Auto-scaling Techniques for Elastic Applications in Cloud Environments

Tania Lorido-Botrán, José Miguel-Alonso, José A. Lozano

Technical Report EHU-KAT-IK-09-12

Department of Computer Architecture and Technology
University of the Basque Country

September 5, 2012

Contents

1 Introduction **3**

2 Problem Definition **5**

3 Performance Evaluation in the Cloud: Experimental Platforms, Workloads and Application Benchmarks **7**
- 3.1 Experimental Environments . . . 7
- 3.2 Synthetic Workloads . . . 8
- 3.3 Real Traces . . . 9
- 3.4 Application Benchmarks . . . 10

4 Classification for Auto-scaling Techniques **11**

5 Review of Auto-scaling Techniques **13**
- 5.1 Static Threshold-based Rules . . . 14
  - 5.1.1 Definition of the Technique . . . 14
  - 5.1.2 Review of Proposals . . . 15
- 5.2 Reinforcement Learning (Q-Learning) . . . 16
  - 5.2.1 Definition of the Technique . . . 16
  - 5.2.2 Review of Proposals . . . 20
- 5.3 Queuing Theory . . . 21
  - 5.3.1 Definition of the Technique . . . 21
  - 5.3.2 Review of Proposals . . . 24
- 5.4 Control Theory . . . 25
  - 5.4.1 Definition of the Technique . . . 25
  - 5.4.2 Review of Proposals . . . 27
- 5.5 Time-series Analysis . . . 29
  - 5.5.1 Definition of the Technique . . . 30
  - 5.5.2 Review of Proposals . . . 32

6 Discussion and Open Research Lines **34**

7 Conclusions and Future Work **36**

1 Introduction

Cloud computing is an emerging technology that is becoming more and more popular. This is due primarily to its elastic nature: users can acquire and release resources on-demand, and pay only for the resources they need (pay-per-use or pay-as-you-go model). Those resources are usually in the form of Virtual Machines (VM). Companies can use clouds for different purposes, such as running batch jobs, hosting web applications or for backup and storage. Three main markets are associated to cloud computing:

- Infrastructure-as-a-Service (IaaS) designates the provision of IT and network resources such as processing, storage and bandwidth as well as management middleware. Examples are Amazon EC2 [3], RackSpace [26] and the new Google Compute Engine [17].
- Platform-as-a-Service (PaaS) designates programming environments and tools supported by cloud providers that can be used by consumers to build and deploy applications onto the cloud infrastructure. Examples of PaaS include Amazon Elastic Beanstalk [4], Heroku [20], Force.com [11], Google App Engine [12], and Microsoft Windows Azure [38].
- Software-as-a-Service (SaaS) designates hosted vendor applications. For example, Google Apps [13] (such as Google Docs, Google Calendar or Google Sites), Microsoft Office 365 [24] and Salesforce.com [31].

In the current work, we will focus on the IaaS client's perspective. A typical scenario could be a user that wants to host a web application, and for this purpose contracts several resources from a known IaaS provider such as Amazon EC2. From now on we will use the following terminology:

- **Provider**. It refers mainly to the IaaS provider, that offers virtually *unlimited* resources in the form of VMs. It could also apply to a PaaS provider, although they sometimes offer limited scaling capacity, that usually cannot be configured by the user.
- **Client**. The client is the user of the IaaS or PaaS service, that uses it for hosting the application. In other words, it is the application owner or provider.
- **User**. It is the final user that accesses the web application.

A real case could be the Groupon [19] website that is deployed on both the Amazon EC2 infrastructure and the Force.com PaaS provider, enabling it to adjust seamlessly to a changing demand [2]. The Groupon website is the *client* of two *providers*: Amazon EC2 and Force.com. The website presents daily several offers and discounts to its *users*. Those users can visit Groupon anytime from their browser. But Groupon is just an example: Amazon EC2 presents a large list of client companies which use the infrastructure for diverse purposes

[5] such as application hosting, web hosting, e-commerce, search engines, High Performance Computing (HPC) or simply for backup and storage service.

As we said previously, the key characteristic of cloud computing is *elasticity*. However, it is also a double-edged sword. Elasticity allows users to acquire and release resources dynamically according to changing demands, but deciding the right amount of resources is not an easy task. Indeed, appropriately dimensioning resources to applications is a crucial issue in cloud computing. Many web applications face large, fluctuating loads. In predictable situations (new campaigns, seasonal), resources can be provisioned in advance through capacity planning techniques. But for unplanned, spike loads, it would be desirable an automatic scaling system (or **auto-scaling system**) that adjusts resources allocated to an application based on its needs at any given time. This would free the user from the burden of deciding how many resources are necessary at each time.

Resource allocation actions can focus on horizontal scaling: i.e adding new server replicas and load balancers to distribute load among all available replicas, or vertical scaling: on-the-fly changing the resources assigned to an already running instance, for example, allocating more physical CPU or memory to a running VM). Unfortunately, the most common operating systems do not support on-the-fly (without rebooting) changes on the available CPU or memory to support this vertical scaling. For this reason, most of the cloud providers only offer horizontal scaling.

The final objective of an auto-scaling system is to automatically adjust acquired resources to minimize cost while complying with the SLO. To do this, it must take into account:

- Cloud provider prizing model. For example, Amazon offers three charge models: pay-per-use model, reserved instance pricing model with long-term commitment of availability, and spot instances. Typically, several VM types are offered, with a pre-configured set of resources (ideal scenario for horizontal scaling).

- Unit charge: Cloud providers usually charge the user per hour of VM use. There are some particular cases such as Windows Azure that charges users per *natural* hour, i.e, if an VM is started at 10.55AM and stopped at 11.05AM, the user will pay for 2 hours. Generally, partial hour usage is rounded up to one hour. Therefore, even if the load is low and machine is not required, the entire hour has been paid for, so there is no reason to terminate it before the hour is over. This is sometimes called *smart kill*.

- VM boot-up time o acquisition lag: It takes several minutes for a new VM to become ready to operate. This needs to be taken into account in auto-scaling.

The problem of automatic scaling can be addressed using different approaches. These have been reviewed in the current work. The remainder of the paper is organized as follows. Section 2 provides a general definition of the auto-scaling problem. Section 3 covers the details about the experimental platform, workload generation and application benchmarks used to test the different algorithms. A classification for the different auto-scaling techniques is introduced in Section 4. Section 5 further describes each of these auto-scaling techniques. Finally, we conclude with an outline of research lines (Section 6) and several conclusions extracted from the study (Section 7).

2 Problem Definition

We will consider a typical web application with a 3-tier architecture (see Figure 1), managed by a IaaS user and deployed on a IaaS infrastructure:

- Load balancer (LB): It receives all the incoming requests from users and routes them to the application servers.
- Business-logic tier (BT): It contains the application servers that execute the application logic.
- Storage or persistence tier (ST): It refers to the database system.

Each tier can be scaled separately, but we will focus on scaling the business tier (the application servers).

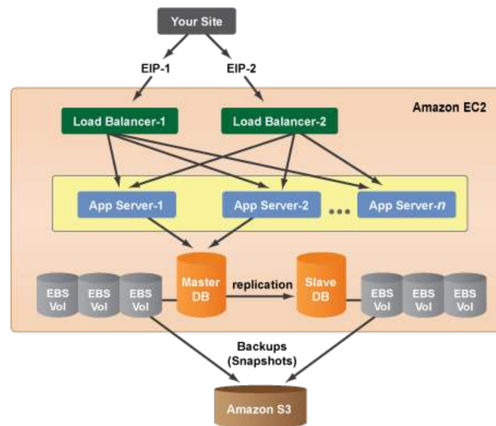

Figure 1: Web application with a 3-tier architecture[1].

Let us consider an application deployed over a pool of n VMs. VMs may have the same or different resource assignment, but each VM has its own unique identifier (it could be the IP address). Final users will send requests to the load balancer. Those requests can be either considered as simple independent jobs, with homogeneous or heterogeneous processing times, or as part of sophisticated models of user behavior, which consider sequences of actions called sessions (e.g. login, list, order, ...), each one with a different duration, thinking times, and so on. The load balancer will receive all the incoming requests and forward them to one of the servers in the pool. Several balancing policies can be considered such as random, round-robin and least-connection. Each request will be assigned to a unique server.

Each application server will be hosted in a different VM. We assume that servers can process several requests simultaneously, but note that the service time will increase depending

[1]Figure taken from `http://support.rightscale.com/03-Tutorials/02-AWS/02-Website_Edition/Set_up_Autoscaling_using_Voting_Tags`

on the number of concurrent requests. No that in a context in which we only scale horizontally the business-logic tier, we may use *server* and *VM* with the same meaning.

The auto-scaling system requires two elements: a monitor and the scaling unit. Any auto-scaling method needs a good monitoring system, gathering different and updated metrics about system and application current state, and at a suitable granularity (e.g. per second, per minute). Ghanbari et al. [56] propose a list of **performance metrics** or variables for scaling purposes:

- Hardware: CPU utilization, disk access, network interface access, memory usage.
- General OS Process: CPU-time, 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.
- Web server: transmitted bytes and requests, number of connections in specific states (e.g. closing, sending, waiting, starting, ...).
- Application server: total threads count, active threads count, used memory, session count, processed requests, pending requests, dropped requests, response time.
- Database server: number of active threads, number of transactions in a particular state (write, commit, roll-back, ...) .

The scaling unit will use this information to decide the scaling action to be performed, e.g. remove a VM or add some extra memory. Each scaling action should be decided taking into account the different cloud provider prizing models and the VM boot-up time. The final *objective* is to find a trade-off between meeting Service Level Objectives or SLO (for example, 99.9% of availability or a maximum response time of 2 seconds) and minimizing the cost of renting cloud resources.

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

Before proceeding to introduce the different auto-scaling techniques proposed in the literature, it is necessary to describe the variety of experimental platforms that have been used. There is no standard method for evaluating auto-scaling techniques, and researchers in the field have built their own testing environments, suitable to their own needs. However, there is a common pattern for all of them: evaluation requires a real or realistic environment, using simulators, real cloud providers or custom testbeds.

Regardless the selected experimental platform, controlled workloads are required to drive the experiments. The term *workload* refers to a list of user requests, together with the arrival timestamp. The workload can be either synthetic, generated with specific programs, or obtained from real cloud platforms and stored in *trace* files (traces, for short). Both synthetic workloads and real traces will be described in the current section.

Apart from selecting a suitable workload, an application benchmark is required to execute the input requests in real cloud providers or custom testbeds. We will also review the benchmarks most commonly found in the literature.

3.1 Experimental Environments

Experimentation could be done in *production* infrastructures, either from real cloud providers or in a private cloud. The major advantage 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 a real provider, the infrastructure is already given, but we still need 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 avoid the experimentation cost and to have a more controlled environment, we could utilize a *custom testbed*. This has a cost in terms of system configuration effort. The most relevant step consists of installing the virtualization software, that will manage the VM creation, scaling and so on. Virtualization can be applied at the server level, OS level or at the application level. For custom testbeds, virtualization at the server level is needed, commonly referred as *Hypervisor* or *Virtual Machine Monitor* (VMM). Some popular hypervisors include Xen [40], VMWare ESXi [37] and Kernel Virtual Machine (KVM) [23]. There are several platforms for deploying custom clouds, including open-source alternatives such as Eucalyptus [9] and OpenStack [25], and commercial software like vCloud Director [36]. OpenStack is an open-source initiative supported by many enterprises such as RackSpace, HP, Intel and AMD, but still under development. 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.

In contrast to real infrastructure, we could use software to *simulate* the functioning of a cloud platform, including resource allocation and deallocation, VM execution, monitoring and the remaining cloud management tasks. We could select an already existing simulator and adapt it to our needs, or implement a custom software from scratch. Obviously, using a simulator implies an initial effort to prepare the software but, in contrast, has many advantages. The evaluation process is shortened in many ways. It makes possible to test multiple algorithms without having to re-configure all the infrastructure each time. Besides, the simulator prevents the influence from external factors (this would be impossible in a real cloud provider). Experiments carried out in real infrastructures may last hours, whereas in an event-based simulator, this process may only take minutes. Simulators are highly configurable and allow the user 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 [18], and GroudSim [73].

3.2 Synthetic Workloads

Synthetic workloads can be generated based on different patterns. According to Mao and Humphrey [69], there are four representative workload patterns in the cloud environment: 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. The Growing workload pattern may represent a scenario in which a piece of news or a video suddenly becomes popular, or the consequent *Slashdot effect*. The Cyclic/Bursting workload may represent the workload pattern of an online retailer, in which daytime has more workload than the night and holiday shopping seasons might handle more traffic than normal. The On-and-off workload pattern represents the work to be processed periodically or occasionally, such as batch processing and data analysis performed everyday.

There is a broad range of workload generators, that can be used to generate simple requests based on any of the patterns mentioned above, 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* [10]: A Markov-based workload generator, included in the CloudStone stack.
- *Apache JMeter* [22]: A Java workload generator used for load testing and measuring performance. It can be used to test performance both on static and dynamic resources (files, Servlets, Perl scripts, Java Objects, Data Bases 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 overall performance under different scenarios.
- *Rain* [27]: A statistics-based workload generation toolkit that uses parameterized and empirical distributions to model the different classes of workload variations.

- *Httperf* [21]: 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, we can tune the workload 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 traces from real production systems.

3.3 Real Traces

Application workloads for cloud-based systems can be separated into two classes: batch and transactional. Batch workloads consist of arbitrary, long running, resource-intensive jobs, such as text mining, video transcoding and graphical rendering. The most well-known examples of transactional workloads are web applications built to serve online 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.

To the best of our knowledge, there are no publicly available, real traces from cloud providers, 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] or the World Cup 98 trace [39].

The ClarkNet trace [6] contains the HTTP requests received by the ClarkNet server over a two-weeks period in 1995. ClarkNet is a full Internet access provider for the metro Baltimore-Washington DC area. It shows a clear cyclic workload pattern (see Figure 2): daytime has more workload than the night, and the workload on weekends is lower than that taking place on weekdays.

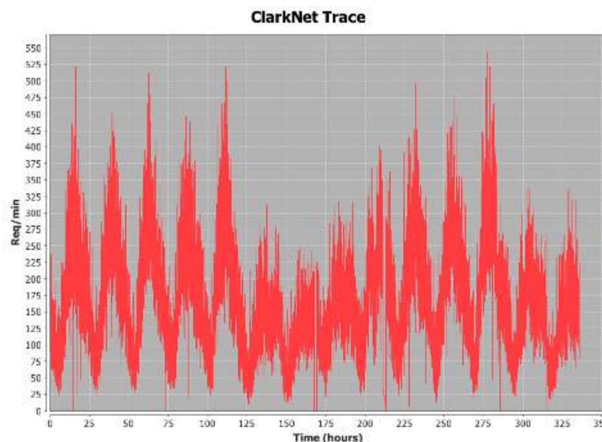

Figure 2: Number of requests per minute for ClarkNet Trace.

The World Cup 98 trace [39] 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.

Some authors have used traces from Grid environments (Caron et al. [47]), but although there is an extensive number of public traces, the job execution scheme is not suitable for evaluating web applications. However, they could be useful for batch-based workloads.

It is also worth mentioning the Google Cluster Data [14], two sets of traces that contain the workloads running on Google compute cells. The first dataset [15] refers to a 7-hour period and consists of a set of tasks. However, the data have been anonymized, and the CPU and RAM consumption have been normalized and obscured using a linear transformation. The second trace [16] 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. Similarly to the first trace, all the numeric data have been normalized, and there is no information about the job type. For this reason, no Google trace can be utilized to test auto-scaling techniques, but they would be useful in other scenarios such as the IaaS level.

3.4 Application Benchmarks

Application benchmarks are used to evaluate server's performance and scalability. Typically, they 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 [35] and CloudStone [8]. 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.com. 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 in 2008.

- *TPC-W* [35]: TPC [33] is a is a non-profit 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-ased 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, multi-language performance measurement tool for Web 2.0 and Cloud Computing, developed by the Rad Lab group at the University of Berkeley. CloudStone involves using a flexible, realistic workload generator (Faban) to generate load against a realistic Web 2.0 application (Olio). The stack is deployed

on Amazon EC2 instances. As explained before, Faban is an open-source Markov-chain, session-based synthetic workload generator, while 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.

There are other, less used benchmarks such as SpecWeb [82], TPC-C [34] and RUBBoS [30]. SpecWeb is a benchmark tool, created by the Standard Performance Evaluation Corporation (SPEC), that is able to create banking, e-commerce and support (large downloads) workloads. SpecWeb has been discontinued in early 2012 and now the SPEC company has created a cloud benchmarking group [32]. 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. RUBBoS is a bulletin board benchmark, modeled after an online news forum like Slashdot. The last update was in 2005.

4 Classification for Auto-scaling Techniques

Managing cloud computing elasticity is typically a per-application task and it implies mapping performance requirements to the underlying available resources. This process of adapting resources to the on-demand requirements of an application, called *scaling*, can be very challenging. Resource under-provisioning will inevitably hurt performance and create SLO violations, while resource over-provisioning can result in idle instances, thereby incurring unnecessary costs.

The first thought could lead us to plan capacity for the average load or for the peak load. When planed for the average load, there is less cost incurred, but performance will be a problem if peaks of load occurs. Bad performance will discourage customers, and revenue will be affected. On the other hand if capacity is planned for peak workload, resources will remain idle most of the time.

Therefore, it seems necessary a more sophisticated technique for resource allocation, that automatically scales resources according to demand. These are called *auto-scaling techniques*. To date, cloud practitioners have pursued schedule-based and rule-based mechanisms to attempt to automate this matching between computing requirement and computing resources. Schedule-based techniques take into account the cyclical pattern of the daily workload. The scaling actions are configured manually, based on the time of the day, so the system cannot adapt to the unexpected changes in the load. For this reason, schedule-based techniques will not be discussed in this work.

Well-known cloud providers such as Amazon EC2 usually offer rule-based auto-scaling. This simple approach typically involves creating two rules to determine when to scale. For each rule, the user has to define a condition based on a target variable, for example CPU load $> 80\%$. When the condition is met, it triggers a pre-defined scaling up or scaling-down action; e.g. add a new VM. The rule-based approach can be classified as a *reactive*

algorithm, which means that it *reacts* to system changes, but do not anticipate to them. In contrast, *predictive* or *proactive* auto-scaling techniques try to anticipate to future needs and consequently acquire or release resources in advance, to have them ready when they are needed. If we consider the usage of a particular resource (CPU, memory,...) or input workload as a time series, we can think of several predictive alternatives. For example, we could consider the mean over the last W samples in the time series as the predicted resource demand. Alternatively, we could consider the maximum resource usage over the samples in the window W. Of course, using the mean or the max value is not always a good option, but there are several alternatives that will be further discussed later.

In the literature, auto-scaling for cloud computing has been extensively discussed from several points of view, including both predictive and reactive techniques. Authors have put their focus on the different layers of cloud computing: IaaS, PaaS and SaaS, but on the current study, we will concentrate on the IaaS client's perspective. Hence, IaaS management is out of our scope, including tasks such as VM migration, physical allocation of VMs in a datacenter or server consolidation. As defined in Section 2, the target scenario is an application deployed over a pool of VMs, with theoretically unlimited resources to be scaled. Throughout this paper we will focus on the middle (business) tier, but most techniques are applicable to all tiers (e.g. Lim et al. [67] designed an auto-scaling algorithm for the business tier, and later applied the same technique to the storage layer [68]). Some specific papers consider the scaling task for all three application tiers as a whole. In particular, Urgaonkar et al. [84] claim that adding servers to the bottleneck tier does not necessarily solve the problem, but just shifts the bottleneck to a downstream tier.

Cloud providers offer different billing options, that usually follow a pay-as-you-go model. Amazon EC2 billing schemes are the most commonly used in the literature. The IaaS provider offers on-demand, reserved and spot instances, to satisfy the different needs of the users. Spot instances enable the users to bid on unused Amazon EC2 capacity, at lower prices than regular instances. Amazon sets a *spot price* that changes based on supply and demand, and customers whose bids exceeds this spot price gain access to the available spot instances. Some research has been done in this area, proposing methods to guide users with the bidding (e.g. Andrzejak et al. [42]). A second billing scheme offered by Amazon EC2 are reserved instances, intended for long-term use. The user initially pays a fixed amount for each instance, and the per-hour fee is greatly reduced, in comparison to on-demand instance price. In the current work, we will focus on standard on-demand instances, that are charged on an hourly basis.

It is difficult to work out a proper classification of auto-scaling techniques, due to the wide diversity of approaches found in the literature, that are sometimes hybridizations of two or more methods. Considering the anticipation capacity as the main criteria, techniques could be divided into two main classes: reactive and proactive. Threshold-based policies clearly belong to the reactive category, whereas time-series analysis is a purely proactive approach. In contrast, reinforcement learning, queuing theory and control theory can be used with both reactive and proactive approaches. In the present review, we will consider auto-scaling techniques grouped into these categories:

1. Static, threshold-based policies
2. Reinforcement Learning
3. Queuing theory
4. Control theory
5. Time-series analysis

Each technique will be described separately, and then the literature using that technique will be discussed. As some articles compare two or more auto-scaling proposals, they may be cited in several sections.

5 Review of Auto-scaling Techniques

The proposed classification for auto-scaling techniques covers different areas of knowledge, including simple threshold-based rules, reinforcement learning, queuing theory, control theory and time-series analysis. Each of the categories usually includes many diverse methods, that can follow either a reactive or a proactive approach.

Reactive techniques, as the name suggests, refer to the set of methods that *react* to the current system and/or application state. Decisions are taken based on the last values obtained from a set of monitored variables. Among the mentioned categories, threshold-based rules or policies follow a purely reactive approach. They are the most extended auto-scaling technique, including a small variation used by the RightScale [28] vendor.

The lack of anticipation of a reactive approach clearly affects the auto-scaling performance. The system might not be able to scale proportionally with the *Slashdot effect* or sudden traffic bursts resulting from special offers or market campaigns. In addition to this, the time it takes to instantiate a new VM can be up to 15 minutes, and the effect of a scaling-up action might arrive too late. Therefore, proactive or prediction-based resource scaling is required in order to deal with the ever fluctuating resource usage pattern, and being able to scale in advance. Among the categories in the classification, time-series analysis covers a wide range of methods that follow a proactive approach and seems to be a promising research area. This category considers all auto-scaling approaches that use the past history of a time-series to predict future values.

The rest of the auto-scaling techniques (control theory, reinforcement learning and queuing theory) cannot be clearly classified into a reactive or proactive approach. Classic queuing theory requires modeling each application VM (or even application tier) as a queue of requests, and there are established methods to estimate performance metrics for every scenario. In contrast, some reinforcement learning algorithms are able to cope with the auto-scaling task, without any *a priori* knowledge or system model, but the time for the method to converge to an optimal policy can be unfeasibly long. The last auto-scaling type included in the classification is the control theory area. It involves creating a reactive or proactive controller to automatically adjust the required resources to the application demand.

Each of the auto-scaling techniques will be discussed in the current section, also considering their limitations when applied to the scaling task.

5.1 Static Threshold-based Rules

Threshold-based rules or policies are very popular among cloud providers such as Amazon EC2, and third-party tools like RightScale [28]. The simplicity and intuitive nature of these policies make them very appealing to users. However, setting thresholds is a per-application task, and requires a deep understanding of workload trends.

5.1.1 Definition of the Technique

The number of VMs in the target application (defined in Section 2) will vary according to a set of rules, typically two: one for scaling up and one for scaling down. Those rules can use one or more performance metrics, such as the request rate, CPU load or average response time. Each rule or policy involves several parameters defined by the user: an upper threshold *thrUp*; a lower threshold *thrDown*; two time values *vUp* and *vDown* that define how long the condition must be met to trigger a scaling action, and (optionally) two inertia durations: *inUp* for scaling up and *inDown* for scaling down. The upper and lower thresholds must be defined for each performance metric x. The scaling actions differ depending on the type of scaling. For horizontal scaling, the user should define a fixed amount s of VMs to be allocated or deallocated, but for vertical scaling, the same variable s refers to the amount of resource that will be added (CPU, RAM,...). The parameters *vUp*, *vDown*, *inUp* and *inDown* are optional and not included in all models. The resulting rules will have a structure similar to these:

$$\begin{aligned} &\text{if } x > thrUp \text{ for } vUp \text{ seconds then} \\ &\quad n = n + s \text{ and} \\ &\quad \text{do nothing for } inUp \text{ seconds} \end{aligned} \tag{1}$$

$$\begin{aligned} &\text{if } x < thrDown \text{ for } vDown \text{ seconds then} \\ &\quad n = n - s \text{ and} \\ &\quad \text{do nothing for } inDown \text{ seconds} \end{aligned} \tag{2}$$

The best way to understand static threshold-based rules is by means of an example: *add 2 small instances when the average CPU is above 70% for more than 5 minutes.*

Focusing on horizontal scaling, the user can define a maximum $nMax$ and a minimum $nMin$ number of VMs, both to control the overall cost and to guarantee a minimum level of availability. At each iteration, if the performance metric x violates *thrUp* for *vUp* seconds, s VMs are requested and added to the pool of application servers. Then, the auto-scaling system inhibits itself for *inUp* seconds. If the performance metric x goes below *thrDown* for *vDown* seconds, s VMs are removed and their resources released. Again, the system inhibits itself for *inDown* seconds.

5.1.2 Review of Proposals

Threshold-based rules can easily manage the amount of resources assigned to an application hosted in a cloud platform, and perform auto-scaling actions to adapt resources to the input demand (e.g. [54], [70], [58], [59]). However, setting the rules requires an extra effort from the user, who needs to select the suitable performance variable or logical combination of variables, and also to set several parameters. Among those parameters, the upper and lower thresholds for the performance variable (e.g. 30% and 70% of CPU load) 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 also advisable to set an *inertia*, *cooldown* or *calm* period, a time during which no scaling decisions can be committed.

Most authors and cloud providers use only two thresholds per performance metric. However, Hasan et al. [59] have considered using a set of four thresholds and two durations: *ThrU*, the upper threshold; *ThrbU*, which is slightly below the upper threshold; *ThrL*, the lower threshold; *ThroL*, which is slightly above the lower threshold; and two durations (in seconds) used for checking persistence of metric value above/below *ThrU*/*ThrL* and *ThrbU*/*ThroL*. This parameter configuration better tracks the metric value trends: trending up or down with persistence, trending down from above,... The system can then perform finer auto-scaling decisions than using only the two common thresholds.

Conditions in the rules are usually based on one 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. [58] use the average response time of the application. On the contrary, Hasan et al. [59] prefer using performance metrics from multiple domains (compute, storage and network) or even a correlation of some of them, for example:

- Add a new VM when both CPU Load and the response time from a customer edge router are high (above certain thresholds)
- Increase the bandwidth of the network link resource when both the response time and the network link load are high.

A complement to reactive rules is *RightScale's auto-scaling algorithm* [29]. It is a simple democratic voting process whereby, if a majority of the VMs agree that they should scale up or down, that action is taken; otherwise no action occurs. Each VM votes to scale up or down based on a set of rules. After each scaling action, there is a period called the resize calm time (equivalent to the inertia or cooldown time), where no action can be performed. It prevents the algorithm from continually allocating resources as new instances boot. RightScale recommends 15 minute-period of calm time because new machines generally take between 5 to 10 minutes to start.

This auto-scaling technique that combines rules and a voting system has been adopted by several authors (Kupferman et al. [65], Chieu et al. [51], Ghanbari et al. [56], Simmons

et al. [81]). Chieu et al. [51] initially proposed a set of reactive rules based on the number of active sessions, but this work was extended in Chieu et al. [52] following the RightScale approach: if all instances have active sessions above the given upper threshold, a new instance is provisioned; if there are instances with active sessions below a given lower threshold and with at least one instance that has no active session, the idle instance will be shut down.

As RightScale's voting system is based on rules, it has the same disadvantage: the algorithm is highly dependent on user-defined threshold values, and therefore, to the workload characteristics. This was the conclusion reached by Kupferman et al. [65] after comparing RightScale with other algorithms. Simmons et al. [81] try to overcome this problem 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 elasticity 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, Kupferman et al. [65] come up with an interesting idea called *smart kill*: There is no reason to terminate a VM before the hour is over, even if the load is low, because usually partial hours are charged as full hours.

Before finishing this section, it is worth-mentioning the idea of using dynamic thresholds depending on the application conditions. Beloglazov and Buyya [44] propose a technique for dynamic consolidation of VMs based on adaptive utilization thresholds. Authors collect the CPU utilization of each VM allocated in a host. They try to determine the probability distribution of the CPU utilization for a host U_i, that is the sum of utilizations by m Vms allocated to that host. Then, it is possible to find out an interval of the CPU utilization, which will be reached with a low probability (for example $> 90\%$ and $< 20\%$), and therefore, set the corresponding thresholds. In the same line of dynamic thresholds, Lim et al. [67] described a proportional thresholding technique, based on an integral controller.

Table 1 shows a summary of the articles cited in this section.

5.2 Reinforcement Learning (Q-Learning)

Reinforcement Learning (RL) is a type of automatic decision-making approach that can be applied for auto-scaling. It online captures the performance model of a target application and its policy without any *a priori* knowledge. We will describe the problem modeling as a Markov Decision Process, and the Q-learning algorithm, which is one of the existing methods to solve RL problems.

5.2.1 Definition of the Technique

Reinforcement learning is a computational approach to understanding and automating goal-directed learning and decision-making. It focuses on learning through direct interaction between an agent and its environment (see Figure 3). The decision-maker is called the *agent*, that will learn from experience (trial-and-error method) the best *action* to execute for each *state* of the environment, always trying to maximize the returned *reward*. In terms of our problem, the auto-scaler is the agent that interacts with the scalable application

Ref	Auto-scaling Techniques	H/V Scal.	Target tier	Metric	Workloads	Experimental Platform
[54]	Rules + RL	H	BT	Response time	Synthetic. Made up of 5 sinusoidal oscillations	Custom testbed + RUBiS
[58]	Rules	Both	All tiers	Response time	Synthetic. Browsing and ordering behavior of customers.	Custom testbed (called IC Cloud) + TPC
[59]	Rules	Both	LB + BT	CPU load, response time, network link load, jitter and delay.		Only algorithm is described, no experimentation is carried out.
[52]	RightScale + MA to performance metric	H	BT	Number of active sessions	Synthetic. Different number of HTTP clients	Custom testbed. Xen + custom collaborative web application
[65]	RightScale + Time-series: LR and AR(1)	H	BT	CPU load	Synthetic. Three traffic patterns: weekly oscillation, large spike and random	Custom simulator, tuned after some real experiments.
[56]	RightScale	H	BT	CPU load	Real. World Cup 98	Real provider. Amazon EC2 + RightScale (PaaS) + a simple web application
[81]	RightScale + Strategy-tree	H	BT	CPU load	Real. World Cup 98	Real provider. Amazon EC2 + RightScale (PaaS) + a simple web application.
[70]	Rules	V	BT	Cpu, memory, bandwidth, storage	Synthetic traffic	Custom simulator + Java rule engine Drools

Table 1: Summary of references about static threshold-based rules.

environment defined in Section 2. It will decide whether to add or remove resources to the application (actions), depending on the current input workload, performance or other set of variables (state), and always trying to minimize the application response time (or other scalar reward).

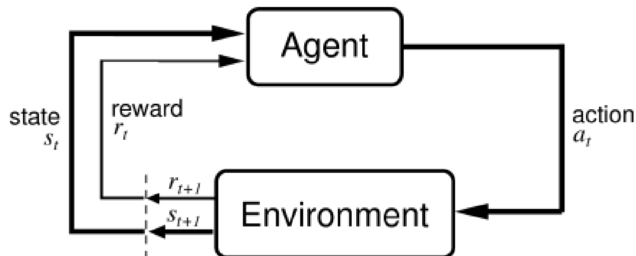

Figure 3: The agent-environment interaction in reinforcement learning[2].

More formally, at each time step t, where $t = 0, 1, 2, \ldots$ is a sequence of discrete time steps, the agent receives some representation of the environment's state $s_t \in S$, where S is the set of possible states, and on that basis selects an action, $a_t \in A(s_t)$, where $A(s_t)$ is the set of actions available in state s_t. One time step later, as a consequence of its action, the agent receives a numerical reward, r_{t+1}, and finds itself in a new state s_{t+1}. At each time step, the agent implements a mapping from states to probabilities of selecting each possible action. This mapping is called the agent's *policy* and is denoted π_t, where $\pi_t(s, a)$ is the probability that $a_t = a$ if $s_t = s$.

[2]Figure taken from `http://webdocs.cs.ualberta.ca/~sutton/book/ebook/node28.html`

RL environments can be thought of as *memoryless*, in the sense that future states can be determined only with the current state, regardless of the past history. This is called the *Markov property*, which states that the probability of a transition to a new state s_{t+1} only depends on the current state s_t and the decision maker's action a; it is conditionally independent of all previous states and actions:

$$P\left\{s_{t-1} = s', r_{t+1} = r | s_t, a_t\right\} = P\left\{s_{t-1} = s', r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0\right\} \quad (3)$$

A stochastic process that satisfies the Markov property is called a Markov Decision Process (MDP). It provides a mathematical framework for modeling decision making problems, and is typically used to formulate reinforcement learning scenarios. A MPD is represented as a 4-tuple consisting of states, actions, transitions probabilities and rewards.

- S, represents the environmental state space
- A, represents the action space
- $R : S \times A \to \mathbb{R}$ defines the reward for each state-action pair. $R(s, a)$ is the reward received after executing action a from state s.
- $T : S \times A \to P(S)$, for each state s and action a, it specifies a probability distribution over the S set. It defines the probability $P(s'|s, a)$ of a transition to state s', given current state s and execution of action a.

The core problem of MDPs (and also of reinforcement learning) is to find a policy for the decision maker or agent: a function $\pi : S \to A$ that maps every state s with the best action a, trying to maximize the cumulative future rewards. A parameter called *discount rate* γ is introduced to avoid infinite reward values in continued tasks such as the auto-scaling problem. The agent's policy should maximize the expected discounted rewards obtained in the long run:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{0}^{\infty} \gamma^k r_{t+k+1} \quad (4)$$

In order to find the optimal policy π^*, the RL algorithm should estimate an utility value for each state s. The utility is the sum of the expected discounted rewards for that state. Once the agent has learned all the estimated values, given the current state s_t, it will choose the action that leads to the state s_{t+1} with the highest utility value. The *value-state function* $V^\pi(s)$ is used to represent the estimated utility values for each state, where $V^\pi(s)$, is the expected return when starting in s and following π thereafter.

$$V^\pi(s) = E_\pi\left\{R_t | s_t = s\right\} = E_\pi\left\{\sum_{0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right\} \quad (5)$$

However, in order to determine the value-state function, we need to know the transition probabilities T for each state-action pair. In order to solve this problem, some RL algorithms focus on estimating a *action-value function* Q^{π}. We define the utility value of taking action a in state s under a policy π, denoted $Q^{\pi}(s,a)$, as the expected return starting from s, taking the action a, and thereafter following policy π:

$$Q^{\pi}(s,a) = E_{\pi}\left\{R_t | s_t = s, a_t = a\right\} = E_{\pi}\left\{\sum_{0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right\} \tag{6}$$

The relationship between the value-state function V and the action-value function Q is defined as:

$$V^{\pi}(s) = \max_{a \in A} Q^{\pi}(s,a) \tag{7}$$

As explained above, π^* will choose the action that leads to the state with the highest utility value, given that all the state-action values are known in either the $V^*(s)$ or the $Q^*(s,a)$ functions. In this case, we will focus on the action-value function:

$$\begin{aligned} Q^*(s,a) &= \max_{\pi} Q^{\pi}(s,a) \forall s \in S, \forall a \in A \\ \pi^*(s) &= \arg_{a \in A} \max Q^* \pi(s,a) \end{aligned} \tag{8}$$

Almost all reinforcement learning algorithms are based on estimating value function of states (or of state-action pairs) from experience. There are three main classes of methods: dynamic programming, Monte Carlo methods, and temporal-difference learning. Dynamic programming require the model (both transition probabilities T and rewards R) to be known, whilst the other two are model-free. Temporal-difference (TD) methods are adequate for step-by-step learning, and the most suitable for the auto-scaling scenario. Indeed, Q-learning is a one-step TD algorithm used in the literature for this type of environments. It is based on learning the action-value function Q that directly approximates the optimal Q^*, independent of the policy being followed. For this reason, it is called an *off-policy* method: the optimal policy π^* can be derived at any time, considering the action with the highest Q-function value for each state.

At each iteration step, the function $Q(s_t, a_t)$ is updated with the immediate reward R for making an action a_t, plus the best utility value $Q(s_{t+1}, a)$ for the resulting state s_{t+1}. The update formula is defined as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[R_{t+1} + \gamma \max_{a} Q(s_{t+1}, a)) - Q(s_t, a_t)] \tag{9}$$

Two parameters need to be set: the *learning rate* α and the *discount factor* γ. α defines the impact of the new data on the Q function, while γ adjusts the importance given to future rewards.

Typically, the $Q(s,a)$ values are stored in a lookup table, that maps all system states s_t to their best action a_t and can be initialized with chosen values or based on a specific, but simple, performance model. Another important task related to the Q-learning algorithm is the action selection, that must balance both exploration and exploitation. In order to

obtain a higher reward, a reinforcement learning agent will choose those previously explored actions that have been found to be effective in producing a reward. However, to discover these actions, the agent will have to execute actions that have not been previously picked. A typical strategy is called ϵ-greedy: most of the time (with probability ϵ), the action with the best reward will be executed ($\arg_a \max Q(s, a)$); and a random action will be selected with a low probability $1 - \epsilon$, in order to explore non-visited actions.

5.2.2 Review of Proposals

The basic elements of a MDP, specifically, the action set A, the state space S, and the reward function R, can be defined in a number of ways even for the same problem. Authors have proposed different definitions for the auto-scaling scenario. For example, in case of horizontal scaling, we could consider a state s defined as (w, u, p), where w is the total number of user requests observed per time period, u is the number of VMs allocated to the application, and p is the performance in terms of average response time to requests, bounded by a value P_{max} chosen from experimental observations [55]. Another approach focused on vertical scaling could consider that the state is given by the memory and CPU usage for each VM [78]. The possible set of actions A depends on the type of scaling: add, remove or maintain the number of VMs, in horizontal scaling; or increase or reduce the amount of assigned CPU and memory. Regarding the reward function, it usually takes into account both the cost of the resources (renting VMs, bandwidth, ...), and the cost derived from SLO violations [43], [55], [78].

Although RL seems a promising technique for auto-scaling, it presents several problems, that have been addressed in a number of ways [54], [55] [43], [83], [78]:

- Bad initial performance and large training time: The performance obtained during live online training may be unacceptably poor, both initially and during an unfeasibly long training period.

- Large state-space: It is often called *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, the performance worsens.

In order to improve the bad performance in early steps, a different initial function could be used. For example, Dutreilh et al. [55] 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 [78] or using parallel learning agents [43]. 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 performance in online training consists of using an alternative model (e.g. a queuing model) to control the system, whilst the RL model is trained offline on collected data.

In order to cope with larger state spaces, other nonlinear function approximators can be used, instead of lookup tables, such as neural networks, regression trees, support vector machines, wavelets and regression splines. For example, neural networks [83], [78] 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.

As we have said previously, the action selection policy used by the Q-learning algorithm has a great impact in the learning stages. ϵ-greedy is a commonly used strategy for this purpose, in which random actions are selected with a low probability $1 - \epsilon$. This allows the RL algorithm to adapt the optimal policy to relatively smooth changes in the behavior of the application, but not to sudden burst in the input workload. This is still a problem that should be addressed in order to apply RL in production systems.

Table 2 shows a summary of the articles cited in this section:

Ref	Auto-scaling Techniques	H/V Scal.	Target tier	Metric	Workloads	Experimental Platform
[43]	RL	H	BT	Number of user requests, and number of VMs	Synthetic (Poisson distribution)	Custom simulator (Matlab)
[55]	RL	H	BT	Number of user requests, number of VMs, and performance (response time)	Synthetic. With sinusoidal pattern	Custom testbed. Olio application + Custom decision agent (VirtRL)
[83]	RL(+ANN) + Queuing model	H	BT	Arrival rate, Response time, number of servers	Synthetic. Poisson distribution (open-loop), different number of users and exponentially distributed think times (closed-loop)	Custom testbed (shared datacenter). Trade3 application (a realistic simulation of an electronic trading platform)
[78]	RL(+ANN)	V	BT	CPU and memory usage	Synthetic: 3 workload mixes (browsing, shopping and ordering)	Custom testbed. Xen + 3 applications (TPC-C, TPC-W, SpecWeb)

Table 2: Summary of references about reinforcement learning.

5.3 Queuing Theory

Classical queuing theory has been extensively used to model Internet applications and traditional servers, in order to estimate performance metrics such as the queue length or the average waiting time for requests. In the current section, we will describe the main characteristics of a queuing model and how they can be applied to scalable scenarios.

5.3.1 Definition 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 Figure 4. Client 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.

Kendall's notation is the standard system used to describe and classify queuing models. A queue is described in shorthand notation by $A/B/C/K/N/D$ or the more concise $A/B/C$.

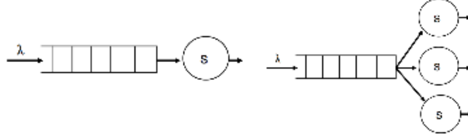

Figure 4: A simple queuing model with one server (left) and multiple servers (right).

The elements K, N and D are optional; if not present, it is assumed $K = \infty$, $N = \infty$ and $D =$FIFO. This is the description for each element in 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 number is at this maximum, further arrivals are turned away. If this number is omitted, the capacity is assumed to be unlimited, or infinite.
- N, *Calling population*: The size of the population from which the customers come. If this number is omitted, the population is assumed to be unlimited, or infinite. When the requests come from an infinite population of customers, we have an *open* queuing model, whereas a *closed* queuing model is based on a finite population of customers.
- D, *Service discipline* or *priority order*: The service discipline or priority order that 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 in. There are alternatives such as LIFO/LCFS (Last in First Out/Last Come First Served) and PS (Processor Sharing).

The most typical values for both inter-arrival A and service B times, are M, D and G. M value 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. Besides, the inter-arrival or the service time will follow an exponential distribution. In case of D value, those times are deterministic or constant all the time. Another commonly used value is G, that corresponds to a general distribution with a known mean and variance.

The cloud application scenario described in Section 2 can be formulated using a simple queuing model, considering a single queue for the load balancer, that distributes the requests among n VMs (see Figure 4). Usually, several queues are combined in the same model (a *queuing network*). More complex, realistic system can be formulated as a queuing network, such as multi-tier applications. For example, each tier can be modeled as a queue with one or n servers (see Figure 5).

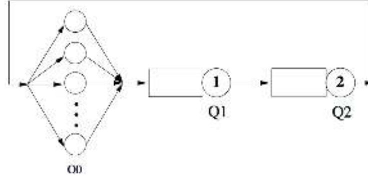

Figure 5: A queuing network modeling a 3-tier application, with one queue per tier[3].

Queuing theory is intended for systems with a stationary nature, characterized by constant arrival and service rates. In case of scenarios with changing conditions, such as our target, scalable application, the parameters of the queuing model have to be periodically recalculated. Based on a stationary queuing model, several useful performance metrics can be estimated, including the arrival rate λ, the inter-arrival time, the average number of requests in the queue or in the whole system, the average time waiting in the queue, and the service time. There are two main approaches to obtain those metrics: analytic methods (usually valid for simple models), and simulation (that can be applied to more complex scenarios).

Analytic methods are only available for relatively simple queuing models, with well-defined distributions for arrival and service process (e.g. standard statistical distributions such as Poisson or Normal). A typical formula in this context is the *Little's Law*. It states that the average number of customers (or requests) $E[N]$ in the system is equal to the average customer arrival rate λ multiplied by the average duration of each customer $E[T]$. The formula is as follows: $E[N] = \lambda \times E[T]$. A similar definition for the Little's Law says that the average queue length can be calculated as the product of the mean arrival rate and the average waiting time in queue.

There is a limited set of queuing models that can be solved using analytic methods, including M/M/1 and the G/G/1 processes. The easiest case are Poisson-based models, where both the inter-arrival times and the service times follow the exponential distribution. For example, 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. Another simple queuing model is G/G/1, in which the system's inter-arrival and service times are governed by a general distribution with known means and variances. The behavior of a G/G/1 system can be captured using the following formula: $\lambda \geq \left[s + \frac{\sigma_a^2 + \sigma_b^2}{2(R-s)} \right]^{-1}$, where R is the mean response time, s is the average service time for a request, σ_a^2 and σ_b^2 are the variance of inter-arrival time and the variance of service time, respectively. All those values can be monitored online.

The clear drawback of analytic methods are the imposed assumptions that are not usually valid for real scenarios. Furthermore, as the complexity of the system grows, the analytic formulas become less and less tractable. The alternative to analytic methods are discrete-event simulation (DES) techniques. They are free from assumptions about the particular type of the arrival process (Poisson or not), as well as about the service time (exponential or not), and can be applied to complex systems composed of several queues. A DES model

[3]Figure taken from `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4273121&tag=1`

is a computer model that mimics the dynamic behavior of a real process as it evolves with time in order to visualize and quantitatively analyze its performance.

5.3.2 Review of Proposals

In the literature, both simple queuing models and queuing networks have been used to model applications. For example, general Internet applications have been usually formulated using simple queuing models, and even a cloud infrastructure can be modeled as a G/G/N queue, in which the number of servers N is variable [41]. The model is used to estimate different parameters, for example, the necessary resources required for a given input workload λ, or the mean response time for requests. Then, this information is included in different techniques, such as a predictive controller [41] or to solve an optimization problem (e.g. distributing servers among different applications, while maximizing the revenue [85]).

Due to the limitation of models with a single queue, we will focus on the use of queuing networks to formulate a single application tier as n parallel servers [83], [85], or a whole multi-tier application [84], [89]. The first approach is adopted by Tesauro et al. [83], who considered both open and closed-loop models with the following specifications:

- *Open Queuing Network Model*: Given an application with an overall demand λ distributed on a round-robin basis among n servers, the system can be modeled as n independent and identical parallel open networks each with one server and a demand level of λ/n. Authors used a parallel M/M/1 queuing formulation to model the mean response time characteristics of an application, along with exponential smoothing for estimates of μ.

- *Closed-Loop Queuing Network Model*: In this case, the application has a finite number of customers M. Thus, it can be model as n independent parallel closed networks each with one server and M/n customers. Mean Value Analysis (MVA) formulation [71] is used to model the mean response time characteristics of the application, defined by two parameters: the average think time Z and the average service time at the server, $1/\mu$.

Similarly, Villela et al. [85] focus on the application tier of e-commerce systems, and model each server as a M/GI/1/PS queue. They characterized the arrival process of requests to an application, using a real trace of an e-commerce system, and found that for their horizon time of interest, this arrival process is adequately described by a Poisson process (M or Markovian). The arrival rate of request is uniformly split among the servers allocated.

Multi-tier applications can also be addressed using queuing networks, either considering a queue per server [84], or just a queue per tier [89]. The first approach is adopted by Urgaonkar et al. [84], who use a network of G/G/1 queues (one per server). Based on this model, they combine the formula for λ (previously defined) in a G/G/1 system, and the Little's Law in order to determine the peak workload and the number of servers needed on each tier to satisfy the demand. This peak workload is predicted using histograms, and corrected using reactive methods. The clear drawback is that provisioning for peak load

drives to high under-utilization of resources. The other mentioned approach is to model each tier with a single queue. Zhang et al. [89] 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 Mean-Value Analysis (MVA) [71].

The information required for a queuing model, such as the input workload (number of requests, transactions) or service time can be obtained by online monitoring [84] or estimated using different methods. For example, Zhang et al. [89] used a regression-based approximation in order to estimate the CPU demand on client transactions.

Table 3 shows a summary of the articles cited in this section:

Ref	Auto-scaling Techniques	H/V Scal.	Target tier	Metric	Workloads	Experimental Platform
[84]	QT + Histogram + Thresholds	H	All	PEAK workload	Synthetic and Real (World Cup 98)	Custom testbed. Xen + 2 applications (RUBiS and RUBBOS)
[85]	QT	H	BT	Arrival rate	Real. E-commerce website (2001) + Synthetic traces	Custom simulator (Monte-Carlo)
[89]	QT + Regression (Predict CPU load)	-	All	Number and type of transactions (requests)	Synthetic (browsing, ordering and shopping)	Custom simulator, based on C++Sim. + Data collected from TPC-W
[83]	RL(+ANN) + QT	H	BT	Arrival rate, Response time, number of servers	Synthetic. Poisson distribution (open-loop), different number of users and exponentially distributed think times (closed-loop)	Custom testbed (shared datacenter). Trade3 application (a realistic simulation of an electronic trading platform)

Table 3: Summary of references about reinforcement learning.

5.4 Control Theory

Control theory has been applied to automate the management of web server systems, storage systems, data centers/server clusters, and other systems, and it also shows interesting results on cloud computing platforms. Control systems are mainly reactive, but there are also some proactive approximations such as Model Predictive Control, or even combining a control system with a predictive model.

5.4.1 Definition of the Technique

There are three types of control systems: open loop, feedback and feed-forward. *Open-loop* controllers, also referred 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 has achieved the desired goal. In contrast, *feedback* controllers observe the output of the system, and are able to correct any deviations from the desired value. *Feed-forward* controllers try to anticipate to errors in the output. They predict the behavior of the system, based on a model, and react before the error actually occurs. The prediction may fail, and for this reason, feedback and feed-forward controllers are usually combined.

In the context of scalable applications (see Section 2), both feedback and feed-forward types seem the most suitable approaches. Indeed, authors have used feedback controllers to address the auto-scaling task. The main objective of the controller is to maintain the output (e.g. performance) of the target system (the application) to the desired level (e.g. response time below 2 seconds), by adjusting the control input (e.g. resource allocation, number of VM).

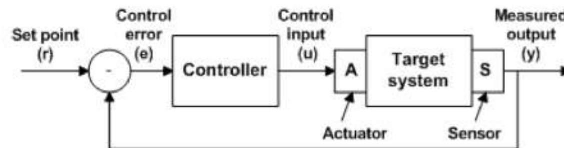

Figure 6: Block diagram of a feedback control system[4].

The basic structure of a feedback controller is shown in Figure 6. These controllers can be divided into four categories [76]:

- *Fixed gain controllers*: They are very popular due to their simplicity. However, after selecting the tuning parameters, they remain fixed during the time the controller is in operation. An example is the Proportional Integral Derivative (PID) controller. An Integral controller can be represented as:

$$u_{k+1} = u_k + K_i(y_{ref} - y_k) \tag{10}$$

  where u_{k+1} is the new actuator value (e.g. new number of VM), u_k is the current actuator value (e.g. current number of VM), K_i is the integral gain parameter, y_k is the current sensor measurement (e.g. CPU load); and y_{ref} is the target value for the variable.

  And this is the formula for a Proportional-Integral (PI) controller:

$$u_{k+1} = u_k + (K_p + K_I)E_k - K_p E_{k-1} \tag{11}$$

  It includes two new parameters: E_k, that refers to an error (reference minus measured actuator value), and K_p, which is a proportional term that determines the ratio of output response to the error in the previous cycle. K_I is an integral term that determines the throttling for accumulated errors in previous cycles.

- *Adaptive control*: Adaptive control addresses some of the limitations of fixed gain controllers, by adjusting the controller tuning parameters online. On each sampling step, a dynamic model of the system is constructed. Examples of adaptive controllers

[4]Figure taken from `http://www.ict.swin.edu.au/personal/tpatikirikorala/docs/Tharindu_Cloudworkshop2010.pdf`

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 online model estimation process may fail to capture the dynamics of the system.

- *Reconfiguring control*: In adaptive control, the parameters are derived online, but the controller remains the same. The reconfiguring controllers remove this limitation by allowing the controller to be changed at run time depending on the conditions. Some examples are Model-Switching and Tuning adaptive control. They can be suitable to handle both predictable and unpredictable bursts in workload.

- *Model predictive control (MPC)*: In contrast to purely feedback controllers, MPC uses a proactive approach and predicts future behavior of the system. This may be an interesting feature in cloud platforms.

The number of input and output variables of a feedback controller can be just one, yielding a *Single-input single-output (SISO)* controller; or more than one, yielding a *Multiple-input multiple-output (MIMO)* controller.

Before designing a control system, a formal relationship between the control input and the output has to be modeled, that is called the *transfer function*. Basic controllers usually consider linear approaches for this problem (e.g. Linear Time Invariant algorithm), but there are alternatives that are able to capture this input-output mapping more accurately. In the literature, several approaches have been followed to create a performance model of the system, such as ARMA model [74] (see Section 5.5), smoothing splines [45], fuzzy logic [88], [86], [66] and Kalman filters [63].

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

- *Kalman filter*: It is a well-established technique in control systems for noise filtering and state prediction. A Kalman filter is a recursive estimator for a new state s_{t+1}, based on both the current measurement z_t and the estimation of the previous state s_t.

- *Fuzzy model*: *Fuzzy logic* is a tool to deal with uncertain, imprecise, or qualitative decision-making problems. Unlike in Boolean logic, where an element either belongs or does not belong to a set A, the membership of x in A has a degree value in a continuous interval between 0 and 1.

5.4.2 Review of Proposals

The first type of controllers are fixed gain type, that include the PID controller. Different variants of this one have been used in the literature. An Integral controller is able to adjust the number of VMs based on average CPU usage [67], [68]. The PI controller has been applied to control the resources required by batch jobs, based on their execution progress [75]. K_p and K_I parameters can be set manually, based on trial-and-error [67] or using a

model for the application. Park and Humphrey [75] construct a model for the progress of a job with respect to provisioned resources.

Many papers discuss adaptive control techniques [74], [77], [41] [45]. Ali-Eldin et al. [41] propose combining two proactive, adaptive controllers for scaling down with dynamic gain parameters based on input workload, and a reactive approach for scaling up. Padala et al. [74] 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. A different approach to create a performance model is based on smoothing splines, that can be trained to map the workload and number of servers to the application performance. Bodík et al. [45] combine this performance model with a gain-scheduling (adaptive controller), with α and β parameters, that controlled the scaling rate. Kalyvianaki et al. [63] designed different SISO and MIMO controllers to determine the CPU allocation of VMs, relying on Kalman filters.

Finally, fuzzy models are also used as a complement to control systems to map 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 the 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 online monitored information [88], [86]. Xu et al. [88] applied an adaptive fuzzy controller to the business-logic tier of an application, and estimated the required CPU load for the input workload. A similar approach is followed by [86], 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 one period of time. The same authors [87] have also used a Fuzzy Model Predictive Controller, that again combines control theory and fuzzy rules.

A further improvement is the *neural fuzzy controller*, which utilizes 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 [66] 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 online learning (a reconfiguring controller type).

Table 4 shows a summary of the articles cited in this section:

Ref	Auto-scaling Techniques	H/V Scal.	Target tier	Metric	Workloads	Experimental Platform
[76]	Control Theory: Feedback controllers classification					
[75]	Control Theory: PI controller	V	Batch jobs	CPU usage and job progress	Batch jobs	Custom testbed. HyperV + 5 applications (ADCIRC, OpenLB, WRF, BLAST and Montage) + Sensor library (to get progress of the application)
[67]	Control Theory: PI controller (Proportional thresholding) + Exponential Smoothing for performance variable	H	BT	Cpu utilization	Synthetic. Different number of threads.	Custom testbed. Xen + simple web service
[68]	Control Theory: PI controller (Proportional thresholding)	H	ST	Cpu utilization	Synthetic.	Custom testbed. Xen + Modified CloudStone (using Hadoop Distributed File System) + Hyperic as monitor
[74]	Control Theory: MIMO adaptive controller + ARMA (performance model)	V	BT	Cpu usage and disk I/O	Synthetic and realistic (generated with MediSyn)	Custom testbed. Xen + 2 applications (RUBiS and TPC-W)
[41]	Control Theory: Adaptive controllers + QT	H	BT	Number of requests, service rate	Real. World Cup 98.	Custom simulator in Python
[45]	Control Theory: Gain-scheduler (adaptive) + Smoothing splines (performance model) + Linear Regression	H	BT	Number of requests	Synthetic (Faban generator)	Real provider. Amazon EC2 + CloudStone benchmark
[88]	Controller Theory: fuzzy controller)	V	BT	Number of requests, CPU load	Real (World Cup 98) and Synthetic (Httperf)	Custom testbed. VMware ESX Server + Java Pet Store.
[86]	Fuzzy model	V	DT	Number of queries, CPU load, disk I/O bandwidth	Synthetic and realistic (based on World Cup 98)	Custom testbed. Xen + 2 applications (RUBiS and TPC-H)
[66]	Control Theory: Reconfiguring control + Fuzzy model (+ ANN)	V	All	Number of requests, resource usage	Synthetic. (Pareto distribution)	Simulation
[63]	Control Theory: Adaptive SISO and MIMO controllers + Kalman filter	V	All	CPU usage	Synthetic (Browsing and bidding mix)	Custom testbed. Xen + RUBiS application

Table 4: Summary of references about control theory techniques.

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 could be used to find repeating patterns in the input workload or to try to forecast future values.

5.5.1 Definition of the Technique

The general scenario has already been described in Section 2. In this case, a certain performance metric, such as average CPU load or the input workload, will be periodically sampled at fixed intervals (e.g. each minute). The result will be a time-series X containing a sequence of the last w observations:

$$X = x_t, x_{t-1}, x_{t-2}, ..., x_{t-w+1} \tag{12}$$

The auto-scaling problem can be divided into two parts: prediction of a future value (using time-series analysis) and decision making. Time-series analysis is only applied in the first part, that involves making an estimation of the future workload or resource usage. Based on this predicted value, the second step consists of deciding the suitable scaling action to take. Several approaches can be used in decision making such as a set of predefined rules [64] or solving an optimization problem for the resource allocation [79]. In the current section we will focus on the first step that uses time-series based techniques.

As stated before, there are two main goals of time-series analysis: (1) forecasting future values of the time-series, based on the last observations, (2) identifying the pattern (if present) that follows the time-series, and then extrapolate it to predict future values. In both cases, the required information is a list of the last w observations of the time-series, that we will denote as *input window* or *history window*.

Forecasting techniques can be applied either to resource usage or workload prediction. Based on the last w consecutive observations ($x_t, x_{t-1}, x_{t-2}, ..., x_{t-w+1}$), a future value y_{t+r} is predicted, which is r intervals ahead of the input window. Some of the techniques used for this purpose in the literature are Moving Average, Auto-regression, ARMA (combining both), exponential smoothing and different approaches based on machine learning.

- *Averaging methods:* They can be used to smooth a time-series in order to remove noise or to make predictions. The forecast value y_{t+1} is calculated as the weighted average of the last w consecutive values. The general formula is as follows: $y_{t+r} = a_1 x_t + a_2 x_{t-1}, ...,$ where $a_1, a_2, ..., a_w$ are a set of positive weighting factors that must sum 1. Depending on the way of determining those weights, several methods are defined:
  - Moving average MA(q): Simple MA is the arithmetic mean of the last q or w values, i.e., it assigns equal weights $\frac{1}{w}$ to all observations.
  - Weighted moving average WMA(q): Different weights are assigned 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: It assigns *exponentially* decreasing weights over time. A new parameter is introduced, a smoothing factor α that weakens the influence of past data. The predictor formula for single exponential smoothing is:

$$
\begin{aligned}
y_{t+1} =& \alpha x_t + (1-\alpha) y_t \\
=& \alpha x_t + (1-\alpha)[\alpha x_{t-1} + (1-\alpha) y_{t-1}] \\
=& \alpha x_t + \alpha(1-\alpha) x_{t-1} + (1-\alpha)^2 [\alpha x_{t-2} + (1-\alpha) y_{t-2}] \\
& \vdots \\
=& \alpha x_t + \alpha(1-\alpha) x_{t-1} + (1-\alpha)^2 x_{t-2} + \ldots + (1-\alpha)^{w-1} x_{t-w+1}
\end{aligned} \tag{13}
$$

where y_{t+1} represents prediction value for the period $t+1$, x_t is the value at time t, and y_t is the forecast made for period t. *Simple* exponential smoothing is suitable for time-series that have no significant trend changes, whereas *double* smoothing can be applied to time-series with an existing linear trend. *Triple* exponential smoothing can be used for time-series with trend and seasonality. Double and triple exponential smoothing are derived by applying exponential smoothing to the already smoothed data.

- *Auto-regression* of order p, AR(p): In this case, the weighting factors $a_1, a_2, ...$ are determined by calculating auto-correlation coefficients and solving linear equations: $\sum_{i=1}^{w} a_i R(i-j) = -R(j)$, for $1 \leq j \leq w$, where R is the auto-correlation coefficients of the time series, and p or w is the length of the sample.

- *Auto-regressive Moving Average*, ARMA(p,q): It combines both auto-regression (of order p) and moving average (of order q). The predicted output y_{t+1} is based on the previous outputs $y_{t-1}, y_{t-2}, ...$ and the inputs x_t, x_{t-1}, x_{t-1}. The general formula is:

$$
y_{t+1} = a_1 y_{t-1} + a_2 y_{t-2} + \ldots + b_0 x_t + b_1 x_{t-1} + \ldots \tag{14}
$$

- *Machine Learning-based techniques*:
  - Regression is a statistical method used to determine the polynomial function that is 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 best. When the number of input variables is more than one, it is referred as the Multiple Linear Regression.
  - Neural networks: They consist of an interconnected group of artificial neurons, disposed on 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 this particular problem, the input layer will contain 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 described, another goal in time-series analysis is identifying the pattern that the series follows, and then extrapolate it to predict 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. As explained in Section 3, input workloads of internet servers and different cloud applications may show different periodic components. The trend identifies the overall slope of load to the application, whereas seasonality and cyclical determines the peeks at specific point 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, signal processing techniques and auto-correlation.

- *Pattern matching*: Pattern matching consists on searching for similar patterns on the history time-series, that are similar to the present pattern. It is very close to the *string matching problem*, that has already been explored and several efficient algorithms are available (e.g. Knuth-Morris-Prat [53]).

- *Signal processing techniques*: Fast Fourier Transform (FFT) is a technique that decomposes the 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 fixed threshold (e.g. 0.9) after s shifts, a repeating pattern is declared, with duration s steps.

A basic technique for time-series representation is an *histogram*. It involves dividing the time-series into several equal-width bins, and representing the frequency for each bins. It has been used in the literature to represent the resource usage pattern or distribution, and then predict future values (see next Section).

5.5.2 Review of Proposals

In the literature, time-series techniques have been applied mostly to workload or resource usage prediction. A simple moving average could be used for this purpose, but with poor results [57]. For this reason, authors have applied this method only to remove noise from the time-series [75], [67], or just to have a comparison yardstick. For example, Huang et al. [60] 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). Exponential smoothing clearly obtained better results, because it takes into account both the current data and the history records for the prediction. Mi et al. [72] also used a quadratic exponential smoothing against real workload traces (World Cup 98 and

ClarkNet), and showed good accurate results, with a small amount of error (a mean relative error of 0.064 for the best case).

The auto-regression method has been largely used ([65], [50], [57], [49], [64]). Kupferman et al. [65] applied auto-regression of order 1 to predict the request rate (requests per second) and found that its performance depends largely on several user-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 local versus global trends, while the size of the adaptation window determines how far into the future the model extends.

Combining moving average (MA) and auto-regression (AR), the resulting algorithm is called auto-regressive moving average method (ARMA). Roy et al. [79] use a second order ARMA 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 cost of SLO violations, cost of leasing resources and reconfiguration cost.

The history window values can also be the input for a neural network [62] or a multiple linear regression equation [65], [45], [62]. The accuracy of both methods depends on the input window size. Indeed, [62] obtained better results when using more than one past value for prediction. Kupferman et al. [65] further investigated the topic and found that it is necessary to balance the size of each sample in the window, to avoid overreaction, 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 point is the prediction interval r that should be considered. Islam et al. [62] propose using a 12-minute interval, because the setup time of VM instances in the cloud is typically around 5-15 min.

Time-series forecasting can be combined with reactive techniques. Iqbal et al. [61] 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 number of application-tier and database-tier instances using polynomial regression (of degree two).

Apart from those techniques that deal with time-series forecasting, a number of authors have focused on identifying repeated patterns in the input workload [47], [48], [57], [80]. The most complete comparison of this class of techniques is done by Gong et al. [57]. 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. [47], [48], 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 algorithm, and the time required to explore the past history trace.

The last technique considered in this section is the histogram, that has 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 highest frequency [57].

Table 5 shows a summary of the articles cited in this section.

Ref	Auto-scaling Techniques	H/V Scal.	Target tier	Metric	Workloads	Experimental Platform
[47]	Time series: Pattern matching	H	BT	Total number of CPUs in 100-seconds interval	Real. A cloud application (Animoto), and from Grid systems (LCG, NorduGrid, SHARCNET)	Analytical models
[48]	Time series: Pattern matching	H	BT	Total number of CPUs in 100-seconds interval	Real cloud workloads: from Animato and 7 IBM cloud applications	Analytical models
[57]	Time series: FFT and Discrete Markov Chains. Compared with auto-regression, auto-correlation, histogram, max and min.	V	BT	CPU load, memory, I/O and network	Real. World Cup 98 and ClarkNet. Also Synthetic trace	Custom testbed. Xen + RUBiS + part of Google Cluster Data trace for CPU usage.
[80]	Time series: FFT	V (CPU & Memory)	BT	CPU load, memory usage	Synthetic. RUBiS generator	Custom tesbed. Xen + RUBiS
[79]	Time series: second order ARMA	H	All tiers	Number of users in the system	Real. World Cup 98	No experimentation on systems
[60]	Time series: Double Exponential Smoothing	Only resource pred.		CPU load, memory usage	Synthetic traffic generated with simulator	CloudSim simulator
[72]	Time series: Brown's Quadratic exponential Smoothing.	Only resource pred.		Number of requests per VM	Real. World Cup 98 and ClarkNet. Synthetic. Poisson distribution)	Custom testbed. TPC-W. 3 VM roles: Browsing, Shopping and Ordering
[50]	Time series: AR	H	BT	CPU load	Real. From Windows Live Messenger (login rate, number of active connections)	Custom testbed, with real traces
[62]	Time-series: ML – Neural Network and (Multiple) LR + Sliding window	H	BT	CPU load (aggregated value for all VMs)	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.
[61]	Threshold-based rules (scale up) + Time series, polynomial regression (scale down)	H	BT, ST	CPU load (scale up) / Response time (scale down)	Synthetic. Httperf	Custom testbed. Eucalyptus + RUBiS
[46]	Time series: Histogram			CPU load (and other resources)		Nodes with similar resource usage distribution are grouped in the same resource bundle.
[49]	Time series: AR(1) and Histogram + Queuing theory	H	BT	Request rate and service demand	Synthetic (Poisson distribution) and Real (World Cup 98).	Custom simulator + algorithms in Matlab

Table 5: Summary of references about time-series techniques.

6 Discussion and Open Research Lines

The current review has described different reactive and proactive auto-scaling methods. Threshold-based rules follow a simple reactive approach. They are the most popular auto-scaling technique, even in commercial systems, probably due to their apparent simplicity. However, we have seen that setting the suitable thresholds is a very tricky task, and may lead to instability in the system. Besides, static thresholds become invalid if the application

behavior changes. Little research has been done in the use of dynamic threshold, including the proportional thresholding proposed Lim et al. [67] or the adaptive thresholds introduced by Beloglazov and Buyya [44], and it could be interesting to improve this technique.

The main drawback of reactive techniques is that they not anticipate to unexpected changes in the workload, and therefore, resources cannot be provisioned in advance. Furthermore, it is important to take into account that adding a new VM in real cloud providers might take up to 15 minutes, and the effect of a scaling-up action might arrive too late. Therefore, research on auto-scaling techniques should focus on proactive approaches. Time-series analysis already includes a number of methods to predict future values of a metric, based on its past history. However, the prediction accuracy highly depends on the history window size (the number of past values considered) and the adaptation window (the prediction interval).

Apart from threshold-based rules and time-series analysis, three other categories have been defined that can be used with a reactive or proactive approach: reinforcement learning, control theory and queuing theory. Reinforcement learning is able to learn an auto-scaling policy from experience, without any *a priori* knowledge, but in addition to the long time required during the learning step, this technique adapts only to slowly changing conditions. Therefore, it cannot be applied to real applications that usually suffer from sudden traffic bursts. Queuing theory models impose hard assumptions that may not be valid for real, complex systems. Besides, as they are intended for stationary scenarios, the models need to be recalculated when the conditions of the application change. Indeed, the input workload rate is varying constantly. Finally, controllers are able to maintain the output (performance) of the application at the desired level, depending on the input (e.g. workload). However, setting the gain parameters can be a difficult task, and as in the case of threshold-based rules, it may cause oscillations in the assigned resources (e.g. the number of VMs).

Another, more general issue regarding these techniques is the lack of a formal evaluation methodology, including a scoring metric for comparison and a widely accepted evaluation scenario. Due to the heterogeneity of existing auto-scaling methods, it would be interesting to define a scoring metric to objectively determine whether one algorithm performs better than another. Kupferman et al. [65] proposed a scoring algorithm for comparing heterogeneous auto-scaling techniques, that considers availability and cost. However, the cost model considered for this scoring metric is quite limited, as it does not take into account the VM types, bandwidth used and other elements that are also charged to the user. Besides, several parameters have to be set based on experimentation.

Additionally, a common evaluation scenario (or scenarios) should be defined to efficiently compare different auto-scaling algorithms. Developing a cloud simulator would be an indispensable tool for this purpose, avoiding the burden of setting a whole virtualized cluster or the leasing costs of the real cloud provider.

7 Conclusions and Future Work

The present study has focused on the problem of auto-scaling applications in cloud environments. Cloud computing is a widely used technology, characterized by offering resources in an elastic manner. Users can acquire and release resources on demand, and pay only for the required resources (this is often called a pay-as-you-go scheme). The scaling task can be done manually, but to really take advantage of these facilities, an automatic scaling system is needed. This element should be able to adapt the amount of required resources (typically, VMs) to the input workload, always trying to avoid under-utilization in order to minimize the cost, but maintaining the service level objective (for example, a pre-defined response time).

We have proposed a clear definition of the auto-scaling problem, and explained the different experimental platforms used to test auto-scaling algorithms, together with the different input workloads and evaluation benchmarks available. Then, we have focused on the different auto-scaling methodologies that have appeared throughout the literature. Those techniques come from a variety of many knowledge areas and present different performance characteristics. A classification has been proposed, including five main categories: threshold-based rules, reinforcement learning, queuing theory, control theory and time-series analysis. Each technique has been described separately, including a review of some relevant references.

The general conclusion extracted from this study is the need to develop an efficient auto-scaling approach, able to cope with the varying conditions of applications derived from unanticipated changes in traffic. To continue this work, we propose using a predictive auto-scaling technique based on time-series forecasting algorithms. We have seen that the accuracy of these algorithms depend on different parameters such as the history window size and the adaptation window. Optimization techniques could be used to adjust this values, in order to tune the parameters for the given scenario.

We have also pointed out the lack of a formal testing and comparison framework for auto-scaling in the cloud. Each proposal found in the literature has been applied to different scenarios, from batch jobs to e-commerce applications, either focusing on the business-logic tier or considering the 3-tier architecture as a whole. Besides, authors have tested their proposal under very different conditions, considering a simulator, custom testbed or real provider as the testing platform; real or synthetic workloads; and also different metrics to evaluate the performance of the algorithms. Given such heterogeneity in the testing methodology, we cannot give an answer to the question of what is the best auto-scaling technique. We plan to propose a testing environment, including a cloud simulator and workloads based on different patterns. Currently, we are developing a simulator based on CloudSim [7], that will be the common workbench to compare the different auto-scaling techniques and the new ones that will for sure appear in the future.

References

[1] RUBiS: Rice University Bidding System. http://rubis.ow2.org/, 2009. [Online; accessed 13-September-2012].

[2] Cloud Computing Powers Groupon: The Fastest Growing Company Ever. http://cloud-computing.learningtree.com/2011/03/24/cloud-computing-powers-fastest-growing-company-ever/, 2011. [Online; accessed 13-September-2012].

[3] Amazon Elastic Compute Cloud (Amazon EC2). http://http://aws.amazon.com/ec2/, 2012. [Online; accessed 13-September-2012].

[4] AWS Elastic Beanstalk (beta). Easy to begin, Impossible to outgrow. http://aws.amazon.com/elasticbeanstalk/, 2012. [Online; accessed 13-September-2012].

[5] Customer Success. Powered by the AWS Cloud. http://aws.amazon.com/solutions/case-studies/, 2012. [Online; accessed 13-September-2012].

[6] ClarkNet HTTP Trace (From the Internet Traffic Archive). http://ita.ee.lbl.gov/html/contrib/ClarkNet-HTTP.html, 2012. [Online; accessed 13-September-2012].

[7] CloudSim: A Framework for Modeling and Simulation of Cloud Computing Infrastructures and Services. http://www.cloudbus.org/cloudsim/, 2012. [Online; accessed 18-September-2012].

[8] CloudStone Project by Rad Lab Group. http://radlab.cs.berkeley.edu/wiki/Projects/Cloudstone/, 2012. [Online; accessed 13-September-2012].

[9] Eucalyptus Cloud. http://www.eucalyptus.com/, 2012. [Online; accessed 18-September-2012].

[10] Greencloud - The green cloud simulator. http://greencloud.gforge.uni.lu/, 2012. [Online; accessed 18-September-2012].

[11] force.com (salesforce). http://www.force.com/, 2012. [Online; accessed 13-September-2012].

[12] Google App Engine. http://cloud.google.com/products/, 2012. [Online; accessed 13-September-2012].

[13] Google Apps for Business. http://www.google.com/intl/es/enterprise/apps/business/products.html, 2012. [Online; accessed 13-September-2012].

[14] Google Cluster Data. Traces of Google workloads. /, 2012. [Online; accessed 13-September-2012].

[15] Google Cluster Data. Trace Version 1. 7 hours of usage data. http://code.google.com/p/googleclusterdata/wiki/TraceVersion1/, 2012. [Online; accessed 13-September-2012].

[16] Google Cluster Data. Trace Version 2. Second format of cluster-usage traces. http://code.google.com/p/googleclusterdata/wiki/TraceVersion2/, 2012. [Online; accessed 13-September-2012].

[17] Google Compute Engine. http://cloud.google.com/products/compute-engine.html/, 2012. [Online; accessed 13-September-2012].

[18] Greencloud - The green cloud simulator. http://greencloud.gforge.uni.lu/, 2012. [Online; accessed 18-September-2012].

[19] Groupon. http://www.groupon.es/deals/bilbao/, 2012. [Online; accessed 13-September-2012].

[20] Heroku. Cloud application platform. http://www.heroku.com/, 2012. [Online; accessed 13-September-2012].

[21] The httperf HTTP load generator. http://code.google.com/p/httperf/, 2012. [Online; accessed 18-September-2012].

[22] Apache JMeter. http://jmeter.apache.org/, 2012. [Online; accessed 18-September-2012].

[23] Kernel Based Virtual Machine. http://www.linux-kvm.org/, 2012. [Online; accessed 18-September-2012].

[24] Microsoft Office 365. http://www.microsoft.com/en-us/office365/online-software.aspx, 2012. [Online; accessed 13-September-2012].

[25] OpenStack Cloud Software. Open source software for building private and public clouds. http://www.openstack.org/, 2012. [Online; accessed 18-September-2012].

[26] Rackspace. The open cloud company. http://www.rackspace.com/, 2012. [Online; accessed 13-September-2012].

[27] Rain Workload Toolkit. https://github.com/yungsters/rain-workload-toolkit/wiki, 2012. [Online; accessed 13-September-2012].

[28] RightScale Cloud Management). http://www.rightscale.com/, 2012. [Online; accessed 13-September-2012].

[29] RightScale. Set up Autoscaling using Voting Tags. http://support.rightscale.com/03-Tutorials/02-AWS/02-Website_Edition/Set_up_Autoscaling_using_Voting_Tags, 2012. [Online; accessed 13-September-2012].

[30] RUBBoS: Bulletin Board Benchmark. http://jmob.ow2.org/rubbos.html/, 2012. [Online; accessed 18-September-2012].

[31] Salesforce.com. http://www.salesforce.com/, 2012. [Online; accessed 13-September-2012].

[32] SPEC forms cloud benchmarking group. http://www.spec.org/osgcloud/press/cloudannouncement20120613.html, 2012. [Online; accessed 18-September-2012].

[33] TPC. Transaction Processing Performance Council. http://www.tpc.org/default.asp, 2012. [Online; accessed 13-September-2012].

[34] TPC-C. http://www.tpc.org/tpcc/default.asp/, 2012. [Online; accessed 18-September-2012].

[35] TPC-W. http://www.tpc.org/tpcw/default.asp, 2012. [Online; accessed 13-September-2012].

[36] VMware vCloud Director. Deliver Complete Virtual Datacenters for Consumption in Minutes. http://www.eucalyptus.com/, 2012. [Online; accessed 18-September-2012].

[37] VMware vSphere ESX and ESXi Info Center. http://www.vmware.com/es/products/datacenter-virtualization/vsphere/esxi-and-esx/overview.html, 2012. [Online; accessed 18-September-2012].

[38] Microsoft Windows Azure. https://www.windowsazure.com/en-us/, 2012. [Online; accessed 13-September-2012].

[39] World Cup 98 Trace (From the Internet Traffic Archive). http://ita.ee.lbl.gov/html/contrib/WorldCup.html, 2012. [Online; accessed 13-September-2012].

[40] Xen hypervisor. http://http://www.xen.org/, 2012. [Online; accessed 18-September-2012].

[41] A Ali-Eldin, J Tordsson, and E Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 204–212. IEEE, 2012.

[42] A Andrzejak, D Kondo, and S Yi. Decision model for cloud computing under sla constraints. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, pages 257–266. IEEE, 2010.

[43] E Barrett, E Howley, and J Duggan. Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. *Concurrency and Computation: Practice and Experience*, 2012.

[44] A Beloglazov and R Buyya. Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers. In *Proceedings of the 8th International Workshop on Middleware for Grids, Clouds and e-Science*, page 4. ACM, 2010.

[45] Peter Bodík, Rean Griffith, Charles Sutton, Armando Fox, Michael Jordan, and David Patterson. Statistical machine learning makes automatic control practical for internet datacenters. page 12, June 2009. URL `http://dl.acm.org/citation.cfm?id=1855533.1855545`.

[46] Michael Cardosa and Abhishek Chandra. Resource Bundles: Using Aggregation for Statistical Large-Scale Resource Discovery and Management. *IEEE Transactions on Parallel and Distributed Systems*, 21(8):1089–1102, August 2010. ISSN 1045-9219. doi: 10.1109/TPDS.2009.143. URL `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5226621`.

[47] Eddy Caron, Frédéric Desprez, and Adrian Muresan. Forecasting for Cloud computing on-demand resources based on pattern matching. Research Report RR-7217, INRIA, 2010. URL `http://hal.inria.fr/inria-00460393`.

[48] Eddy Caron, Frédéric Desprez, and Adrian Muresan. Pattern Matching Based Forecast of Non-periodic Repetitive Behavior for Cloud Clients. *Journal of Grid Computing*, 9(1):49–64, January 2011. ISSN 1570-7873. doi: 10.1007/s10723-010-9178-4. URL `http://www.springerlink.com/content/5371210671434mkk/`.

[49] Abhishek Chandra, Weibo Gong, and Prashant Shenoy. Dynamic resource allocation for shared data centers using online measurements. *Proceedings of the 11th international conference on Quality of service*, pages 381–398, June 2003. URL `http://dl.acm.org/citation.cfm?id=1784037.1784065`.

[50] G Chen, W He, J Liu, S Nath, L Rigas, L Xiao, and F Zhao. 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*, volume 8, pages 337–350. USENIX Association, 2008.

[51] T C Chieu, A Mohindra, A A Karve, and A Segal. Dynamic scaling of web applications in a virtualized cloud computing environment. In *e-Business Engineering, 2009. ICEBE'09. IEEE International Conference on*, pages 281–286. Ieee, 2009.

[52] T C Chieu, A Mohindra, and A A Karve. Scalability and Performance of Web Applications in a Compute Cloud. In *e-Business Engineering (ICEBE), 2011 IEEE 8th International Conference on*, pages 317–323. IEEE, 2011.

[53] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms, Chapter 32: String Matching.* McGraw-Hill Higher Education, July 2001. ISBN 0070131511. URL `http://dl.acm.org/citation.cfm?id=580470`.

[54] X Dutreilh, A Moreau, J Malenfant, N Rivierre, and I Truck. From data center resource allocation to control theory and back. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 410–417. IEEE, 2010.

[55] Xavier Dutreilh, Sergey Kirgizov, Olga Melekhova, Jacques Malenfant, Nicolas Rivierre, and Isis Truck. 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*, pages 67–74. IEEE, May 2011. ISBN 978-1-61208-006-2.

[56] H Ghanbari, B Simmons, M Litoiu, and G Iszlai. Exploring Alternative Approaches to Implement an Elasticity Policy. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 716–723. IEEE, 2011.

[57] Z Gong, X Gu, and J Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9–16. IEEE, 2010.

[58] R Han, L Guo, M.M Ghanem, and Y Han, R. and Guo, L. and Ghanem, M.M. and Guo. Lightweight Resource Scaling for Cloud Applications. *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, 2012. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6217477`.

[59] M Z Hasan, E Magana, A Clemm, L Tucker, and S L D Gudreddi. Integrated and autonomic cloud resource scaling. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 1327–1334. IEEE, 2012.

[60] J Huang, C Li, and J Yu. Resource prediction based on double exponential smoothing in cloud computing. In *Consumer Electronics, Communications and Networks (CECNet), 2012 2nd International Conference on*, pages 2056–2060. IEEE, 2012.

[61] Waheed Iqbal, Matthew N. Dailey, David Carrera, and Paul Janecek. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Generation Computer Systems*, 27(6):871–879, June 2011. ISSN 0167739X. doi: 10.1016/j.future.2010.10.016. URL `http://dl.acm.org/citation.cfm?id=1967762.1967921`.

[62] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Gener. Comput. Syst.*, 28(1): 155–162, 2012. ISSN 0167-739X. doi: 10.1016/j.future.2011.05.027. URL `http://dx.doi.org/10.1016/j.future.2011.05.027`.

[63] E Kalyvianaki, T Charalambous, and S Hand. Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In *Proceedings of the 6th international conference on Autonomic computing*, pages 117–126. ACM, 2009.

[64] Sunirmal Khatua, Anirban Ghosh, and Nandini Mukherjee. Optimizing the utilization of virtual resources in Cloud environment. In *2010 IEEE International Conference on Virtual Environments, Human-Computer Interfaces and Measurement Systems*, pages 82–87. IEEE, September 2010. ISBN 978-1-4244-5904-9. doi: 10.1109/VECIMS.2010.5609349. URL http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=5609349\&contentType=Conference+Publications.

[65] Jonathan Kupferman, Jeff Silverman, Patricio Jara, and Jeff Browne. Scaling into the cloud. Technical report, University of California, Santa Barbara; CS270 - Advanced Operating Systems, 2009. URL http://cs.ucsb.edu/~jkupferman/docs/ScalingIntoTheClouds.pdf.

[66] Palden Lama and Xiaobo Zhou. 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*, pages 151–160. IEEE, August 2010. ISBN 978-1-4244-8181-1. doi: 10.1109/MASCOTS.2010.24. URL http://dl.acm.org/citation.cfm?id=1906481.1906523.

[67] Harold C Lim, Shivnath Babu, Jeffrey S Chase, and Sujay S Parekh. Automated control in cloud computing: challenges and opportunities. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, ACDC '09, pages 13–18, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-585-7. doi: 10.1145/1555271.1555275. URL http://doi.acm.org/10.1145/1555271.1555275.

[68] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. Automated control for elastic storage. In *Proceeding of the 7th international conference on Autonomic computing - ICAC '10*, page 1, New York, New York, USA, June 2010. ACM Press. ISBN 9781450300742. doi: 10.1145/1809049.1809051. URL http://dl.acm.org/citation.cfm?id=1809049.1809051.

[69] Ming Mao and Marty Humphrey. 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*, page 1, New York, New York, USA, November 2011. ACM Press. ISBN 9781450307710. doi: 10.1145/2063384.2063449. URL http://dl.acm.org/citation.cfm?id=2063384.2063449.

[70] Michael Maurer, Ivona Brandic, and Rizos Sakellariou. Enacting SLAs in clouds using rules. pages 455–466, August 2011. URL http://dl.acm.org/citation.cfm?id=2033345.2033393.

[71] Daniel A. Menasce, Lawrence W. Dowdy, and Virgilio A. F. Almeida. *Performance by Design: Computer Capacity Planning By Example*. Upper Saddle River, NJ: Prentice Hall, January 2004. ISBN 0130906735. URL http://dl.acm.org/citation.cfm?id=995032.

[72] H Mi, H Wang, G Yin, Y Zhou, D Shi, and L Yuan. Online self-reconfiguration with performance guarantee for energy-efficient large-scale cloud computing data centers. In *Services Computing (SCC), 2010 IEEE International Conference on*, pages 514–521. IEEE, 2010.

[73] Simon Ostermann, Kassian Plankensteiner, Radu Prodan, and Thomas Fahringer. GroudSim: an event-based simulation framework for computational grids and clouds. pages 305–313, August 2010. URL http://dl.acm.org/citation.cfm?id=2031978.2032020.

[74] P Padala, K Y Hou, K G Shin, X Zhu, M Uysal, Z Wang, S Singhal, and A Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 13–26. ACM, 2009.

[75] Sang-Min Park and Marty Humphrey. Self-Tuning Virtual Machines for Predictable eScience. In *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 356–363. IEEE, May 2009. ISBN 978-1-4244-3935-5. doi: 10.1109/CCGRID.2009.84. URL http://www.computer.org/portal/web/csdl/doi/10.1109/CCGRID.2009.84.

[76] T Patikirikorala and A Colman. Feedback controllers in the cloud. *APSEC 2010, Cloud workshop*, 2010.

[77] Tharindu Patikirikorala, Alan Colman, Jun Han, and Liuping Wang. A multi-model framework to implement self-managing control systems for QoS management. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 218–227, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0575-4. doi: 10.1145/1988008.1988040. URL http://doi.acm.org/10.1145/1988008.1988040.

[78] Jia Rao, Xiangping Bu, Cheng-Zhong Xu, Leyi Wang, and George Yin. VCONF: a reinforcement learning approach to virtual machines auto-configuration. In *Proceedings of the 6th international conference on Autonomic computing*, ICAC '09, pages 137–146, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-564-2. doi: 10.1145/1555228.1555263. URL http://doi.acm.org/10.1145/1555228.1555263.

[79] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 500–507. IEEE, July 2011. ISBN 978-1-4577-0836-7. doi: 10.1109/CLOUD.2011.42. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6008748.

[80] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. *Proceedings of the 2nd ACM Symposium on Cloud Computing*, 2011. URL http://dl.acm.org/citation.cfm?id=2038921.

[81] B Simmons, H Ghanbari, M Litoiu, and G Iszlai. Managing a SaaS application in the cloud using PaaS policy sets and a strategy-tree. In *Network and Service Management (CNSM), 2011 7th International Conference on*, pages 1–5, 2011.

[82] SPECweb2009. The httperf HTTP load generator. `http://www.spec.org/web2009/`, 2012. [Online; accessed 18-September-2012].

[83] G Tesauro, N K Jong, R Das, and M N Bennani. A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation. In *Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, ICAC '06, pages 65–73, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 1-4244-0175-5. doi: 10.1109/ICAC.2006.1662383. URL `http://dx.doi.org/10.1109/ICAC.2006.1662383`.

[84] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of multi-tier Internet applications. *ACM Transactions on Autonomous and Adaptive Systems*, 3(1):1–39, March 2008. ISSN 15564665. doi: 10.1145/1342171.1342172. URL `http://dl.acm.org/citation.cfm?id=1342171.1342172`.

[85] D Villela, P Pradhan, and D Rubenstein. Provisioning servers in the application tier for e-commerce systems. In *Quality of Service, 2004. IWQOS 2004. Twelfth IEEE International Workshop on*, pages 57–66. IEEE, 2004.

[86] L Wang, J Xu, M Zhao, Y Tu, and J A B Fortes. Fuzzy Modeling Based Resource Management for Virtualized Database Systems. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*, pages 32–42. IEEE, 2011.

[87] Lixi Wang, Jing Xu, Ming Zhao, and José Fortes. Adaptive virtual resource management with fuzzy model predictive control. In *Proceedings of the 8th ACM international conference on Autonomic computing - ICAC '11*, page 191, New York, New York, USA, June 2011. ACM Press. ISBN 9781450306072. doi: 10.1145/1998582.1998623. URL `http://dl.acm.org/citation.cfm?id=1998582.1998623`.

[88] Jing Xu, Ming Zhao, Jose Fortes, Robert Carpenter, and Mazin Yousif. On the Use of Fuzzy Modeling in Virtualized Data Center Management. In *Proceedings of the Fourth International Conference on Autonomic Computing*, ICAC '07, pages 25—-, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2779-5. doi: 10.1109/ICAC.2007.28. URL `http://dx.doi.org/10.1109/ICAC.2007.28`.

[89] Q Zhang, L Cherkasova, and E Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *Autonomic Computing, 2007. ICAC'07. Fourth International Conference on*, page 27. IEEE, 2007.