is FIFO/FCFS (First In First Out / First Come First Served), in which the requests are served in the order they arrived. There are alternatives such as LIFO/LCFS (Last in First Out / Last Come First Served) and PS (Processor Sharing), among others.

Elements K, N and D are optional; if not present, it is assumed that $K = \infty$, $N = \infty$ and $D =$ FIFO. The most typical values for both inter-arrival time A and service time B, are M, D and G. M stands for Markovian and it refers to a Poisson process, which is characterized by a parameter λ that indicates the number of arrivals (requests) per time unit. Therefore, the inter-arrival or the service time will follow an exponential distribution. D means deterministic or constant times. Another commonly used value is G, that corresponds to a general distribution with known parameters.

The elastic application scenario described in Section 2 can be formulated using a simple queuing model, considering a single queue representing the load balancer, that distributes the requests among n VMs (see Fig. 1). In order to represent more complex systems such as multi-tier applications, a *queuing network* can be utilized. For example, each tier can be modeled as a queue with one or n servers.

Queuing theory is used to analyze systems with a stationary nature, characterized by constant arrival and service rates. Its objective is to derive some performance metrics based on the queuing model and some known parameters (e.g. arrival rate λ). Examples of performance metrics are the average time waiting in the queue, and the mean response time. In case of scenarios with changing conditions (i.e. non-constant arrival and services rates), such as our target, scalable applications, the parameters of the queuing model have to be periodically recalculated, and the metrics recomputed.

There are two main ways to solve queuing models: analytically and by means of simulation. The former can only be used with simple models with well-defined distributions for arrival and service processes, such as M/M/1 and G/G/1. A well-known analytical way of solving queuing networks is Mean Value Analysis (MVA) [83]. When analytical approaches are not feasible, that is, for relatively complex models, simulation can still be used to obtain the desired metrics.

M/M/1 is the simplest queuing (Poisson-based) model, where both the arrival times and the service times follow exponential distributions. In this case, the mean response time R of a M/M/1 model can be calculated as $R = \frac{1}{\mu - \lambda}$, given a service rate μ and an arrival rate λ (the response time R is the sum of the enqueued time and the service time). Another simple queuing model is G/G/1, in which the system's inter-arrival and service times are governed by general distributions with known mean and variance. The behavior of a G/G/1 system can be captured using the following equation: $\lambda \geq \left[s + \frac{\sigma_a^2 + \sigma_b^2}{2(R-s)} \right]^{-1}$, where R is the mean response time, s is the average service time for a request and σ_a^2, σ_b^2 are the variances of inter-arrival time service time, respectively.

In many queuing scenarios, it is useful to apply *Little's Law*, which states that the average number of customers (or requests) $E[C]$ in the system is equal to the average customer arrival rate λ multiplied by the average time of each customer in the system $E[T]$: $E[C] = \lambda \times E[T]$.

5.3.2 Review of Proposals

In the literature, both simple queuing models and more complex queuing networks have been widely used to analyze the performance of computer applications and systems.

Ali-Eldin et al. [39, 40] model a cloud-hosted application as a G/G/n queue, in which the number of servers n is variable. The model can be solved to compute, for example, the necessary resources required to process a given input workload λ, or the mean response time for requests, given a configuration of servers.

Queuing networks can also be used to model elastic applications, representing each VM (server) as a separate queue. For example, Urgaonkar et al. [100] use a network of G/G/1 queues (one per server). They use histograms to predict the peak workload. Based on this value and the queuing model, the number of servers per application tier is calculated. This number can be corrected using reactive methods. The clear drawback is that provisioning for peak load drives to high under-utilization of resources.

Multi-tier applications can be studied using one or more queues per tier. Zhang et al. [107] considered a limited number of users, and thus, they used a closed system with a network of queues; this model can be efficiently solved using MVA. Han et al. [62] modeled a multi-tier application as an open system with a network of G/G/n queues (one per each tier); this model is used to estimate the number of VMs that need to be added to, or removed from, the bottleneck tier, and the associated cost (VM usage is charged per minute, instead of the typical per-hour cost model). Finally, Bacigalupo et al. [41] considered a queuing network of three tiers, solving each tier to compute the mean response time.

The discussed approaches are used as part of the analysis phase of the MAPE loop. Many different techniques can be used to implement the planning phase, such as a predictive controller [40] or an optimization algorithm (e.g. distributing servers among different applications, while maximizing the revenue [101]). The information required for a queuing model, such as the input workload (number of requests) or service time can be obtained by on-line monitoring [62, 100] or estimated using different methods. For example, Zhang et al. [107] used a regression-based approximation in order to estimate the CPU demand, based on the number and type (browsing, ordering or shopping) of client requests.

Queuing models have been extensively used to model applications and systems. They usually impose a fixed architecture, and any change in structure or parameters require solving the model (with analytical or simulation-based tools). For this reason, they are not cheap when used with elastic (dynamically variable) applications that have to deal with a changing input workload and a varying pool of resources.

Additionally, a queuing system is an analysis tool, that requires additional components to implement a complete auto-scaler. Queuing models could be useful for some particular cases of applications, e.g. when the relationship between the number of requests and needed resources is quite linear. Although there are efforts to model more complex multi-tier applications, queuing theory might not be the best option when trying to design a general-purpose auto-scaling system.

Table 3 contains a summary of the articles reviewed in this section.

5.4 Control Theory

Control theory has been applied to automate the management of different information processing systems, such as web server systems, storage systems and data centers/server clusters. For cloud-hosted, elastic applications, a control system may combine both phases of the auto-scaling task (analysis and planning).

5.4.1 Description of the Technique

The main objective of a controller is to automate the management (e.g. scaling task) of a target system (e.g. a cloud application as defined in Section 2). The controller has to maintain the value of a *controlled variable* y (e.g. CPU load), close to the desired level or *set point* y_{ref}, by adjusting the *manipulated variable* u (e.g. number of VMs). The manipulated variable is the input to the target system, whereas the controlled variable is measured by a sensor and considered the output of the system.

There are three main types of control systems: open loop, feedback and feed-forward. *Open-loop* controllers, also referred to as non-feedback, compute the input to the target system using only the current state and its model of the system. They do not use feedback to determine whether the system output y has achieved the desired goal y_{ref}. In contrast, *feedback* controllers observe system output, and are able to correct any deviation from the desired value (see Fig. 2). *Feed-forward* controllers try to anticipate to errors in the output. They predict, using a model, the behavior of the system, and react before the error actually occurs. The prediction may fail and, for this reason, feedback and feed-forward controllers are usually combined.

Table 3 Summary of the reviewed literature about queuing theory

Ref	Auto-scaling techniques	H/V	R/P	Metric	Monitoring	SLA	Workloads	Experimental platform
[100]	QT + Histogram + Thresholds	H	R/P	Peak workload	Custom tool. 15 minutes	Response time	Synthetic and Real (World Cup 98)	Custom testbed. Xen + 2 applica tions (RUBiS and RUBBOS)
[101]	QT	H	R	Arrival rate, service time	Simulated	Response time	Real. E-commerce website (2001) + Synthetic traces	Custom simulator (Monte-Carlo)
[107]	QT + Regression (Predict CPU load)	–	P	Number and type of transactions (requests), CPU load	Custom tool. 1 minute	–	Synthetic (browsing, ordering and shopping)	Custom simulator, based on C++Sim. + Data collected from TPC-W
[62]	QT (model) + Reactive scaling	H	R	Arrival rate, service time	Custom tool. 1 minute	Response time, cost	Synthetic (browsing, ordering and shopping)	Custom testbed (called IC Cloud) + TPC-W benchmark
[41]	QT + Historical performance model	H	P	Arrival rate	–	Response time	Synthetic (browsing, buying)	Custom testbed. Eucalyptus + IBM Performance Benchmark Sample Trade

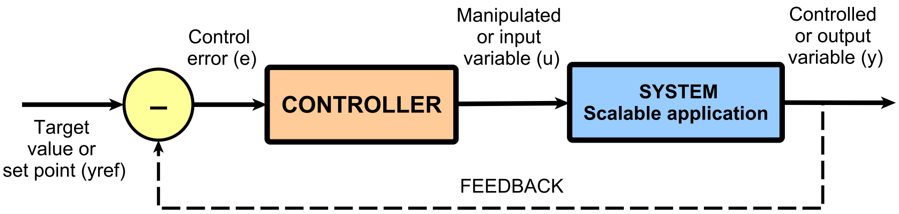
Control error (e)
Manipulated or input variable (u)
Controlled or output variable (y)
–
CONTROLLER
SYSTEM Scalable application
Target value or set point (yref)
FEEDBACK

Fig. 2 Block diagram of a feedback control system

From now on, focus will be put on feedback controllers, as they are frequently used in the literature. They can be classified into several categories [90]:

Fixed gain controllers: This class of controllers are very popular due to their simplicity. However, after selecting the tuning parameters, they remain fixed during the operation time of the controller. The most common controller in this class is called Proportional Integral Derivate (PID), with the following control algorithm:

$$u_k = K_p e_k + K_i \sum_{j=1}^{k} e_j + K_d(e_k - e_{k-1}) \qquad (7)$$

where u_k is the new value for the manipulated variable (e.g. new number of VM); e_k is the difference between the output y_k and the set point y_{ref}; and K_p, K_i and K_d are the proportional, integral and derivative gain parameters (respectively) that need to be adapted to a given target system. Different variants of the PID controller are used, such as the Proportional Integral (PI) controller or the Integral (I) controller. The latter can be represented as $u_k = u_{k-1} + K_i e_k$

Adaptive controllers: As the name suggests, adaptive control is able to adjust the parameter values on-line, in order to adapt the controller to the changing conditions of the environment. Examples of adaptive controllers are self-tuning PID controllers, gain-scheduling and self-tuning regulators. They are suitable for slowly varying workload conditions, but not for sudden bursts; in that case, the on-line model estimation process may fail to capture the dynamics of the system.

Model predictive controllers (MPC): MPCs follow a proactive approach; the future behavior (output) of the system is predicted, based on the model and the current output. In order to maintain this output close to the target value, the controller solves an optimization problem taking into account a predefined cost function. An example of MPC is the look-ahead controller [94].

As explained before, the controller has to adjust the input variable (e.g. number of VMs), in order to maintain the desired value in the output variable (e.g. average CPU load of 90 %). For this purpose, a formal relationship between the input and the output has to be modeled, so as to determine how a change in the former affects the value of the output. This formal relationship is denoted as *transfer function* in classical control theory, *state-space function* in modern control theory, or simply as *performance model*. PID controllers consider a simple linear equation, but there are many alternatives that consider non-linear approaches, and even several input and output variables (yielding a *Multiple-Input Multiple-Output* (MIMO) controller, instead being *Single-Input Single-Output* (SISO)). In the literature, authors have proposed different performance models of the system, including these:

ARMA(X) [88]: An ARMA (auto-regressive moving average) model is able to capture the characteristics of a time series and then makes predictions of future values. ARMAX (ARMA with eXogenous input) models the relationship between two time series. Both models are further explained in Section 5.5.

Kalman filter [69]: It is a recursive algorithm for making predictions based on time series.

Smoothing splines [43]: It is a method of smoothing (i.e fitting a smooth curve to a set of noisy observations) using splines. This term refers to a polynomial function, defined by multiple subfunctions.

Kriging model or Gaussian Process Regression [58]: It extends traditional linear regression with a statistical framework that is able to predict the value of the target function in un-sampled locations together with a confidence measure.

Fuzzy model [74, 103, 106]: Fuzzy models work with fuzzy rules. They are based on the idea that the membership of an element to a set has a degree value in a continuous interval between 0 and 1 (in contrast to Boolean logic). Fuzzy models are further described in Subsection 5.4.2.

5.4.2 Review of Proposals

Fixed-gain controllers, including PID, PI and I, are the simplest controller types, and have been widely used in the literature. For example, Lim et al. [75, 76] use an I controller to adjust the number of VMs based on average CPU usage, while Park and Humphrey [89] apply a PI controller to manage the resources required by batch jobs, based on their execution progress. Gain parameters K_p and K_I can be set manually, based on trial-and-error [75] or using an application-specific model. For example, in [89] a model is constructed to estimate the progress of a job with respect to the resources provisioned. Zhu and Agrawal [108] rely on a RL agent in order to estimate the derivative term of a PID controller. With the trial-and-error training, the RL agent learns to minimize the sum of the squared error of the control variables (the adaptive parameters) without violating the time and budget constraints over time.

Adaptive control techniques are also rather popular. For example, Ali-Eldin et al. [40] propose combining two proactive, adaptive controllers for scaling down, using dynamic gain parameters based on input workload. A reactive approach is used for scaling up. The same authors propose in [39] an adaptive, proportional controller, using a proactive approach for both scaling up and down, and taking into account the VM startup time. As stated before, adaptive control techniques rely on the use of performance models. Padala et al. [88] propose a MIMO adaptive controller that uses a second-order ARMA to model the non-linear and time-varying relationship between the resource allocation and its normalized performance. The controller is able to adjust the CPU and disk I/O usage. Bodík et al. [43] combine smoothing splines (used to map the workload and number of servers to the application performance) with a gain-scheduling adaptive controller. Kalyvianaki et al. [69] designed different SISO and MIMO controllers to determine the CPU allocation of VMs, relying on Kalman filters, whereas Gambi and Toffetti [58] utilized a Kriging model to predict job completion time as a function of the number of VM, the number of incoming requests and the jobs enqueued at the master node.

Fuzzy models have been used as a performance model in control systems to relate the workload (input variable) and the required resources (output variable). First, both input and output variables of the system are mapped into fuzzy sets. This mapping is defined by a membership function that determines a value within the interval [0,1]. A fuzzy model is based on a set of rules, that relate the input variables (pre-condition of the rule), to the output variables (consequence of the rule). The process of translating input values into one or more fuzzy sets is called fuzzification. Defuzzification is the inverse transformation which derives a single numeric value that best represents the inferred fuzzy values of the output variables. A control system that relies on a rule-based fuzzy model is called a *fuzzy controller*. Typically, the rule set and membership functions of a fuzzy model are fixed at design time, and thus, the controller is unable to adapt to a highly dynamic workload. An adaptive approach can be used, in which the fuzzy model is repeatedly updated based on on-line monitored information [103, 106]. Xu et al. [106] applied an adaptive fuzzy controller to the business-logic tier of a web application, and estimated the required CPU load for the input workload. A similar approach is followed by [103], but here authors focus on the database tier. They claim that they use the fuzzy model to predict the future resource needs; however, they use the workload of the current time step t, to calculate the resource needs r_{t+1} of the time step $t + 1$, based on the assumption that no sudden change happened within the duration of a time step.

A further improvement is the *neural fuzzy controller*, which uses a four-layer neural network (see Section 5.5) to represent the fuzzy model. Each node in the first layer corresponds to one input variable. The second layer determines the membership of each input variable to the fuzzy set (the fuzzification process). Each node in layer 3 represents the precondition part of one fuzzy logic rule. An finally, the output layer acts as a defuzzifier, which converts fuzzy conclusions from layer 3 into numeric output in terms of resource adjustment. At early steps, the neural network only contains the input and output layers. The membership and the rule nodes are generated dynamically through the structure and parameters learning. Lama and Zhou [74] relied on a neural fuzzy controller, that is capable of self-constructing its structure (both the fuzzy rules and the membership functions) and adapting its parameters through fast on-line learning (a reconfiguring controller type).

Finally, the last type of controllers that follow a proactive approach are MPCs. Roy et al. [94] combined an ARMA model for workload forecasting, with the look-ahead controller in order to optimize the resource allocation problem. Fuzzy models can also be used to create a Fuzzy Model Predictive Controller [102].

The suitability of controllers for the auto-scaling task highly depends on the type of controller and the dynamics of the target system. The idea of having a controller that automates the process of adding/ removing resources is very appealing, but still, the problem is how to create a reliable performance model that maps the input and output variables. Simple reactive controllers could be used for applications with easy to predict needs. However, it seems advisable to focus efforts towards both adaptive and MPCs that are able to adapt the application model and could be more suitable to produce a general auto-scaling solution.

Table 4 shows a summary of the articles reviewed in this section.

5.5 Time Series Analysis

Time series are used in many domains including finance, engineering, economics and bioinformatics, generally to represent the change of a measurement over time. A time series is a sequence of data points, measured typically at successive time instants spaced at uniform time intervals. An example is the number of requests that reaches an application, taken at one-minute intervals. The time series analysis can be used in the analysis phase of the auto-scaling process, in order to find repeating patterns in the input workload or to try to forecast future values.

5.5.1 Description of the Technique

Given the scenario described in Section 2, a certain performance metric, such as average CPU load or the input workload, will be sampled periodically, at fixed intervals (e.g. each minute). The result will be a time series X containing a sequence of w observations (w is the time series length):

$$X = x_t, x_{t-1}, x_{t-2}, ..., x_{t-w+1} \tag{8}$$

Time series techniques can be applied in order to predict future values of the metric, e.g. the future workload or resource usage. Based on this predicted value, a suitable auto-scaling action can be planned, using for example a set of predefined rules [71], or solving an optimization problem for the resource allocation [94].

Formally, the objective of time series analysis is to forecast future values of the time series, based on the last q observations, which is denoted as *input window* or *history window* (where $q \leq w$). The future value $\hat{x}_{t+r}$ is r intervals ahead of the input window. Time series analysis techniques can be classified into two broad groups: some of them focus on the direct prediction of future values, whereas other techniques try to identify the pattern (if present) followed by the time series, and then extrapolate it to predict future values. The first group includes moving average, auto-regression, ARMA (combining both), exponential smoothing and different approaches based on machine learning:

Moving average methods (MA): They can be used to smooth a time series in order to remove noise or to make predictions. The forecast value $\hat{x}_{t+r}$ is calculated as the weighted average of the last q consecutive values. Typically, the prediction interval r is set to 1. Then, the general formula is as follows: $\hat{x}_{t+r} = a_1x_t + a_2x_{t-1} + \ldots + a_qx_{t-q+1}$, where $a_1, a_2, \ldots, a_q$ are a set of positive weighting factors that must sum 1. Simple moving average MA(q) considers the arithmetic mean of the last q values, i.e., it assigns equal weight $\frac{1}{q}$ to all observations. In contrast, the weighted moving average WMA(q) assigns a different weight to each observation. Typically, more weight is given to the most recent terms in the time series, and less weight to older data.

Exponential smoothing (ES): Similarly to moving average, it calculates the weighted average of past observations, but exponential smoothing takes into account all the past history of the time series (w observations). It assigns *exponentially* decreasing weights over time. A new parameter is introduced, a smoothing factor α that weakens the influence

Table 4 Summary of the reviewed literature about control theory techniques

Ref	Auto-scaling Techniques	H/V	R/P	Metric	Monitoring	SLA	Workloads	Experimental Platform
[89]	CT: PI controller	V	R	Job progress	Sensor library	Job deadline	Batch jobs	Custom testbed. HyperV + 5 applications (ADCIRC, OpenLB, WRF, BLAST and Montage)
[75]	CT: PI controller (Proportional thresholding) + Exponential Smoothing for performance variable	H	R	CPU load, request rate	Hyperic HQ (Xen)	-	Synthetic. Different number of threads.	Custom testbed. Xen + ORCA + simple web service
[76]	CT: PI controller (Proportional thresholding)	H	R	CPU load, request rate	Hyperic SIGAR. 10 seconds	-	Synthetic.	Custom testbed. Xen + Modified CloudStone (using Hadoop Distributed File System)
[88]	CT: MIMO adaptive controller + ARMA (performance model)	V	P	CPU usage, disk I/O, response time	Xen + custom tool. 20 seconds	Response time	Synthetic and realistic (generated with MediSyn)	Custom testbed. Xen + 3 applications (RUBiS, TPC-W, media server)
[40]	CT: Adaptive controllers + QT	H	R/P	Number of requests, service rate	Simulated	Number of requests not handled	Real. World Cup 98.	Custom simulator in Python
[39]	CT: Adaptive, Proportional controller + QT	H	R/P	Number of requests, requests in buffer, service rate	Simulated. 1 minute	-	Real. World Cup 98 and Google Cluster Data	Custom simulator in Python
[43]	CT: Gain-scheduler (adaptive) + Smoothing splines (performance model) + Linear Regression	H	P	Number of requests, number of servers, response time	20 seconds	Response time	Synthetic (Faban generator)	Real provider. Amazon EC2 + CloudStone benchmark
[58]	CT: Self-adaptive controller + Kriging model (performance model)	H	P	Number of incoming and enqueued requests, number of VMs	-	Execution time	Synthetic (Batch jobs)	Custom testbed. Private cloud + Sun Grid Engine (SGE)
[106]	CT: fuzzy controller	V	R	Number and mixture of requests, CPU load	Custom tool. 20 seconds	Reply rate	Real (World Cup 98) and Synthetic (Httperf)	Custom testbed. VMware ESX Server + Java Pet Store.
[103]	Fuzzy model	V	P	Number of queries, CPU load, disk I/O bandwidth	Xentop. 10 seconds	Response time, throughput	Synthetic and realistic (based on World Cup 98)	Custom testbed. Xen + 2 applications (RUBiS and TPC-H)
[74]	CT: Adaptive controller + Fuzzy model (+ ANN)	V	P	Number of requests, resource usage	Custom tool. 3 minutes	End-to-end delay	Synthetic. (Pareto distribution)	Simulation
[69]	CT: Adaptive SISO and MIMO controllers + Kalman filter	V	P	CPU load	Custom tool. 5-10 seconds	Response time	Synthetic (Browsing and bidding mix)	Custom testbed. Xen + RUBiS application

of past data. There are different versions of exponential smoothing such as *simple* or single ES, and *Brown's double ES* [44]. In simple ES, the following formula is used recursively to calculate the current smoothed value s_t, based on the current observation x_t and the previous smoothed value s_{t-1}: $s_t = \alpha x_t + (1-\alpha)s_{t-1}$. The forecast for the next period $\hat{x}_{t+1}$ is simply the current smoothed value s_t. The predictor formula for simple exponential smoothing can be expanded as follows:

$$\begin{aligned}\hat{x}_{t+1} &= \alpha x_t + (1-\alpha)\hat{x}_t \\ &= \alpha x_t + (1-\alpha)[\alpha x_{t-1} + (1-\alpha)\hat{x}_{t-1}] \\ &= \alpha x_t + \alpha(1-\alpha)x_{t-1} + \\ &\quad (1-\alpha)^2[\alpha x_{t-2} + (1-\alpha)\hat{x}_{t-2}] \\ &\quad \vdots \\ &= \alpha x_t + \alpha(1-\alpha)x_{t-1} + \alpha(1-\alpha)^2 x_{t-2} + \\ &\quad \ldots + (1-\alpha)^{w-1}\hat{x}_{t-w+1}\end{aligned} \tag{9}$$

where $\hat{x}_{t+1}$ represents the forecast value for the period $t+1$, based on actual x_t value and the previous values of the time series. $\hat{x}_t$ refers to the forecast made for period t. The first smoothed value $\hat{x}_{t-w+1}$ can be set to the initial value of the time series x_{t-w+1}, or to the mean of the few first observations.

Simple ES is suitable for time series that have no significant trend changes. For time series with an existing linear trend, the Brown's double ES should be applied. It calculates two smoothing series:

$$\begin{aligned}s_t^1 &= \alpha x_t + (1-\alpha)s_{t-1}^1 \\ s_t^2 &= \alpha s_t^1 + (1-\alpha)s_{t-1}^2\end{aligned} \tag{10}$$

The second smooth series s_t^2 is obtained by applying simple ES to series s_t^1. Both smoothed values s_t^1 and s_t^2 are used to estimate the level c_t and trend d_t of the time series at the time t. Based on these, the forecast value $\hat{x}_{t+r}$ is calculated as follows:

$$\begin{aligned}\hat{x}_{t+r} &= c_t + r d_t \\ c_t &= 2s_t^1 - s_t^2 \\ d_t &= \tfrac{\alpha}{1-\alpha}s_t^1 - s_t^2\end{aligned} \tag{11}$$

Auto-regression of order p, AR(p): The prediction formula is determined as the linear weighted sum of the p previous terms in the series: $\hat{x}_{t+1} = b_1 x_t + b_2 x_{t-1} + \ldots + b_p x_{t-p+1} + \epsilon_t$. Parameter p corresponds to the number of terms in the AR equation, that may be different from the history window length w. The formula may include a white noise term ϵ_t. The key is to derive the best values for the weights or auto-regression coefficients $b_1, b_2, \ldots, b_p$. There are a diversity of techniques for computing AR coefficients, such as least squares or the maximum likelihood method. In the literature, the most common method is based on the calculation of auto-correlation coefficients and the Yule-Walker equations. The auto-correlation function (ACF) of a time series gives correlations between x_t and x_{t-k} for lag $k = 1, 2, 3, \ldots$:

$$r_k = \frac{covariance(x_t, x_{t-k})}{var(x_t)} = \frac{E[(x_t - \mu)(x_{t-k} - \mu)]}{var(x_t)} \tag{12}$$

The autocorrelation can be estimated as:

$$r_k = \frac{1}{(w-k)\sigma^2}\sum_{t=1}^{w-k}(x_t - \bar{x}) * (x_{t-k} - \bar{x}) \tag{13}$$

The full autocorrelation function can be derived by recursively calculating $r_k = \sum_{i=1}^{p} b_k r_{i-k}$. The result is a set of linear equations called the Yule-Walker equations, that can be represented in matrix form as:

$$\begin{bmatrix} 1 & r_1 & r_2 & r_3 & \ldots & r_{p-1} \\ r_1 & 1 & r_1 & r_2 & \ldots & r_{p-2} \\ r_2 & r_1 & 1 & r_1 & \ldots & r_{p-3} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ r_{p-1} & r_{p-2} & r_{p-3} & r_{p-4} & \ldots & 1 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_p \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ \vdots \\ r_p \end{bmatrix} \tag{14}$$

By solving this set of equations, auto-regression coefficients $b_1, b_2, \ldots, b_p$ can be derived for any p value. For example, for AR(1) (with $p = 1$), the auto-regression coefficient b_1 is equal to the corresponding autocorrelation coefficient r_1 ($b_1 = r_1$).

Auto-Regressive Moving Average, ARMA(p, q): This model combines both auto-regression (of order p) and moving average (of order q). As stated before, AR takes into account the last p

observations in the time series $x_t, x_{t-1}, x_{t-2}, \ldots, x_{t-p}$. The MA model, which is different from the MA method described previously, is the sum of the time series mean μ, plus the innovations or white error terms $\epsilon_t, \epsilon_{t-1}, \ldots, \epsilon_{t-q}$.

$$x_t = \mu + \epsilon_t + a_1\epsilon_{t-1} + a_2\epsilon_{t-2} + \ldots + a_q\epsilon_{t-q} \quad (15)$$

where $\epsilon_t \sim N(0, \sigma^2)$. Then, the ARMA model is represented as:

$$x_t = b_1x_{t-1} + \ldots + b_px_{t-p} + \epsilon_t + a_1\epsilon_{t-1} + \ldots + a_q\epsilon_{t-q} \quad (16)$$

Note that an ARMA$(0, q)$ is a pure MA model; and an ARMA$(p, 0)$ corresponds to an AR model. There are several techniques for selecting the appropriate values for the orders p and q, and estimating the coefficients $a_1, a_2, \ldots, a_q$ and $b_1, b_2, \ldots, b_p$.

ARMA is a suitable model for stationary processes, i.e. the mean and variance of the time series remain constant over time. Thus, the time series must not show any trend (the variations of the mean) or seasonal variations. If the AR model correctly fits the time series, the residual ϵ is a white noise that shows no pattern. An extension of the ARMA model, called ARIMA (Auto-Regressive Integrated Moving Average), can be applied to non-stationary time series. Another extension called ARMAX(p, q, b), or ARMA with eXogenous inputs, is able to capture the relationship between a given time series X and another external time series D. It contains the AR(p) and MA(q) models of X and a linear combination of the last b terms of the time series D.

Machine Learning-based techniques: These are two popular techniques used to carry out analysis of time series that can be considered part of the broader "machine learning" field:

Regression is a statistical method used to determine the polynomial function that is the closest to a set of points (in this case, the w values of the history window). *Linear regression* refers to the particular case of a polynomial of order 1. The objective is to find a polynomial such that the distance from each of the points to the polynomial curve is as small as possible and therefore fits the data the best. When the number of input variables is more than one, it is referred to as the Multiple Linear Regression. The Linear Regression equation is the same used in AR, but the weight estimation method differs.

Neural networks consist of an interconnected group of artificial neurons, arranged in several layers: an input layer with several input neurons; an output layer with one or more output neurons; and one or more hidden layers in between. For time series analysis, the input layer contains one neuron for each value in the history window, and one neuron for the predicted value in the output layer. During the training phase, it is fed with input vectors and random weights. Those weights will be adapted until the given input shows the desired output, at a learning rate ρ.

As previously stated, a group of time series analysis techniques try to identify the pattern that the series follows, and then use this pattern to extrapolate future values. Time series patterns can be described in terms of four classes of components: trend, seasonality, cyclical and randomness. The general trend (e.g. increasing or decreasing pattern), together with the seasonal variations that appear repeated over a specific period (e.g. day, week, month, or season), are the most common components in a time series. Input workloads of cloud applications may show different periodic components. The trend identifies the overall slope of the workload, whereas seasonality and cyclical determine the peaks at specific points of time in a short term and in a long term basis, respectively.

A wide diversity of methods can be used to find repetitive patterns in time series, including:

Pattern matching: It searches for similar patterns in the history time series, that are similar to the present pattern. It is very close to the *string matching problem*, for which several efficient algorithms are available (e.g. Knuth-Morris-Prat [53]).

Signal processing techniques: Fast Fourier Transform (FFT) is a technique that decomposes a signal time series into components of different frequencies. The dominant frequencies (if any) will correspond to the repeating pattern in the time series.

Auto-correlation: In auto-correlation, the input time series is repeatedly shifted (up to half the total window length), and the correlation is calculated between the shifted time series and the original one. If the correlation is higher than a given threshold (e.g. 0.9) after s shifts, a repeating pattern is declared, with duration s steps.

A basic tool for time series representation is the *histogram*. It involves distributing the values of the time series into several equal-width bins, and representing the frequency for each bin. It has been used in the literature to represent the resource usage pattern or distribution, and then predict future values.

5.5.2 Review of Proposals

In the context of elastic applications, time series analysis have been applied mostly to predict workload or resource usage. A simple moving average could be used for this purpose, but with poor results [60]. For this reason, authors have applied MA only to remove noise from the time series [75, 89], or just to have a comparison yardstick. For example, Huang et al. [65] present a resource prediction model (for CPU and memory utilization) based on double exponential smoothing, and compare it with simple mean and weighted moving average (WMA). ES clearly obtained better results, because it takes into account the history records w (not only the input window q) of the time series for the prediction. Mi et al. [85] also used Brown's double ES in order to forecast input workload of real traces (World Cup 98 and ClarkNet), and obtained good accuracy results, with a small amount of error (a mean relative error of 0.064 for the best case).

The auto-regression method has also been used for resource or workload forecasting ([49, 50, 60, 71, 73, 94]). For example, Roy et al. [94] applied AR for workload prediction, based on the last three observations. The predicted value is then used to estimate the response time. An optimization controller takes this response time as an input and computes the best resource allocation, taking into account the costs of SLA violations, leasing resources and reconfiguration. Kupferman et al. [73] applied auto-regression of order 1 to predict the request rate (requests per second) and found that its performance depends largely on several manager-defined parameters: the monitoring-interval length, the size of the history window and the size of the adaptation window. The history window determines the sensitivity of the algorithm to short-term versus long-term trends, while the size of the adaptation window determines how far into the future the model extends.

ARMA models are able to capture characteristics of a time series such as the input workload or the CPU usage. Fang et al. [57] found it useful to predict the future CPU usage of VMs. However, they remark the computational cost of this technique, that includes the choice of p and q, the estimation of the coefficients of each term and other parameters.

The history window values can also be the input for a neural network [67, 91] or a multiple linear regression equation [43, 67, 73]. The accuracy of both methods depends on the input window size. Indeed, Islam et al. [67] obtained better results when using more than one past value for prediction. Kupferman et al. [73] further investigated the topic and found that it is necessary to balance the size of each sample in the window, to avoid overreacting, but also to maintain a correct level of sensitivity to workload changes. They propose regressing over windows of different sizes, and then using the mean of all predictions. Another important issue is the choice of the prediction interval r. Islam et al. [67] propose using a 12-minute interval, because the setup time of VM instances in the cloud is typically around 5-15 min. In another context, Prodan and Nae [91] use a neural network to predict a game load (i.e. the number of entities or players) in the next two minutes. The neural network obtained better accuracy than moving average and simple exponential smoothing.

Most time series analysis techniques have been applied to vertical or horizontal scaling separately. Dutta et al. [56] claim that vertical scaling has limited range but has lower resource and configuration costs, while horizontal scaling can allow the application to achieve a much larger throughput but at a potentially higher cost. For this reason, they combine both VM resizing in case of small increments in the request rate, and apply horizontal scaling for major changes in the input workload. They use a polynomial regression to estimate the expected number of requests for the next interval. Fang et al. [57] focus on vertical scaling (CPU and memory) for *regular* changes in workload, whereas horizontal scaling is applied in order to handle sudden spikes and flash crowds.

Time series forecasting (associated to proactive decision making) can be combined with reactive

Table 5 Summary of the reviewed literature about techniques based on time series analysis

Ref	Auto-scaling Techniques	H/V	R/P	Metric	Monitoring	SLA	Workloads	Experimental Platform
[47]	TS: Pattern matching	H	P	Total number of CPUs	100 seconds	Number of serviced requests, cost	Real cloud workloads: from Animoto and 7 IBM cloud applications	Analytical models
[60]	TS: FFT and Discrete Markov Chains. Compared with auto-regression, auto-correlation, histogram, max and min.	V	P	CPU load	Libxenstat library. 1 minute	Response time	Real. World Cup 98 and ClarkNet. Also Synthetic trace	Custom testbed. Xen + RUBiS + part of Google Cluster Data trace for CPU usage.
[95]	TS: FFT and Discrete-time Markov Chain	V	P/R	CPU load, memory usage	Libxenstat library. 1 second	Response time, job progress	Synthetic. Based on World Cup 98 and EPA web server	Custom testbed. Xen + 3 appli cations (RUBiS, Hadoop MapReduce, IBM System S)
[57]	TS: ARMA	Both	P	Number of requests, CPU load	-	Prediction accuracy	Real traces. Collected from real applications and a real data center	Custom testbed. Xen and KVM
[65]	TS: Brown's double ES. Compared with WMA	-	P	CPU load, memory usage	Simulated	Prediction accuracy	Synthetic traffic generated with simulator	CloudSim simulator
[85]	TS: Brown's double ES.	-	P	Number of requests per VM	10 minutes	-	Real. World Cup 98 and ClarkNet. Synthetic. Poisson distribution	Custom testbed. TPC-W
[50]	TS: AR	H	P	Login rate, number of active connections, CPU load	Simulated	Service not available (login), energy consumption	Real. From Windows Live Messenger (login rate, number of active connections)	Simulator.
[71]	TS: AR + Threshold-based rules	Both	P	Number of requests	Zabbix	-	Synthetic	Hybrid: Amazon EC2 + Custom testbed (Xen + Eucalyptus + Php-Collab application)
[94]	TS: AR	H	P	Number of users in the system	-	Response time, VM cost, application reconfiguration cost	Real. World Cup 98	No experimentation on systems

Table 5 (continued)

Ref	Auto-scaling Techniques	H/V	R/P	Metric	Monitoring	SLA	Workloads	Experimental Platform
[67]	TS: ML Neural Network and (Multiple) LR + Sliding window	H	P	CPU load (aggregated value for all VMs)	Amazon CloudWatch. 1 minute	Prediction accuracy	Synthetic. TPC-W generator, constant growing	Real provider. Amazon EC2 and TPC-W application to generate the dataset. Prediction models are only evaluated using cross-validation and several accuracy metrics. Experiments in R-Project.
[91]	TS: ML Neural Network (compared to MA, last value and simple ES)	H	P	Number of entities (players)	2 minutes	Prediction accuracy	Synthetic. Entities (players) with different behaviors	Simulator of a MMORPG game
[56]	TS: Polynomial regression	Both	P	Number of requests	Custom tool. 1 minute	Response time, cost for VM, application license and reconfiguration	Real traces. From a production data center of a company	Custom testbed. KVM + Olio
[66]	Threshold-based rules (scale out) + TS, polynomial regression (scale in)	H	R/P	CPU load (scale out), number of requests, number of VMs (scale in)	1 minute	Response time	Synthetic. Httperf	Custom testbed. Eucalyptus + RUBiS
[49]	TS: AR(1) and Histogram + QT	H	P	Request rate and ser vice demand	Simulated. 1, 5, 10 and 20 minutes	Response time	Synthetic (Poisson distribution) and Real (World Cup 98)	Custom simulator + algorithms in Matlab

techniques. For example, Iqbal et al. [66] proposed a hybrid scaling technique that utilizes reactive rules for scaling up (based on CPU usage) and a regression-based approach for scaling down. After a fixed number of intervals in which response time is satisfied, they calculate the required number of application-tier and database-tier instances using polynomial regression (of degree two).

Some auto-scaling proposals use time series analysis techniques that deal with pattern identification, applied to the input workload [46, 47, 60, 95]. The most complete comparison of this class of techniques is done by Gong et al. [60]; they propose using FFT to identify repeating patterns in resource usage (CPU, memory, I/O and network), and compare it with auto-correlation, auto-regression and histogram. Pattern matching, proposed by Caron et al. [46, 47], has two main drawbacks: the large number of parameters in the algorithm (such as the maximum number of matches or the length of the predicted sequence), that highly affect the performance of the algorithm, and the time required to explore the past history trace.

Simple histograms have also been used by some authors to predict the resource usage of applications, considering the mean of the distribution [49], or the mean of the bin with the highest frequency [60].

Time series analysis techniques are very appealing for implementing auto-scalers, as they are able to predict future demands arriving to elastic applications. Having this information, it is possible to provide resources in advance and deal with the time required to start up new VMs or add resources to a particular instance. Despite the potential of this set of techniques, their main drawback relies on the prediction accuracy, that highly depends on the target application, input workload pattern and/or burstiness, the selected metric, the history window and prediction interval, as well as on the specific technique being used. Efforts should be focused on automating the selection of the best prediction technique for a particular application or application class.

Table 5 contains a summary of the articles reviewed in this section.

6 Conclusions and Future Work

The focus of this review has been on auto-scaling elastic applications in cloud environments. While capacity planning is required for any scalable applications, the elasticity provided by cloud infrastructures allows for an almost-immediate adaptation of resources to application needs (driven by an external demand). Auto-scalers try to automate this adaptation, minimizing resource-related costs while allowing the application to comply with the SLA.

In Section 3, the auto-scaling task was defined as a MAPE process, composed of four phases: monitor, analyze, plan and execute. Given the extensive literature about auto-scaling for elastic applications, a classification criterion has been proposed to organize the different proposals into five main categories: threshold-based rules, reinforcement learning, queuing theory, control theory and time series analysis. Each of these categories has been described separately, including their pros/cons, together with a critical review of relevant articles using the technique that defines the category.

One of the conclusions that can be extracted from this survey is that reactive auto-scaling systems might not be able to cope with abrupt changes in the input workload, specially in the case of sudden traffic surges. One of the reasons is the time required to acquire and set-up new resources (e.g. the boot-up time of a new instance). It seems advisable to redirect research efforts towards developing proactive auto-scaling systems able to predict future needs and acquire the corresponding resources with enough anticipation, maybe using some reactive methods to correct possible prediction errors. As an example, Authors of [86] demonstrate that, compared to a purely reactive controller, their auto-scaling system (based on both reactive and proactive models) was able to make better provisioning decisions, yielding few QoS violations. Additionally, attention should be paid to methods to reduce the time necessary to provision new VMs, including those that take advantage of vertical scaling actions that usually needs only seconds to be fulfilled.

In our opinion, auto-scaling systems should take advantage of the prediction capabilities of time series analysis techniques, together with the automation capabilities of controllers. It is our plan to propose a predictive auto-scaling technique based on time series forecasting algorithms. The review of the literature has shown that the accuracy of these algorithms depend on different parameters such as the sizes of the history window and the adaptation window. Optimization techniques could be used to adjust these values, in order to customize the parameters for a given scenario.

When going through this review, the reader should have noticed the large diversity of testing methodologies that authors have used to assess their proposals (many of them are described in the companion Appendix). Authors use different ways of generating input workload, different types of applications (realistic or simulated), different SLA definitions, different execution platforms, etc. In fact, the lack of a common testing platform capable of generating a well-defined set of standardized metrics is the reason that prevented a comparative analysis of the reviewed techniques in quantitative terms. We are currently working in the development of a simulation-based workbench that could provide a common testing platform. It has been used already to implement and compare several auto-scaling techniques [77], although this work is still preliminary. In the same line, it would be of interest to define a commonly accepted scoring metric to objectively determine whether one auto-scaling algorithm performs better than another in a particular scenario.

Throughout this review it has been assumed that a client runs an elastic application on a single and homogeneous cloud infrastructure. However, clients may need to deploy their applications on hybrid or federated clouds. Hybrid clouds combine resources from both a private and a public cloud, while federated systems comprise different public providers. The combination of different cloud platforms represents additional challenges for the auto-scaling task. For example, the monitoring information has to be gathered from different sources, probably using different tools with their own APIs. The list of performance metrics available and the granularity level will depend on the provider itself. In the analysis phase, an extra effort will be necessary to select the suitable metric from each provider, maybe requiring mapping functions to combine different metrics. After making a scaling decision, a variety of APIs may be available to implement it in different platforms, and the options and their consequent effects may vary greatly. Examples of these platform-dependent particularities are the availability of horizontal or vertical scaling (or any of them), the capabilities of the different VM templates and the billing schemes (per hour, per minute). Different efforts have been carried out towards simplifying the use of federated/hybrid models [70, 84], or the interoperability among different cloud platforms, including the proposal of open standards, such as Open Virtualization Format (OVF), Open Cloud Computing Interface (OCCI) and Cloud Data Management Interface (CDMI). However, orchestrating the auto-scaling capabilities of different cloud providers is still an open challenge.

Finally, the problem of auto-scaling is closely related to an infrastructure-related one: VM placement, the actual mapping of the VMs forming an application onto the physical servers of the cloud provider. We are currently working on ways to optimize the placement of fixed-size applications in order to maximize the revenue for the cloud provider, while satisfying the resource SLA. Auto-scaling capability of (elastic) applications imposes an additional challenge to the provider, as the set of VMs of an application varies with time.

Acknowledgments This work has been partially supported by the Saiotek and Research Groups 2007-2012 (IT-242-07) programs (Basque Government), TIN2013-41272P and COMBIOMED network in computational biomedicine (Carlos III Health Institute). Dr. Miguel-Alonso is member of the HIPEAC European Network of Excellence. Mrs Lorido-Botrán is supported by a doctoral grant from the Basque Government. Finally, we would like to thank the anonymous reviewers for their suggestions and comments that greatly contributed to improve this survey.

Appendix

Performance Evaluation in the Cloud: Experimental Platforms, Application Benchmarks and Workloads

The auto-scaling techniques proposed in the literature have been tested under very different conditions, which makes it impossible to come up with a fair comparative assessment. Researchers in the field have built their own evaluation platforms, suitable to their own needs. These evaluation platforms can be classified into simulators, custom testbeds and public cloud providers. Except for simple simulators, a scalable application benchmark has to be executed on the platform to carry out an evaluation. The input workload to the application can be either synthetic, generated with specific programs, or obtained from real users.

A.1 Experimental Platforms

Experimentation could be done in *production* infrastructures, either from real cloud providers or in a private cloud. The major advantage of using a real cloud platform is that proposals can be checked in actual scenarios, thus proving a proof of suitability. However, it has a clear drawback: for each experimentation, the whole scenario needs to be set. In case of using a public cloud provider, the infrastructure is already given, but the researcher still needs to configure the monitoring and auto-scaling system, deploy an application benchmark and a load generator over a pool of VMs, figure out how to extract the information and store it for later processing. Probably, each execution will be charged according to the fees established by the cloud provider.

In order to reduce the experimentation cost and to have a more controlled environment, a *custom testbed* could be set up. This has a cost in terms of system configuration effort (in addition to buying the hardware, if not available). An initial, non-trivial step consists of installing the virtualization software that will manage the VM creation, support scaling and so on. Virtualization can be applied at the server level, OS level or at the application level. For custom testbeds, a server level virtualization environment is needed, commonly referred to as *Hypervisor* or *Virtual Machine Monitor* (VMM). Some popular hypervisors are Xen [36], VMWare ESXi [32] and Kernel Virtual Machine (KVM) [15]. There are several platforms for deploying custom clouds, including open-source alternatives like OpenStack [19] and Eucalyptus [9], and commercial software like vCloud Director [31]. OpenStack is an open-source initiative supported by many cloud-related companies such as RackSpace, HP, Intel and AMD, with a large customer-base. Eucalyptus enables the creation of on-premises Infrastructure as a Service clouds, with support for Xen, KVM and ESXi, and the Amazon EC2 API. VCloud Director is the commercial platform developed by VMWare.

As an alternative to a real infrastructure (custom testbed or public cloud provider), software tools could be used to *simulate* the functioning of a cloud platform, including resource allocation and deallocation, VM execution, monitoring and the remaining cloud management tasks. The researcher could select an already existing simulator and adapt it to her needs, or implement a new one from scratch. Obviously, using a simulator implies an initial effort to adapt or develop the software but, in contrast, has many advantages. The evaluation process is shortened in many ways. Simulation allows testing multiple algorithms, avoiding infrastructure re-configurations. Besides, simulation isolates the experiment from the influence of external, uncontrolled factors (e.g. interference with other applications, provider-induced VM consolidation processes, etc.), something impossible in real cloud infrastructures. Experiments carried out in real infrastructures may last hours, whereas in an event-based simulator, this process may take just a few minutes. Simulators are highly configurable and allow the researcher to gather any information about system state or performance metrics. In spite of the advantages mentioned, simulated environments are still an abstraction of physical machine clusters, thus the reliability of the results will depend on the level of implementation detail considered during the development. Some research-oriented cloud simulators are CloudSim [7], GreenCloud [14], and GroudSim [87].

A.2 Application Benchmarks

A scalable application benchmark is executed on top of an experimental platform, based on a public cloud provider or a custom testbed, in order to measure the performance of the system. Simulated experimental platforms do not always require the use of a benchmark: a simplified view of an application may be part of the simulated model. Typically, benchmarks comprise a web application together with a workload generator that creates synthetic session-based requests to the application. Some commonly used benchmarks for cloud research are RUBiS [1], TPC-W [30], CloudStone [8] and WikiBench [33]. Although both RUBiS and TPC-W benchmarks are out-dated or declared obsolete, they are still being used by the research community.

RUBiS [1]: It is a prototype of an auction website modeled after eBay. It offers the core functionality of an auction site (selling, browsing and bidding) and supports three kinds of user sessions: visitor, buyer, and seller. The applications consists of three main components: Apache load balancer server, JBoss application server and MySQL database server. The last update in this benchmark was published in 2008.

TPC-W [30]: TPC [29] is a nonprofit organization founded to define transaction processing and database benchmarks. Among them, TPC-W is a complex e-commerce application, specifically an online bookshop. It simulates three different profiles: primarily shopping, browsing and web-based ordering. The performance metric reported is the number of web interactions processed per second. It was declared obsolete in 2005.

CloudStone [8]: It is a multi-platform performance measurement tool for Web 2.0 and cloud computing developed by the Rad Lab group at the University of Berkeley. CloudStone includes a flexible, realistic workload generator (Faban) to generate load against a realistic Web 2.0 application (Olio). The stack can be deployed on Amazon EC2 instances. Olio 2.0 is a two-tier social networking benchmark, with a web frontend and a database backend. The application metric is the number of active users of the social networking application, which drives the throughput or the number of operations per second.

WikiBench [33]: It is a web hosting benchmark, that uses real data from the Wikipedia database, and generates real traffic based on the Wikipedia access traces. The core application is MediaWiki [16], a free open source wiki package originally used on the Wikipedia website.

There are other, less used benchmarks such as RUBBoS [24], SpecWeb [97] and TPC-C [28]. RUBBoS is a bulletin board benchmark, modeled after an online news forum like Slashdot. The last update was in 2005. SpecWeb is a benchmark tool, created by the Standard Performance Evaluation Corporation (SPEC), that is able to generate banking, e-commerce and support (large downloads), synthetic workloads. SpecWeb has been discontinued in 2012 and now the SPEC company has created a cloud benchmarking group [26]. TPC-C is an on-line transaction processing benchmark that simulates a complete computing environment where a population of users executes transactions against a database. The performance metric is the number of transactions per minute.

A.3 Workloads

As described before, workloads represent inputs to be processed by application benchmarks. They can be generated using some patterns or gathered from real cloud applications (and typically stored in trace files).

Cloud-based systems process two main classes of workload: batch and transactional. Batch workloads consist of arbitrary, long running, resource-intensive jobs, such as scientific programs or video transcoding. The most well-known examples of transactional workloads are web applications built to serve on-line HTTP clients. These systems usually serve content types such as HTML pages, images or video streams. All of these contents can be statically stored or dynamically rendered by the servers.

A.3.1 Synthetic Workloads

Synthetic workloads can be generated based on different patterns. According to Mao and Humphrey [2, 79], there are four representative workload patterns in cloud environments: Stable, Growing, Cycle/Bursting and On-and-off. Each of them represents a typical application or scenario. A Stable workload is characterized by a constant number of requests per minute. In contrast, a Growing pattern shows a load that rapidly increases, for example, a piece of news that suddenly becomes popular. The third pattern is called Cyclic/Bursting because it can show regular periods (e.g. daytime has more workload than the night) or bursts of load in a punctual date (e.g. a special offer). The last typical pattern found is the On-and-off workload that may represent batch processing or data analysis performed everyday.

There is a broad range of workload generators, that may create either simple requests, or even real HTTP sessions, that mix different actions (e.g. login or browsing) and simulate user thinking times. Examples of workload generators are:

Faban [8]: A Markov-based workload generator, included in the CloudStone stack.

Apache JMeter [4]: A workload generator implemented in Java, used for load testing and measuring performance. It can be used to test performance on both static and dynamic resources (files, servlets, Perl scripts, databases and queries, FTP servers and more). It can also generate heavy loads for a server, network or object, either to test its strength or to analyze the overall performance under different scenarios.

Rain [21]: A workload generation toolkit that uses parameterized statistical distributions in order to model different classes of workload.

Httperf [27]: A tool for measuring web server performance. It provides a flexible facility for generating various HTTP workloads and for measuring server performance.

Synthetic workloads are suitable to carry out controlled experimentation. For example, the workload could be tuned in order to test the system under different number of users or request rates, with smooth increments or sudden peaks. However, they may not be realistic enough, a reason that makes necessary to use workloads collected from real production systems.

A.3.2 Real Workloads

To the best of our knowledge, there are no publicly available, real traces from cloud-hosted applications, and this is an evident drawback for cloud research. In the literature, some authors have generated their own traces, running benchmarks or real applications in cloud platforms. There are also some references to traces from private clouds that have not been published. However, most authors have used traces from Internet servers, such as the ClarkNet trace [6], the World Cup 98 trace [35] and Wikipedia access traces [34].

The ClarkNet trace [6] contains the HTTP requests received by the ClarkNet server over a two-weeks period in 1995. ClarkNet is an Internet access provider for the metro Baltimore-Washington DC area. The trace shows a clear cyclic workload pattern (see Fig. 3): daytime has more workload than the night, and the workload on weekends is lower than that taking place on weekdays.

The World Cup 98 trace [35] has been extensively used in the literature. It contains all the HTTP requests made to the 1998 World Cup Web site between April 30, 1998 and July 26, 1998.

More recently, several Wikipedia access traces were published for research purposes. They contain the URL requests made to the Wikipedia servers between September, 2007 and January, 2008. Most requests are read-only queries to the database.

Some authors have used traces from Grid environments [46], but although there is an extensive number of public traces, the job execution

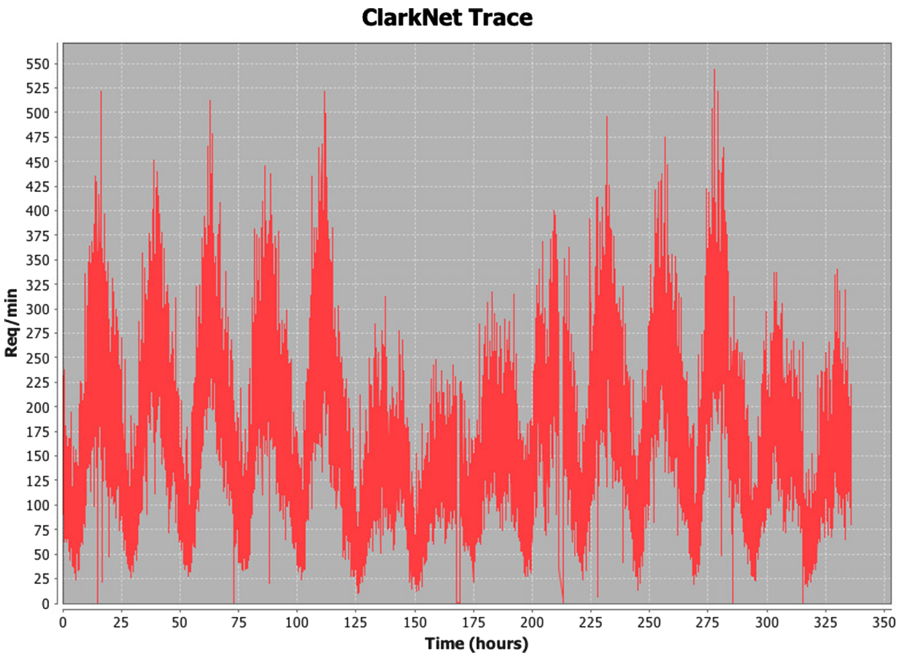

Fig. 3 Number of requests per minute for ClarkNet Trace

scheme is not suitable for evaluating elastic applications. However, they could be useful for batch-based workloads.

It is also worth mentioning the Google Cluster Data [12], two sets of traces that contain the workloads running on Google compute cells. The first dataset refers to a 7-hour period and consists of a set of tasks. However, the data records have been anonymized, and the CPU and RAM consumption have been normalized and obscured using a linear transformation. The second trace includes significantly more information about jobs, machine characteristics and constraints. This trace includes data from an 11k-machine cell over about a month-long period. Like in the first trace, all the numeric data have been normalized, and there is no information about the job type. For this reason, these Google traces can be utilized to test auto-scaling techniques, but they could be useful in other cloud-related research scenarios.

References

1. RUBiS: Rice University Bidding System. http://rubis.ow2.org/. Online: Accessed 13 Sept 2012 (2009)
2. Workload Patterns for Cloud Computing. http://watdenkt.veenhof.nu/2010/07/13/workload-patterns-for-cloud-computing/. Online: Accessed 29 Jan 2014 (2010)
3. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/. Online: Accessed 13 Sept 2012 (2012)
4. Apache JMeter., http://jmeter.apache.org/. Online: Accessed 18 Sept 2012 (2012)
5. AWS Elastic Beanstalk (beta). Easy to begin, Impossible to outgrow. http://aws.amazon.com/elasticbeanstalk/. Online: Accessed 13 Sept 2012 (2012)
6. ClarkNet HTTP Trace (From the Internet Traffic Archive). http://ita.ee.lbl.gov/html/contrib/ClarkNet-HTTP.html. Online: Accessed 13 Sept 2012 (2012)
7. CloudSim: A Framework for Modeling and Simulation of Cloud Computing Infrastructures and Services. http://www.cloudbus.org/cloudsim/. Online: Accessed 18 Sept 2012 (2012)
8. CloudStone Project by Rad Lab Group. http://radlab.cs.berkeley.edu/wiki/Projects/Cloudstone/. Online: Accessed 13 Sept 2012 (2012)
9. Eucalyptus Cloud. http://www.eucalyptus.com/. Online: Accessed 18 Sept 2012 (2012)
10. Google App Engine. http://cloud.google.com/products/. Online: Accessed 13 Sept 2012 (2012)
11. Google Apps for Business. http://www.google.com/intl/es/enterprise/apps/business/products.html. Online: Accessed 13 Sept 2012 (2012)
12. Google Cluster Data. Traces of Google workloads. http://code.google.com/p/googleclusterdata/. Online: Accessed 13 Sept 2012 (2012)
13. Google Compute Engine. http://cloud.google.com/products/compute-engine.html/. Online: Accessed 13 Sept 2012 (2012)
14. Greencloud - The green cloud simulator. http://greencloud.gforge.uni.lu/. Online: Accessed 18 Sept 2012 (2012)
15. Kernel Based Virtual Machine. http://www.linux-kvm.org/. Online: Accessed 18 Sept 2012 (2012)
16. MediaWiki. http://www.mediawiki.org/wiki/MediaWiki. Online: Accessed 24 Nov 2012 (2012)
17. Microsoft Office 365. http://www.microsoft.com/en-us/office365/online-software.aspx. Online: Accessed 13 Sept 2012 (2012)
18. Microsoft Windows Azure. https://www.windowsazure.com/en-us/. Online: Accessed 13 Sept 2012 (2012)
19. OpenStack Cloud Software. Open source software for building private and public clouds. http://www.openstack.org/. Online: Accessed 18 Sept 2012 (2012)
20. Rackspace. The open cloud company. http://www.rackspace.com/. Online: Accessed 13 Sept 2012 (2012)
21. Rain Workload Toolkit. https://github.com/yungsters/rain-workload-toolkit/wiki. Online: Accessed 13 Sept 2012 (2012)
22. RightScale Cloud Management. http://www.rightscale.com/. Online: Accessed 13 Sept 2012 (2012)
23. RightScale. Set up Autoscaling using Voting Tags. http://support.rightscale.com/03-Tutorials/02-AWS/02-Website_Edition/Set_up_Autoscaling_using_Voting_Tags. Online: Accessed 13 Sept 2012 (2012)
24. RUBBoS: Bulletin Board Benchmark. http://jmob.ow2.org/rubbos.html/. Online: Accessed 18 Sept 2012 (2012)
25. Salesforce.com. http://www.salesforce.com/. Online: Accessed 13 Sept 2012 (2012)
26. SPEC forms cloud benchmarking group. http://www.spec.org/osgcloud/press/cloudannouncement20120613.html. Online: Accessed 18 Sept 2012 (2012)
27. The httperf HTTP load generator. http://code.google.com/p/httperf/. Online: Accessed 18 Sept 2012 (2012)
28. TPC-C., http://www.tpc.org/tpcc/default.asp/. Online: Accessed 18 Sept 2012 (2012)
29. TPC. Transaction Processing Performance Council. http://www.tpc.org/default.asp. Online: Accessed 13 Sept 2012 (2012)
30. TPC-W. http://www.tpc.org/tpcw/default.asp. Online: Accessed 13 Sept 2012 (2012)
31. VMware vCloud Director. Deliver Complete Virtual Datacenters for Consumption in Minutes. http://www.eucalyptus.com/. Online: Accessed 18 Sept 2012 (2012)
32. VMware vSphere ESX and ESXi Info Center. http://www.vmware.com/es/products/datacenter-virtualization/vsphere/esxi-and-esx/overview.html. Online: Accessed 18 Sept 2012 (2012)

33. WikiBench: A Web hosting benchmark. http://www.wikibench.eu. Online: Accessed 24 Nov 2012 (2012)
34. Wikipedia access traces. http://www.wikibench.eu/?page_id=60. Online: Accessed 24 Nov 2012 (2012)
35. World Cup 98 Trace (From the Internet Traffic Archive). http://ita.ee.lbl.gov/html/contrib/WorldCup.html. Online: Accessed 13 Sept 2012 (2012)
36. Xen hypervisor. http://www.xen.org/s. Online: Accessed 18 Sept 2012 (2012)
37. Albus, J.: A new approach to manipulator control: The cerebellar model articulation controller (CMAC). Transaction of the ASME, Journal of Dynamic Systems, Measurement and Control (1975)
38. Alhamazani, K., Ranjan, R., Mitra, K., Rabhi, F., Khan, S.U., Guabtni, A., Bhatnagar, V.: An Overview of the Commercial Cloud Monitoring Tools: Research Dimensions, Design Issues, and State-of-the-Art. arXiv:13126170 (2013)
39. Ali-Eldin, A., Kihl, M., Tordsson, J., Elmroth, E.: Efficient provisioning of bursty scientific workloads on the cloud using adaptive elasticity control. In: Proceedings of the 3rd workshop on Scientific Cloud Computing Date - ScienceCloud ’12, pp. 31–40. ACM (2012)
40. Ali-Eldin, A., Tordsson, J., Elmroth, E.: An adaptive hybrid elasticity controller for cloud infrastructures. In: Network Operations and Management Symposium (NOMS), 2012, IEEE, IEEE, pp. 204–212 (2012)
41. Bacigalupo, D.A., van Hemert, J., Usmani, A., Dillenberger, D.N., Wills, G.B., Jarvis S.A.: Resource management of enterprise cloud systems using layered queuing and historical performance models. In: 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, (IPDPSW), IEEE, pp. 1–8 (2010). doi:10.1109/IPDPSW.2010.5470782
42. Barrett, E., Howley, E., Duggan, J.: Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. Concurrency and Computation: Practice and Experience (2012)
43. Bodík, P., Griffith, R., Sutton, C., Fox, A., Jordan, M., Patterson, D.: Statistical machine learning makes automatic control practical for internet datacenters. HotCloud’09 Proceedings of the 2009 Conference on Hot Topics in Cloud Computing (2009)
44. Brown, R., Meyer, R.: The fundamental theorem of exponential smoothing. Operations Research (1961)
45. Bu, X., Rao, J., Xu, C.Z.: Coordinated Self-configuration of Virtual Machines and Appliances using A Model-free Learning Approach. IEEE Transactions on Parallel and Distributed Systems (2012)
46. Caron, E., Desprez, F., Muresan, A.: Forecasting for Cloud computing on-demand resources based on pattern matching. Research Report RR-7217, INRIA (2010)
47. Caron, E., Desprez, F., Muresan, A.: Pattern Matching Based Forecast of Non-periodic Repetitive Behavior for Cloud Clients. J. Grid Comput. **9**(1), 49–64 (2011). doi:10.1007/s10723-010-9178-4
48. Casalicchio, E., Silvestri, L.: Autonomic Management of Cloud-Based Systems: The Service Provider Perspective. In: Gelenbe, E., Lent, R. (eds.) Computer and Information Sciences III, pp. 39–47. Springer, London (2013). doi:10.1007/978-1-4471-4594-3_5
49. Chandra, A., Gong, W., Shenoy, P.: Dynamic resource allocation for shared data centers using online measurements. In: Proceedings of the 11th International Conference on Quality of Service, pp. 381–398 (2003)
50. Chen, G., He, W., Liu, J., Nath, S., Rigas, L., Xiao, L., Zhao, F.: Energy-aware server provisioning and load dispatching for connection-intensive internet services. In: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation, USENIX Association, vol. 8, pp. 337–350 (2008)
51. Chieu, T.C., Mohindra, A., Karve, A.A., Segal, A.: Dynamic scaling of web applications in a virtualized cloud computing environment. In: IEEE International Conference on e-Business Engineering, ICEBE09, IEEE, pp. 281–286 (2009)
52. Chieu, T.C., Mohindra, A., Karve, A.A.: Scalability and performance of web applications in a compute cloud. In: 2011 IEEE 8th International Conference on e-Business Engineering (ICEBE), pp. 317–323 (2011)
53. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to algorithms, chapter 32: string matching. McGraw-Hill Higher Education (2001)
54. Dutreilh, X., Moreau, A., Malenfant, J., Rivierre, N., Truck, I.: From data center resource allocation to control theory and back. In: Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on, IEEE, pp. 410–417 (2010)
55. Dutreilh, X., Kirgizov, S., Melekhova, O., Malenfant, J., Rivierre, N., Truck, I.: Using reinforcement learning for autonomic resource allocation in clouds: Towards a fully automated workflow. In: Seventh International Conference on Autonomic and Autonomous Systems, ICAS 2011, IEEE, pp. 67–74 (2011)
56. Dutta, S., Gera, S., Verma, A., Viswanathan, B.: SmartScale: Automatic application scaling in enterprise clouds. In: 2012 IEEE Fifth International Conference on Cloud Computing, IEEE, pp. 221–228 (2012). doi:10.1109/CLOUD.2012.12
57. Fang, W., Lu, Z., Wu, J., Cao, Z.: RPPS: a novel resource prediction and provisioning scheme in cloud data center. In: 2012 IEEE Ninth International Conference on Services Computing, IEEE, pp. 609–616 (2012). doi:10.1109/SCC.2012.47
58. Gambi, A., Toffetti, G.: Modeling cloud performance with Kriging. In: 2012 34th International Conference on Software Engineering, (ICSE), IEEE, pp. 1439–1440 (2012). doi:10.1109/ICSE.2012.6227075
59. Ghanbari, H., Simmons, B., Litoiu, M., Iszlai, G.: Exploring alternative approaches to implement an elasticity policy. In: 2011 IEEE International Conference on Cloud Computing (CLOUD), pp. 716–723 (2011)
60. Gong, Z., Gu, X., Wilkes, J.: Press: predictive elastic resource scaling for cloud systems. In: 2010 International Conference on Network and Service Management (CNSM), pp. 9–16 (2010)
61. Guitart, J., Torres, J., Ayguadé, E.: A survey on performance management for internet applications. Concurrency

and Computation: Practice and Experience **22**(1), 68–106 (2010). doi:10.1002/cpe.1470

62. Han, R., Ghanem, M.M., Guo, L., Guo, Y., Osmond, M.: Enabling cost-aware and adaptive elasticity of multi-tier cloud applications. Futur. Gener. Comput. Syst. **32**, 82–98 (2012)
63. Han, R., Guo, L., Ghanem, M., Han, R., Guo, L., Ghanem, M.M., Guo, Y.: Lightweight Resource Scaling for Cloud Applications. Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on (2012)
64. Hasan, M.Z., Magana, E., Clemm, A., Tucker, L., Gudreddi, S.L.D.: Integrated and autonomic cloud resource scaling. In: Network Operations and Management Symposium (NOMS), 2012 IEEE, IEEE, pp. 1327–1334 (2012)
65. Huang, J., Li, C., Yu, J.: Resource prediction based on double exponential smoothing in cloud computing. In: 2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet), pp. 2056–2060 (2012)
66. Iqbal, W., Dailey, M.N., Carrera, D., Janecek, P.: Adaptive resource provisioning for read intensive multi-tier applications in the cloud. Futur. Gener. Comput. Syst. **27**(6), 871–879 (2011). doi:10.1016/j.future.2010.10.016
67. Islam, S., Keung, J., Lee, K., Liu, A.: Empirical prediction models for adaptive resource provisioning in the cloud. Futur. Gener. Comput. Syst. **28**(1), 155–162 (2012). doi:10.1016/j.future.2011.05.027
68. Jacob, B., Lanyon-Hogg, R., Nadgir, D.K., Yassin, A.F.: A practical guide to the IBM autonomic computing toolkit (2004)
69. Kalyvianaki, E., Charalambous, T., Hand, S.: Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In: Proceedings of the 6th International Conference on Autonomic Computing, ACM, pp. 117–126 (2009)
70. Kertesz, A., Kecskemeti, G., Oriol, M., Kotcauer, P., Acs, S., Rodríguez, M., Mercè, O., Marosi, A.C., Marco, J., Franch, X.: Enhancing Federated Cloud Management with an Integrated Service Monitoring Approach. J. Grid Comput. (2013)
71. Khatua, S., Ghosh, A., Mukherjee, N.: Optimizing the utilization of virtual resources in cloud environment. In: 2010 IEEE International Conference on Virtual Environments, Human-Computer Interfaces and Measurement Systems, IEEE, pp. 82–87 (2010). doi:10.1109/VECIMS.2010.5609349
72. Koperek, P., Funika, W.: Dynamic business metrics-driven resource provisioning in cloud environments. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) Parallel Processing and Applied Mathematics, Lecture Notes in Computer Science, vol. 7204, pp. 171–180. Springer, Berlin Heidelberg (2012). doi:10.1007/978-3-642-31500-8_18
73. Kupferman J, Silverman J, Jara P, Browne J. Scaling into the cloud. Tech. rep., University of California, Santa Barbara; CS270 - Advanced Operating Systems (2009). http://cs.ucsb.edu/jkupferman/docs/ScalingIntoTheClouds.pdf
74. Lama, P., Zhou, X.: Autonomic Provisioning with Self-Adaptive Neural Fuzzy Control for End-to-end Delay Guarantee. In: 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, IEEE, pp. 151–160 (2010). doi:10.1109/MASCOTS.2010.24
75. Lim, H.C., Babu, S., Chase, J.S., Parekh, S.S.: Automated control in cloud computing: challenges and opportunities. In: Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds, ACM, New York, NY, USA, ACDC ’09, pp. 13–18 (2009). doi:10.1145/1555271.1555275
76. Lim, H.C., Babu, S., Chase, J.S.: Automated control for elastic storage. In: Proceeding of the 7th International Conference on Autonomic Computing - ICAC ’10, p. 1. ACM Press, New York (2010)
77. Lorido-Botran, T., Miguel-Alonso, J., Lozano, J.A.: Comparison of Auto-scaling Techniques for Cloud Environments. In: Alberto, A., Del Barrio, G.B. (eds.) Actas de las XXIV Jornadas de Paralelismo, Servicio de Publicaciones (2013)
78. Manvi, S.S., Shyam, G.K.: Resource management for Infrastructure as a Service (IaaS) in cloud computing: A survey. J. Netw. Comput. Appl. **41**, 424–440 (2014)
79. Mao, M., Humphrey, M.: Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC ’11, p. 1. ACM Press, New York (2011)
80. Mao, M., Humphrey, M.: A performance study on the VM startup time in the cloud. In: Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing, IEEE Computer Society, Washington, DC, USA, CLOUD ’12, pp. 423–430 (2012). doi:10.1109/CLOUD.2012.103
81. Maurer, M., Brandic, I., Sakellariou, R.: Enacting slas in clouds using rules. Euro-Par. Parallel Processing (2011)
82. Maurer, M., Breskovic, I., Emeakaroha, V.C., Brandic, I.: Revealing the MAPE loop for the autonomic management of Cloud infrastructures. In: 2011 IEEE Symposium on Computers and Communications (ISCC), pp. 147–152 (2011). doi:10.1109/ISCC.2011.5984008
83. Menasce, D.A., Dowdy, L.W., Almeida, V.A.F.: Performance by Design: Computer Capacity Planning By Example. Prentice Hall, Upper Saddle River (2004)
84. Méndez Muñoz, V., Casajús Ramo, A., Fernández Albor, V., Graciani Diaz, R., Merino Arévalo, G.: Rafhyc: an architecture for constructing resilient services on federated hybrid clouds. J. Grid Comput. **11**(4), 753–770 (2013). doi:10.1007/s10723-013-9279-y
85. Mi, H., Wang, H., Yin, G., Zhou, Y., Shi, D., Yuan, L.: Online self-reconfiguration with performance guarantee for energy-efficient large-scale cloud computing data centers. In: 2010 IEEE International Conference on Services Computing (SCC), pp. 514–521 (2010)
86. Moore, L.R., Bean, K., Ellahi, T.: Transforming reactive auto-scaling into proactive auto-scaling. In: Proceedings of the 3rd International Workshop on Cloud Data and

Platforms, ACM, New York, NY, USA, CloudDP ’13, pp. 7–12 (2013). doi:10.1145/2460756.2460758

87. Ostermann, S., Plankensteiner, K., Prodan, R., Fahringer, T.: GroudSim: An event-based simulation framework for computational grids and clouds. In: Guarracino, M., Vivien, F., Träff, J., Cannatoro, M., Danelutto, M., Hast, A., Perla, F., Knüpfer, A., Martino, B., Alexander, M. (eds.) Euro-Par 2010 Parallel Processing Workshops, Lecture Notes in Computer Science, vol. 6586, pp. 305–313. Springer, Berlin Heidelberg (2011). doi:10.1007/978-3-642-21878-1_38
88. Padala, P., Hou, K.Y., Shin, K.G., Zhu, X., Uysal, M., Wang, Z., Singhal, S., Merchant, A.: Automated control of multiple virtualized resources. In: Proceedings of the 4th ACM European Conference on Computer Systems, ACM, pp. 13–26 (2009)
89. Park, S.M., Humphrey, M.: Self-tuning virtual machines for predictable eScience. In: 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, IEEE, pp. 356–363 (2009). doi:10.1109/CCGRID.2009.84
90. Patikirikorala, T., Colman, A.: Feedback controllers in the cloud. APSEC 2010, Cloud Workshop (2010)
91. Prodan, R., Nae, V.: Prediction-based real-time resource provisioning for massively multiplayer online games. Futur. Gener. Comput. Syst. **25**(7), 785–793 (2009). doi:10.1016/j.future.2008.11.002
92. Rao, J., Bu, X., Xu, C.Z., Wang, L., Yin, G.: VCONF: a reinforcement learning approach to virtual machines auto-configuration. In: Proceedings of the 6th International Conference on Autonomic Computing, ACM, New York, NY, USA, ICAC ’09, pp. 137–146 (2009). doi:10.1145/1555228.1555263
93. Rao, J., Bu, X., Xu, C.Z., Wang, K.: 8. In: 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems, IEEE, pp. 45–54 (2011). doi:10.1109/MASCOTS.2011.47
94. Roy, N., Dubey, A., Gokhale, A.: Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In: 2011 IEEE 4th International Conference on Cloud Computing, IEEE, pp. 500–507 (2011). doi:10.1109/CLOUD.2011.42
95. Shen, Z., Subbiah, S., Gu, X., Wilkes, J.: Cloudscale: Elastic resource scaling for multi-tenant cloud systems. Proceedings of the 2nd ACM Symposium on Cloud Computing (2011)
96. Simmons, B., Ghanbari, H., Litoiu, M., Iszlai, G.: Managing a SaaS application in the cloud using PaaS policy sets and a strategy-tree. In: 2011 7th International Conference on Network and Service Management (CNSM), pp. 1–5 (2011)
97. SPECweb2009, (2012) The httperf HTTP load generator. http://www.spec.org/web2009/. Online: Accessed 18 Sept 2012
98. Sutton, R.S., Barto, A.G.: Introduction to Reinforcement Learning. Cambridge University Press (1998)
99. Tesauro, G., Jong, N.K., Das, R., Bennani, M.N.: A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation. In: Proceedings of the 2006 IEEE International Conference on Autonomic Computing, IEEE Computer Society, Washington, DC, USA, ICAC ’06, pp. 65–73 (2006). doi:10.1109/ICAC.2006.1662383
100. Urgaonkar, B., Shenoy, P., Chandra, A., Goyal, P., Wood, T.: Agile dynamic provisioningof multi-tier Internet applications. ACM Trans. Auton. Adapt. Syst. **3**(1), 1–39 (2008). doi:10.1145/1342171.1342172
101. Villela, D., Pradhan, P., Rubenstein, D.: Provisioning servers in the application tier for e-commerce systems. In: Twelfth IEEE International Workshop on Quality of Service, 2004. IWQOS 2004, pp. 57–66 (2004)
102. Wang, L., Xu, J., Zhao, M., Fortes, J.: Adaptive virtual resource management with fuzzy model predictive control. In: Proceedings of the 8th ACM International Conference on Autonomic Computing - ICAC ’11, p. 191. ACM Press, New York (2011)
103. Wang, L., Xu, J., Zhao, M., Tu, Y., Fortes, J.A.B.: Fuzzy Modeling Based Resource Management for Virtualized Database Systems. In: 2011 IEEE 19th International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), pp. 32–42 (2011)
104. Watkins, C., Dayan, P.: Q-learning. Machine Learning (1992)
105. Xu, C.Z., Rao, J., Bu, X.: URL: A unified reinforcement learning approach for autonomic cloud management. J. Parallel Distrib. Comput. **72**(2), 95–105 (2012). doi:10.1016/j.jpdc.2011.10.003
106. Xu, J., Zhao, M., Fortes, J., Carpenter, R., Yousif, M.: On the Use of Fuzzy Modeling in Virtualized Data Center Management. In: Proceedings of the Fourth International Conference on Autonomic Computing, IEEE Computer Society, Washington, DC, USA, ICAC ’07, p. 25 (2007). doi:10.1109/ICAC.2007.28
107. Zhang, Q., Cherkasova, L., Smirni, E.: A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In: Fourth International Conference on Autonomic Computing, ICAC’07, p. 27 (2007)
108. Zhu, Q., Agrawal, G.: Resource Provisioning with Budget Constraints for Adaptive Applications in Cloud Environments. IEEE Trans. Serv. Comput. **5**(4), 497–511 (2012). doi:10.1109/TSC.2011.61