A Survey on Elasticity Management in PaaS Systems

Francesc D. Muñoz-Escoí · José M. Bernabéu-Aubán

Abstract Elasticity is a goal of cloud computing. An elastic system should manage in an autonomic way its resources, being adaptive to dynamic workloads, allocating additional resources when workload is increased and deallocating resources when workload decreases. PaaS providers should manage resources of customer applications with the aim of converting those applications into elastic services. This survey identifies the requirements that such management imposes on a PaaS provider: autonomy, scalability, adaptivity, SLA awareness, composability and upgradeability. This document delves into the variety of mechanisms that have been proposed to deal with all those requirements. Although there are multiple approaches to address those concerns, providers' main goal is maximisation of profits. This compels providers to look for balancing two opposed goals: maximising quality of service and minimising costs. Because of this, there are still several aspects that deserve additional research for finding optimal adaptability strategies. Those open issues are also discussed.

Keywords Cloud computing · Scalability · Adaptability · Elasticity · Service Level Agreement · PaaS · Workload prediction · Reactive management

Mathematics Subject Classification (2000) 68-00 · 68M14 · 68M15 · 68M20 · 68N01 · 90B18 · 90B22 · 90B25

1 Introduction

Availability and scalability are two common requirements for distributed services. The advent of cloud computing [7] has partially broken the existing limits on scalability. *Infrastructure as a service* (IaaS) providers may supply a large infrastructure for distributed application deployment. For most applications, this is almost equivalent to an unlimited source of resources. So, infrastructure consumers should minimise

Francesc D. Muñoz-Escoí · José M. Bernabéu-Aubán
Instituto Universitario Mixto Tecnológico de Informática, Universitat Politècnica de València, 46022 Valencia (Spain) E-mail: {fmunyoz, josep}@iti.upv.es

resource usage when the workload being managed by their applications is processed. This introduces the need for a management of elasticity.

Elasticity management is not trivial. Ideally, that management should be done by specialised companies: the *platform as a service* (PaaS) providers. The PaaS layer is placed on top of an IaaS layer in a regular cloud computing architecture. According to the NIST definitions [70], PaaS is a service model where "*the capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment.*" That definition demands that PaaS customers deal with some general "*configuration settings for the application-hosting environment*" but adaptability management is a responsibility of PaaS providers. Since every service should be elastic [29, 43], this means that PaaS providers should deal with the mechanisms that automate service scalability and adaptability.

However, the level of automation needed to approach a cost-optimal service exploitation[1] is still challenging because of the many aspects that should be considered. To begin with, scalability decisions must match what has been stated in *service level agreements* (SLA). This means that those decisions should be taken as soon as the workload or the service performance starts to vary, which strongly suggests the need to include some workload prediction mechanisms. However, forecasting techniques are not perfect. So, they should be complemented with other reactive mechanisms; i.e., when the resulting service performance levels do not comply with what is being specified in the SLA or lead to unnecessary overprovisioning costs, service providers should react. Those reactions may consist in adding service instances or migrating those instances to better VMs, when the service capacity should be increased, or in releasing instances when service capacity needs to be decreased.

Scalable distributed interactive services (e.g., social networks, electronic commerce,...) are concurrently used by many customers [8]. According to the SLAs, service providers should ensure service continuity. Otherwise, service customers will not rely on those providers. Unfortunately, both platform and service components need to be eventually upgraded [96] in order to fix bugs, remove security vulnerabilities or enhance their functionality, and these upgrades usually cause service disruptions. So, this is another source of trouble for service providers.

Our goal in this document is to provide a tutorial on the requirements to be considered and the current solutions to the challenges being present in elastic PaaS systems. Note that in an IaaS system the resource being provided is a virtualised hardware, and elasticity management in that scope refers to the ability to assign or deallocate virtual machines to the customer, but the rules to be used to this end should be decided by IaaS customers. In a PaaS system, on the other hand, the provider receives all application components and a deployment manifest and that provider should manage in a transparent way all the elasticity-related decisions for complying with a set of

[1] Cost optimality implies that a minimal set of resources should be assigned for deploying that service, reducing in this way the provision efforts and the customer payments.

high-level application goals (that should be adequately stated in the SLA). Thus, the responsibility for decision taking is placed at the PaaS layer and should be dealt with in a transparent way to the PaaS customer. That level of transparency and automation is still an ideal target, but such target might be reached with some improvements on the mechanisms described in the following sections.

In order to select the papers to be discussed in the following sections, a bibliographic searcher was used (concretely, Google Scholar). Its results were filtered requiring at least 20 citations per selected paper and applying a short review of each candidate paper in order to check whether it might be used in a PaaS service model. This first stage was followed by both a forward and a backward trace of citations among the most relevant papers returned in those searches in order to find mechanisms for dealing with each one of the identified elasticity requirements. The papers collected in this second stage were filtered again: they do not need a high citation count, but they should be either recent Ph.D. theses or publications in relevant journals (i.e., indexed in Thomson Reuters Journal Citation Reports) or conferences (e.g., indexed in the ranking from the Computing Research & Education Association of Australia[2] or in the Thomson Reuters Conference Proceedings Citation Index).

As a result, this paper will not present an exhaustive survey of PaaS systems and their characteristics (indeed, some surveys of that latter kind already exist, as we will see in Section 2). Instead, it lists and describes the different techniques that may comply with those requirements.

The rest of this paper is structured as follows. Section 2 provides a set of basic definitions and describes some related work. Section 3 describes the provider requirements found in the PaaS service model. Section 4 explains the mechanisms being used in order to deal with those requirements. Section 5 discusses pending problems in this area. Finally, Section 6 concludes this document.

2 Background and Related Work

As it has been outlined in the introduction, cloud service models usually define a layered architecture that was first suggested by Vaquero González et al [112]. It consists, from top to bottom, of the following layers:

- *Software as a Service* (SaaS): In this layer distributed applications are offered as a service to a large set of potential users.
- *Platform as a Service* (PaaS): In this layer a development platform is provided to the customers and application deployment and elasticity management tasks are automated by the provider.
- *Infrastructure as a Service* (IaaS): In this layer a set of virtualised hardware resources (i.e., those needed to provision processing, storage, networks,...) is provided to the customers.

Complete definitions for each service model were proposed by the *National Institute of Standards and Technology* (NIST), and they may be found in [70]. Its PaaS service model definition has been already quoted in Section 1.

[2] That ranking is available at *http://www.core.edu.au/conference-portal*.

In this paper multiple concepts related to services and service-related roles are used. In order to avoid ambiguity, the following definitions should be considered:

- *Distributed application*. Set of distributed programmes that needs to be deployed in order to provide a service to a set of users. Each programme of that set implements a component of the application.
- *Distributed service*. The service provided by a distributed application once it has been deployed in a given infrastructure and its elasticity is correctly managed. Since the goal of this survey is to identify the research challenges being generated by some elasticity requirements in PaaS systems in order to describe some of their solutions, we will assume that distributed services are stateful. This means that servers maintain information on their clients [13, page 115]. Otherwise, with stateless servers, some of the problems being described might be solved in an easier way. For instance, *software upgrading* (Section 4.6) does not need any transformation stage [41] in the stateless case.
- *IaaS provider* (IP). The company that maintains a set of data centres that may be used by external customers following an IaaS service model.
- *PaaS provider* (PP). The company that manages the deployment and elasticity of distributed applications on top of a given scalable infrastructure. A PP, given the layered architecture described above, also behaves as an IaaS customer.
- *SaaS provider* (SP). The company that manages the provision of distributed services to their final users. A SP, given the layered architecture described above, also behaves as a PaaS customer.
- *Service user* (SU). The final user of the distributed services. A SU also behaves as a SaaS customer.

The main goal of our paper is to identify the basic requirements for achieving elasticity at the PaaS layer, describing the main mechanisms being needed to this end. Some previous surveys on elasticity in the cloud computing field have been published, although they are not focused on elasticity requirements. Let us briefly summarise their contributions in a chronological order.

Calcavecchia et al (2012) [18] provide a concise taxonomy of auto-scaling approaches. Its classification considers four axes: (1) contribution focus (customer-centric or provider-centric), (2) self-management type (exogenous or endogenous), (3) autonomic mechanisms (goal, action, utility or heuristic), and (4) provisioning mechanisms (replication, resizing or migration). Despite its short length, the main contributions of the relevant papers at that time are correctly summarised, providing a good first introduction to this research area.

Galante and De Bona (2012) [33] present a global study on cloud elasticity centred on four different axes: scope (infrastructure vs. platform), policy (manual, automatic predictive or automatic reactive), purpose (performance, infrastructure capacity, cost or power consumption) and method (replication, resizing or migration). That classification allows a rapid understanding of the mechanisms being used for managing elasticity and of the challenges that exist in this field. Unfortunately, due to space constraints, that paper cannot enter into detail in its descriptions.

Najjar et al (2014) [80] provide a more extended survey on cloud elasticity. Its first contribution is its deep analysis of elasticity in the PaaS service model, although

infrastructure-related approaches are also described. A second contribution is its consideration of multiple kinds of SLOs. Besides regular QoS SLOs, Najjar et al also consider *quality of experience* (QoE) and *quality of business* (QoBiz) goals.

Coutinho et al (2015) [24] centre their attention in evaluating the metrics to be collected by the monitoring subsystem. It studies all service models: IaaS, PaaS and SaaS. Besides, that survey starts with a description of the literature review procedure that was followed by its authors, presenting some statistics about the journals, conferences, benchmarks and infrastructure providers (among other aspects) found in that literature review. Because of its extensive discussion on metrics and review procedure, other important aspects (mainly scalability and adaptability mechanisms) have not been thoroughly discussed in that paper.

Naskos et al (2015) [81] present the most elaborated taxonomy on cloud elasticity management. In this case, six different axes are identified in a first classification level, and they are later refined in a second level. The resulting classification is complete, the set of considered references is large, and each system being presented is carefully characterised. Additionally, it also analyses the case of federations of cloud providers, that is an interesting topic further discussed in other specific surveys [40, 111]. However, due to space constraints, that paper cannot describe in detail the mechanisms being needed for elasticity management.

Finally, Galante et al [34] have published in 2016 the most recent survey on cloud elasticity we are aware of. It discusses elasticity in the management of scientific applications [16, 17]; i.e., *high performance computing* (HPC) applications that follow a parallel programming paradigm. In general, public cloud providers (at least at the SaaS and PaaS layers) have assumed that users are interested in interactive web service applications [49]. Scientific applications do not match that pattern; indeed, they commonly follow a batch processing model. A user provides an input file (or a set of input files) that should be processed using specific algorithms in order to find a result, but no client-server interaction is needed in the interim. This poses specific challenges [49] that are discussed in [34], complementing in this way what was presented by Galante and De Bona in [33].

3 Elasticity Requirements

Quoting Nikolas Herbst's definition [43], elasticity in cloud computing is "*the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible.*" Such definition states that elasticity combines two complementary dimensions: scalability and adaptability. This means that elastic services should be able to collect new computing resources when their workload is increased, but also to discard any exceeding resources when that workload decreases. Additionally, this resource management should be autonomous; i.e., driven by the system itself.

Therefore, some initial requirements may be directly derived from that definition:

1. *Autonomy*. An *autonomous manager* [45, 48, 53] should exist, with sensors and effectors on its managed elements; i.e., on the server instances. This manager

should be able to *monitor* the server instances and their environment, *analyse* the collected metrics, *plan* its adapting actions and *execute* them, considering an appropriate *knowledge* base. This conforms the *MAPE-K* [48] reference control model that could be taken as a basis for building an autonomic platform.

2. *Scalability*. Services should be able to scale. To this end, when the service workload increases, the number of server instances should be increased (horizontal scalability) or each instance should be upgraded (vertical scalability).
3. *Adaptability*. Traditionally, scalability has been centred in using appropriate techniques for managing larger sets of resources in order to achieve a (close to) linear increase in service throughput [83]. Decreasing workloads were not considered a problem in those first systems, since their population of customers followed an increasing trend.
   However, nowadays, the set of resources being used should be adapted as soon as possible to the current demands, avoiding overcommitment. Resource overprovisioning incurs in excessive costs, since those resources must be paid. Resource underprovisioning is equally bad, and, depending on the SLA, potentially worse, since the incoming workload will overwhelm the existing servers ruining the QoS compromised in the SLA, with explicit economic consequences, or implicit punishment by customers abandoning the service.

Besides this, according to Dustdar et al [29] *quality of service* (QoS) and self-identifying resources (that help in service composition) are two important aspects in every elastic cloud system. Quality of Service (QoS) requirements are stated in SLAs. QoS and composability have special implications in a PaaS service model. Therefore, the set of requirements to be considered might be extended with:

4. *SLA-awareness*. SPs develop and integrate the software of their service, and expect that PPs automate their management. This automation must be based on some *service level objectives* (SLOs) the SaaS must attain, likely captured by the SaaS SLA with its own customers. To make this work, a SaaS should provide their deployed services with a means of expressing their SLOs, which can then be properly interpreted by the PaaS, and drive the PaaS elasticity actions. The actual SLA between the PaaS and the SaaS must, in turn, cover the degree to which the PaaS guarantees that the SaaS is achieving its SLOs.
   Multiple SLOs may be considered in PaaS systems. There is an agreement on considering *service availability* as the most important SLO to be respected [3].
5. *Composability*. A distributed service may consist of multiple components. Those components are related by a flow of data and processing [117]. The scaling unit of action is the component. Thus, if only one component has become the current performance bottleneck, we might scale out only that component in a first stage. In spite of this, we should still consider that those components are inter-related, carefully analysing which are their dependencies [117].
6. *Minimal service disruption in software upgrades*. Requirements 1, 2 and 3 are basic for achieving elasticity. Requirement 2 (scalability) does not make sense without service continuity. A scalable service increases its set of managed resources in order to deal with increasing loads because its main goal is to answer every client request. When that is not possible, the service becomes unavailable,

at least for those users that do not get their intended answers. Therefore, service continuity (i.e., availability) is a pre-condition for scalability [82, 37].
Software systems are not static. Clearly, software defects will drive change. Additionally, market pressures and customer demands for new functionality also drive change, forcing SPs to come up with newer versions for their software. A careful software upgrading procedure should be designed [63] considering the SLA, and that SLA also demands high availability [82]. In spite of this, several IPs and PPs explicitly exclude their periodical maintenance windows from their availability computations, but this may be confusing for their customers. Indeed, customers may consider that those unavailability intervals are an inconvenience. Therefore, the length of those planned maintenance windows should be minimal.
IPs deal with planned upgrades notifying their maintenance intervals to their customers. With this, customers may react appropriately to those outages.
Since scalability management is a requirement for PPs, PPs are aware of the current degree of replication for each deployed customer application. With this information, and assuming that the upgrade may be done in stages, handling a different set of replicas in each stage, a PP might deal with some kinds of software upgrades (those without state transformations) in a transparent way to its customers.

4 Elasticity Mechanisms

Let us discuss in the following subsections which are the main mechanisms being used nowadays in order to deal with the requirements presented in Sect. 3.

4.1 Autonomy

A PaaS system should be a piece of *autonomic software* [53, 46]. In the IBM blueprint for autonomic computing [48] a software system is considered autonomic when it is able to self-manage with a minimum of human interference. In that case, its software elements should be self-configuring (i.e., adaptive), self-healing (i.e., able to detect, diagnose and react to disruptions), self-optimising and self-protecting. To this end, a set of autonomic software elements should be controlled by an autonomic manager. This defines a control loop with four stages:

1. *Monitor*. Some information is collected from the sensors set on each managed element. These data are also aggregated and filtered in order to build the appropriate reports. The data being considered consists of multiple metrics and descriptions of the current resource topology.
2. *Analyse*. This stage correlates the data obtained in the previous stage and considers some behavioural models (e.g., time-series forecasting, queueing networks...) allowing the autonomic manager to learn from the environment and helping to predict future situations for the managed elements.

3. *Plan*. Considering the information provided by those models, the planning stage decides the actions to be performed in order to reach the system goals, according to the existing policies.
4. *Execute*. Finally, the execution stage applies those actions to the managed elements through the corresponding effectors.

Each stage is endlessly applied and its output is considered by the next one in that loop. To this end, the stages access a shared *knowledge base* where the global policies can be found and the information being generated in each stage is stored.

This general picture of autonomic computing can be easily tailored for the cloud computing systems that follow a PaaS service model. Thus, the general business policies being discussed in [48] become now a combination of:

- A general economic goal: to minimise the cost of the infrastructure required for supporting all customer software to be deployed on the platform.
- A specific set of rules, depending on each customer application to be deployed: the SLA that governs the QoS requirements for each of those applications.

When PaaS elasticity is considered, autonomy refers mainly to the ability of automatically adapting the service to its current workload (i.e., self-configuring and self-optimising) and to its agreed QoS levels. Those QoS levels imply self-healing and self-protecting, since they consider service availability (that implies both aspects and is supported through component replication, to be described in the scope of scalability management). This autonomy is provided by the internal platform elements that are managing the application components installed by the SPs.

Multiple academic proposals (e.g., [19, 75, 90, 97, 114]) follow an architecture that complies with the recommendations from the IBM blueprint. As such, these proposals usually have a monitoring component that collects relevant metrics from the deployed services, an analyser component that uses some kind of performance model in order to compute other derived metrics, a planning component that compares these collected values with the intended target ones (as ruled by the existing SLAs) and decides the appropriate (scaling) actions to be taken in order to correctly adapt those customer's deployed components and, finally, an executor component that applies those actions.

Thus, Kingfisher (Sharma et al [97], 2011) is able to apply the MAPE-K control model onto hybrid (i.e., those that combine a public and a private cloud) deployments. It uses a modified version of the OpenNebula [78] toolkit to implement its cloud management mechanisms. Those mechanisms may be deployed onto Xen-based private clouds or onto the Amazon EC2 public cloud. It consists of four main components:

1. *Monitoring Engine* (ME). It is based on Ganglia [67] or on the monitoring mechanisms provided by the IaaS.
2. *Workload Forecaster* (WF): It should implement any workload prediction mechanism. In that paper, it is emulated by a perfect forecaster.
3. *Capacity Planner* (CP): It is the core of the Kingfisher architecture. It implements an integer linear programming (ILP) algorithm in order to adapt the system capacity.

4. *Orchestration Engine* (OE): Once a configuration has been computed, OE instantiates it using the transition plan. OE uses the interfaces exposed by the cloud management platform to resize VMs, start up new instances, or migrate VMs. OE merely specifies the server type for each configuration step, and leaves the problem of placement of these VMs onto physical servers to the OpenNebula cloud manager.

Those four components are correlated with the stages of the MAPE-K control model.

A second academic proposal is presented by Casalicchio and Silvestri [19] (2013). In its scope, the four MAPE-K stages may be implemented using the following components: the *monitor* stage encompasses a performance monitor and a workload monitor; the *analysis* stage consists in a SLA analyser; the *plan* stage has a provisioning manager; finally, the *execute* stage consists of a load balancer and a VM allocator. Its Section 4 discusses four different ways for distributing all those components between IPs and PPs (the latter are called *application service providers*, ASP, in that paper):

1. *Extreme ASP control*. The PP maintains all the presented components, while the IP only manages a small monitoring agent that reports its information to the load balancer, VM allocator and performance monitor managed by the PP. With this, all the autonomy management is driven by the PP with a minimal assistance from the IP. This option may make sense when the PP follows a private cloud deployment model [70] for its infrastructure.
2. *Full ASP control*. In this second alternative, the VM allocator is moved from the PP to the IP. With this, the PP still drives all the autonomy management decisions but the VM allocation mechanisms are managed by the IP. With this, PPs outsource VM management, relying on external data centres. However, in this architecture the PP is still responsible for organising a scalable network infrastructure.
3. *Partial ASP control*. In this third approach, the load balancing service and the performance monitoring service are both moved onto the IP responsibility. With this, the PP is responsible of the analyse and plan stages (i.e., adaptability decisions), while the IP deals with the monitor and execute ones (i.e., scalability mechanisms). This seems to be the best distribution, since it corresponds in a natural way to the definitions of the PaaS and IaaS service models [112, 70].
4. *Limited ASP control*. In this last variant, the IP is able to manage itself all the MAPE-K stages. To this end, the IP provides complete auto-scaling functionalities. In this case, the PP is only centred in setting the appropriate auto-scaling rules that will be driven by the IP.

In [75], Mohamed (2014) proposes the usage of wrapper classes in order to add autonomy and elasticity management on previously developed services that were not autonomic- or elastic-aware. To this end, these wrappers provide endpoints for monitoring operations and different mechanisms for easily implementing scale-in and scale-out actions. With this, both the monitor and execute stages can be supported in those deployed services. On the other hand, the analysis and plan stages should be managed by the platform, using specific components.

The interested reader may refer to Huebscher and McCann (2008) [46] in order to delve deeper in the autonomic computing research field, in general, and to Singh and Chana (2015) [101] for autonomic cloud computing, in particular.

4.2 Scalability

In order to have an elastic system, the software services it provides must be scalable. This means that when the incoming workload is increased, the capacity of the serving nodes should be also enlarged. We assume that the PP has access to an underlying computing infrastructure (in some cases, rented to external IPs) and that it may use as many computing nodes as needed.

There are different scalability mechanisms [33]:

- *Replication* (i.e., *horizontal scalability*). Horizontal scalability consists in using additional computing nodes for executing server instances when such serving capacity needs to be increased. This may be implemented adding new *virtual machines* (VMs) to the current set, deploying there new replicas of the required service components.
  In order to increase scalability and performance, inter-replica consistency should be relaxed since there is a trade-off between performance and consistency in replicated services [8]. To this end, *eventual consistency* [52, 28, 116] was proposed. This is also a consequence of the CAP theorem [32, 38] since highly scalable interactive services may be deployed in multiple data centres [8, 111] (in order to maintain server replicas close to their intended users, reducing interaction latency) and network partitions may arise in that case. The CAP theorem states that it is impossible that a distributed system ensures strong consistency, availability and network partition tolerance simultaneously for its deployed services. Because of this, consistency is usually relaxed and optimistic replication protocols may be used in that case. An excellent tutorial and survey about optimistic replication protocols and eventual consistency has been written by Saito and Shapiro [95].
  Another consistency management is to delegate state sharing. If there is a need of sharing data, such data may be managed by a specialised external component. An example is a scalable datastore (e.g., Google Bigtable [21], Apache Cassandra [58], Microsoft Azure SQL Database,...), that transparently uses replication and *sharding* [108] in order to ensure a fast update management. A variant in this management guarantees causal consistency complemented with eventual convergence [4, 9, 98, 121]. With this, these datastores are providing the best consistency that can be ensured in a partitionable environment [87] with optimal performance. To this end, transaction updates can be forwarded among replicas using lazy causal propagation [26].
- *Resizing* (i.e., *vertical scalability*). Vertical scalability consists in improving the capacity of the server node. A way for implementing vertical scalability is hardware upgrading. This is supported when VMs are used and those VMs get additional resources from their hosting physical computers; e.g., a larger share of CPU time [71], a larger amount of physical memory, a higher bandwidth in their access

to the network, etc. Therefore, the basic mechanism consists in VM resizing and this is a method for achieving IaaS-level elasticity.

There have been several IaaS-related proposals for managing VM resizing [25, 39, 99]. In most cases, resizing management consists in using a VM manager that interacts with the hypervisor software in order to dynamically change the hardware resources currently assigned to each deployed VM. That VM manager should be complemented with a controller that decides how those resources should be allotted. The configuration of the controller is made by the PP while both VM manager and controller are held by the IP.

For instance, in *Elastic VM* [25] its VM manager contains these elements:

- Resource monitor. It dynamically measures the resources (i.e., CPU, memory,...) consumption and reports their values to the QoS controller. To this end, it depends on the measuring utilities provided by the hypervisor.
- CPU scheduler. It may dynamically change the CPU allocation to each VM according to the QoS controller decisions. Again, it depends on the hypervisor for setting those CPU shares.
- Memory manager. It allocates the memory being needed by each VM.
- Performance monitor. It notifies the current values for the relevant SLO metrics to the QoS controller.
- Application manager. It manages the commands for controlling the life cycle of the deployed VMs.

On the other hand, its QoS controller consists of three elements: (1) a CPU controller, (2) a memory controller, and (3) an application controller. Each one decides respectively the current share of CPU, share of memory and amount of running VMs. These controllers consider the information being provided by the resource and performance monitors in order to give commands to the CPU scheduler, memory manager and application manager for optimising resource usage and SLO compliance. In this way, each deployed VM may be dynamically resized considering both its own SLOs, performance and workload, and the current status of their hosting computer.

Resizing demands a fine control of local hardware resources in each hosting node. Besides, local controllers must also exist in each host, balancing and redistributing the local resources among the locally deployed VMs. Because of this complex management, many IPs do not allow VM resizing at run-time. Note that VM resizing requires that either the host computer has free resources at resizing time or the other guest VMs should also adapt their provisioned set of resources to accommodate such resizing. Instead of that dynamic resizing, most IPs facilitate a static set of VM types and the PP is compelled to select which kind of VM should be used in each image allocation. Therefore, in those cases, vertical scaling should be implemented through VM migration.

- *Migration*. Migration consists in moving the server instances from their current VM to another one in a different host computer [69]. Depending on the size of the instance image, the network bandwidth and the location of the hosts, migration may need several seconds to be concluded. Note that a migration is a three-step procedure: (1) the current instance image should be stopped, (2) its image should be transferred to the new container, and (3) its activity should be restarted there.

If the instance ID and network address should be preserved, some routing reconfiguration will be needed in this last step.

Migration solutions have been considered in multiple research papers. A few examples are described in the sequel.

Sharma et al [97] describe the Kingfisher system where all scaling mechanisms (replication, resizing and migration) can be combined in order to improve service performance. Kingfisher is able to federate private and public infrastructures. VM resizing may be implemented in private infrastructures. OpenNebula is used in that case. On the other hand, VM migration is the mechanism needed in public IaaSes and also when VMs should be moved between different IPs. The goal in [97] is to propose an efficient resource provisioning algorithm. The best results are obtained combining multiple algorithms. The first one is centred in minimising the amount and cost of infrastructure resources. The second one minimises the provisioning latency. A third algorithm, based on integer linear programming, is needed for combining all previous algorithms.

Knauth and Fetzer [55] evaluate the costs of live VM migration procedures. Their results show that even for small VMs (512 MBs of RAM and 8 VPUs) being supported by medium hosts (8 GBs of RAM and 2 CPUs with 4 cores per CPU) and a 1 Gbps network, the time needed for migrating a live VM between two different hosts ranges from 8 to 18 seconds depending on the load. This leads to a service disruption of at least 10 seconds in the best case. Despite these "long" intervals, those migrations were able to noticeably improve service performance when the target host provides a better VM to the migrated instance. In their experiments, average response time was reduced from 1 second in the original (overloaded) host to 250 ms in the target (free) host.

Later [56], the same authors propose a mechanism for stopping and restarting VMs. Such solution is not a migration mechanism but it shares many characteristics with live migration. VM restarting actions last less than a second in this case. To this end, VM restarting is implemented on demand. The central part of the image is loaded in RAM immediately but the remaining parts are loaded on demand, leveraging the virtual memory management provided by the host.

CloudScale [99] uses a workload predictor in order to adjust the resources assigned to each VM. So, it is able to implement vertical scalability using the VM resizing technique described above. However, when the forecast workload in a given host exceeds its capacity, VM migration is also used for complying with the SLOs. In order to decide which host should receive those migrated VMs, the "*scaling conflict handler*" component of the CloudScale system analyses the forecast workload in each host and locates the best target for each migration.

Casalicchio et al [20] analyse the negative impact that VM migration may have on an IP when VM allocations have exhausted the available resources and VMs need to be migrated to external IPs. To this end, an optimisation problem is formalised and different algorithms based on heuristics are proposed. An additional constraint is service availability that should be maintained above a given threshold. The best solution is provided by their NOPT algorithm that follows the *hill climbing* local search method.

Calatrava et al [17] illustrate another use of migration that is not directly related to scalability. Calatrava et al propose a PaaS system for scientific computing able to manage hybrid clouds. In public clouds, in order to minimise customer payments, Amazon EC2 spot instances may be used. Spot VM instances have a variable price that depends on the customers demands but is lower than that of regular VMs. They are rented according to a customer bid. When a spot instance price exceeds the bid, it is deallocated from its customer, but this only happens once the current billing unit is completed (by default, one hour). In this case, migration is used for porting each application computing element from its current spot instance to another once the original spot needs to be deallocated. With the checkpointing and migration algorithms proposed in [17], the application execution costs are minimised without endangering a good throughput.
Note that some migration mechanisms described in the previous paragraphs have been defined in IaaS systems. Therefore, they implement the migration procedures providing in this way a scaling option that may be used by their customers. PaaS systems are such customers and they are able to use those mechanisms when scaling actions must be applied.

4.3 Adaptability

We refer to adaptability as the possibility of adjusting the computing capacity of a given service to the current workload. So, besides being scalable, an elastic service should be adjustable to the workload being received at each moment.

Reinecke et al [91] consider two different dimensions of "being adaptive":

- *Adaptability*. It refers to the potential for adapting a system. This means that system components have been designed considering that they will need to be reconfigured or restructured in some way. It is a static dimension.
  Adaptability is not a synonym for elasticity but only one of its two dimensions. An elastic system should be both scalable and adaptive, focusing that adaptability on the means being needed for guaranteeing scalability and QoS.
  Regular adaptability does not necessarily imply that the target of those reconfigurations will be always scalability on size. For instance, Miedes and Muñoz-Escoí [74] and Ruiz-Fuertes and Muñoz-Escoí [94] propose adaptable meta-protocols that manage multiple multicast or replication protocols, respectively. The resulting systems are adaptable since their users or administrators may select, when needed, the most appropriate protocol for the current workload. However, none of those possible reconfigurations has any immediate effect on the number of nodes.
- *Adaptivity*. This goes a step further. Adaptivity is the system ability to adapt itself. Besides being adaptable, an adaptive system is able to find out when such adapting actions should be applied. This means that an adaptable (i.e., reconfigurable) system becomes adaptive when it is able to provide an autonomous management. Since autonomy is one of the elasticity requirements (already described in Section 4.1), elastic systems need to be adaptive. To this end, their architecture should

include monitoring, analysis, planning and execution subsystems that automate all reconfiguration-related decisions.

There are two main approaches for achieving adaptivity: proactive and reactive. In the *proactive* (or predictive) case, some workload analysis or modelling is done in order to forecast the load levels in the near future. With this knowledge the actions needed to correctly adapt the service capacity to the current workload are taken beforehand. Thus, QoS might be easily guaranteed when those predictions are accurate. On the other hand, with a *reactive* approach, both the workload and the service behaviour are thoroughly monitored and adequate thresholds are set on different metrics. When those thresholds are reached some adapting actions are started.

Gambi et al [35] analyse how to test the elasticity of cloud services. They find that there are multiple adaptability and adaptivity dimensions in this scope. Computing elasticity may be compared with mechanical elasticity and, in that case, the following correspondences may be found: (1) a computing service is *plastic* if it can scale out but it cannot scale in, (2) the *impact factor* measures the time needed for scaling out or scaling in when workload changes arise, (3) the *fatigue factor* measures whether the service is able to afford going over budget within an observation interval, (4) the *shear factor* measures the tolerance to interferences from other services or to resource contention (or exhaustion) in the underlying platform. Test cases and metrics should be defined for evaluating all these aspects at testing time. Additionally, all those goals should be considered when the service is being designed, developed and deployed, improving in this way the adaptivity of the resulting service.

Let us discuss in the sequel some of the existing solutions in each type of adaptivity approach.

4.3.1 Proactive Mechanisms

Software performance prediction has become an interesting topic in the cloud computing field in the last years. *Software performance engineering* (SPE) [103], one of the existing performance prediction approaches, is a research area that has received attention since the 80s [102]. SPE states that performance is one important goal of reliable software products and that adequate performance can only be achieved when such goal has been considered since the software design stages. The architecture of a complex software application conditions its achievable performance. So, software architects should take care in their designs about the modules that will compose the application and about their dependences, building an accurate performance model and evolving and refining it until its predicted performance is considered appropriate.

In 2004, Balsamo et al [10] surveyed the existing techniques in the performance modelling and prediction field. At design time, software architects should take care on the mapping between the software behavioural model and the performance model being used in each technique. One of the existing problems is the gap between those two models (behaviour vs. performance) that might generate inaccurate performance prediction results. In order to fill this gap, further details should be considered in the performance model, but those extensions may complicate the model excessively. No clear solution exists for this issue. Those models should also identify potential performance bottlenecks in the software architecture being analysed, leading to a redesign

when the stated performance goals become unattainable. Another desirable goal is the automation of the performance model derivation from the software behavioural model. Unfortunately, none of the proposals at that time reached such objective.

Many solutions surveyed in [10] were based on UML diagrams as their behavioural model and on queuing networks as their performance model. Queuing networks show the advantage of an adaptive abstraction level, since they may be used both for predicting the high-level performance of the overall architecture and also for studying the performance of each one of its components; i.e., queuing networks are compoundable. Its high-level view allows the identification of which components might become performance bottlenecks.

Other software performance models were identified: process algebras, Petri nets, generalised semi-Markov processes and simulation techniques. Process algebras and simulation techniques deserve additional explanation since they had been used in multiple proposals.

Generally, the approaches based on stochastic process algebras [44] use a single model for representing both the behaviour and the performance of the system being modelled; i.e., the process algebra has those two roles. This requires an additional effort from the system architect or software designer since the expressiveness of such tool is quite limited in the behavioural semantics; i.e., in order to specify the software functionality. This may lead to problems in the development stage, needing a complementary design using other tools. Besides that problem, algebras commonly derive Markov chains that may lead to a state explosion when the predicted performance results should be computed. So, this model does not scale appropriately and it can only be used in small systems consisting of a few components, operations and steps in order to get precise performance predictions.

Simulation approaches [6, 27] take as their basis a set of UML diagrams and, using a simulation framework, generate a performance simulator programme for that system. This simulator-generation step demands additional information that can be provided either with extended UML diagrams and/or notations [27] or with additional input requested by the converter [6].

All those performance prediction techniques have been conceived for their usage in the first stages of the software life cycle. They may be used in software services to be deployed in a cloud platform. In that case, they help the architects of those applications; i.e., SPs. On the other hand, PPs may use those approaches to predict the performance of the platform components. However, PPs should consider an additional concern: how to predict the performance of the software components being deployed on their platforms. As it refers to this latter concern, the PP receives a set of components to be deployed. Those components have already been designed and developed by other people. That PP should find a tool for modelling the performance characteristics of those components in order to decide how many instances of each of them should be run to obtain SLA-compliant performance levels. In the end, this might be a very different problem that may be solved using the information provided for deploying services.

Some solutions being used in academic proposals are discussed hereafter.

To begin with, Bennani and Menascé [12] describe a system that uses analytical queueing networks and *mean value analysis* (MVA) for assigning in the best way

the existing hardware resources to a set of customer applications. To this end, local and global controllers are used. A local controller consists of: (i) a *workload monitor* that collects the current workload level and stores it in a workload database; (ii) a *workload forecaster* reads such history of workload levels and predicts the forthcoming workload levels using some statistical techniques; (iii) a *predictive model solver* takes as its input the results of the previous two modules and the current number of servers assigned to that application in order to predict its service performance; (iv) a *performance monitor* reports the current performance levels to the next module; (v) finally, a *utility function evaluator* compares the results of the predictive model solver and the performance monitor in order to compute the current utility function value. Such value depends on the SLA, since SLA violations might imply economical penalties and SLA compliance usually implies economical benefits.

Those per-application local controllers are complemented by a *global controller*. This global controller consists of three elements: (i) the *global controller driver* decides when the other two components should be run (periodically by default, but also each time the global utility function is changed); (ii) the *global controller algorithm* uses a combinatorial search technique for predicting the performance to be achieved by different server distributions among the existing applications; to this end, it uses the local controllers (that receive as their input, the intended number of servers to be analysed); (iii) finally, the *global utility function evaluator* summarises the information provided by each local controller evaluator and reports it to the global controller algorithm. When the global controller algorithm finds a server configuration able to improve the current global utility function value, it starts a server redeployment.

This architecture mixes the two approaches previously commented. On one hand, traditional queueing network models are used for predicting the performance of some server configurations. On the other hand, workload predictors are used to complement such study. Finally, the results of that combination are expanded and multiple server configurations are analysed in order to find the optimum one for the utility function. Besides this, a provider needs to evaluate all resources being used or demanded by all customers. So, two controlling levels are needed: per-application and global. Apparently, such proposal seems to be quite complete, and the paper also provides an experimental evaluation that confirms the adequacy of that solution for dynamically allocating HW resources to a set of applications that are deployed in a given data centre. However, the workload being studied was generated by a simulator. In general, proactive mechanisms still need to be evaluated in a production setting, with a real workload. Depending on the applications and the environment, the workload being supported may show variability trends that could be hard to forecast.

Casalicchio and Silvestri [19] compare five different adaptive strategies for server allocation. Most of its presented adaptive mechanisms can be considered reactive, but one of them bases its decisions on a request arrival rate predictor. Such predictor is based on a simple statistical adjustment applied to the recent history of arrivals. The experimental results show that in one of the configurations being studied, with such a predictor the proposed system is able to minimise its economical costs (using the lowest amount of hosts) providing also one of the best average response times.

The ASAP [51] subsystem combines five different basic predictors. Each predictor tries to find out when a new VM will be requested or when an already used VM

will become unnecessary, using to this end the recent history of deployment requests. Since five different predictors are used, their outputs are compared afterwards with the real values and thus, ASAP may choose the best recent predictor at each time. Once it has been chosen, future demands are computed using a regression model and a correlation model on such predictor output. This strategy should provide good levels of adaptability since the proactive subsystem implements five predictive approaches and is able to compare their results and revise their accuracy afterwards. So, the chosen mechanism would not be bad. However, nothing is said in the paper about the computing efforts needed for dealing with all those predictive approaches at once.

In [10], Petri nets were identified as a valid tool for performance modelling, but only used in a few systems. Mohamed [75] presents an example of this kind. His work consists in providing a framework for adapting any existing distributed service for its deployment on the cloud. To this end, component wrappers are used. Those wrappers deal with the monitoring tasks. Using these monitored values as its input data, Petri nets are able to model the overall performance of the involved components. Using such model, the platform may set some performance thresholds for controlling the scale-out or scale-in decisions. So, this is an example of a hybrid approach where a performance model (initially predictive) is used for setting the performance thresholds that will condition the reactive actions.

Roy et al [93] propose a predictive performance model based on *mean value analysis* (MVA), that is complemented with a workload forecaster based on an *autoregressive moving average* (ARMA) method. ARMA consists in assigning decreasing weights to the previous arrival rates maintained in a history knowledge base. On the other hand, MVA is used in this case in a response time analysis algorithm. Such algorithm starts with a minimal assignment of machines for each involved server and it evaluates how the overall response time evolves when new machines are added for deploying the service. This evaluation concludes when the available machines are exhausted. All computed values are later compared and the optimal ones are taken as those that will drive the redeployment of services. The experimental results show that the combination of a performance model and a workload predictor provides optimal results for managing an adaptive resource allocator.

Kingfisher [97] combines a workload analyser with a capacity planner that uses three different resource management strategies. The first strategy is an *infrastructure cost-aware provisioning*. Given the estimated peak workload λ_1, λ_2... λ_k that must be sustained at each component i, the goal is to compute which type of cloud server to use and how many instances at each component so as to minimise infrastructure cost. This provisioning algorithm involves two steps: (a) for each type of server, compute the maximum request rate that it can service at a component, and (b) given these server capacities, compute a least-cost combination of servers that has a capacity of at least λ_i. The second strategy is a *transition cost-aware provisioning*. The provisioning approach must be able to estimate the latency of using different provisioning mechanisms, such as replication, live migration, shutdown migration and resizing. Considering the latency of such mechanisms, an optimal configuration is chosen. Finally, as its third strategy, an *integer linear programming algorithm* mixes the results of the two previous approaches.

DejaVu [114] is a Kingfisher's workload analyser. Client requests are forwarded by some proxies to a workload profiler. This profiler collects a set of low-level performance and resource usage metrics, building a sufficiently large set of historical data. With this, workload is clustered, identifying a set of workload classes and the amount of provisioned resources being needed in each class for guaranteeing the SLOs. This is the learning phase. Once in the production stage, workload is continuously monitored in order to appropriately identify the current workload class. When such workload class changes, the resource allocation strategy is also changed. The resulting model includes also an "interference" metric that includes the throughput interference caused by other concurrent services deployed in the same platform. DejaVu is able to measure such interference adapting its resource allocation appropriately.

The ADVISE [22] framework monitors multiple resource metrics, analysing their historical information (i.e., profiling their behaviour) and considering multiple service components or parts with their deployment information and flow dependencies. With this, a complete picture of each service is taken and the prediction accuracy is highly improved. Additionally, those predictions are compared afterwards with the really obtained SLO values. In our opinion, this is the most appropriate strategy to adopt in proactive approaches.

PREvent [62] also applied those comparisons to web service compositions, improving thus the prediction accuracy. PREvent based its forecasting mechanisms on machine learning. It was one of the first proposals that complemented its prediction management with adapting actions in order to comply with the SLA of composed services.

4.3.2 Reactive Mechanisms

A strictly reactive mechanism for adaptivity establishes a set of rules based on some metric thresholds. Those rules define the actions to be taken when their associated thresholds are reached or surpassed. Thresholds may be set based on heuristics or on a performance model that has considered the current resource deployment. Such set of rules is generally static, but the thresholds being used may be updated at runtime. The input for those rules is the current set of measurements taken by the platform monitoring subsystem. What distinguishes a reactive from a proactive adaptive mechanism is that in the former no workload prediction is needed: only the current workload levels (e.g., request arrival rate) or any other resource usage metrics (e.g., CPU utilisation, size of the resource queues,...) are considered. Many systems and proposals adhere to these general principles, e.g. [75, 76, 84, 90, 100, 120].

Let us explain an example that presents an evolution on these basic principles. As it has been shown previously, Casalicchio and Silvestri [19] compare multiple adaptive strategies. Only one of them is predictive. The other four are reactive. Those reactive policies are based on two different metrics: CPU utilisation and response latency. On each metric, two different strategies are considered: to use one or two thresholds for each kind of decision. Let us describe two of those policies:

- *UT-1al*: It uses a single threshold for each kind of decision. Thus, when the CPU utilisation is greater than 62%, a new server instance is added to the set and, when the CPU utilisation is below 50%, one instance is removed.

– *UT-2al*: A second threshold is added. When the CPU utilisation exceeds 70% two instances are added. When it is lower than 25% two instances are deallocated.

The performance results from [19] show that its predictive policy is better than all the proposed reactive ones when the metrics are evaluated every minute: it minimises the resource costs and completes all benchmark tasks sooner than in any other strategy. On the other hand, if metrics are evaluated every 5 minutes, then the reactive policy based on the response latency metric with a single threshold has a minimal resource budget (even lower than the predictive policy with a 1-minute evaluation interval) and also with the minimal completion time (26% shorter than with the predictive 1-minute policy), although the reactive policy based on the CPU utilisation metric with two thresholds is able to reach minimal values for the service response time. Therefore, there is no clear winner among all the presented policies.

These results raise some questions about what is the best metric evaluation interval for taking reactive actions. In that particular example, it seems that using a 5-minute length is better than using the shortest one. For instance, in case of a *scaling out* action for a particular service S1, requesting the addition of a new server instance, the platform components should deploy a new server image and might need some component re-configuration. This needs some time and effort, and may require an additional interval for workload stabilisation among all S1 instances. If the metrics being considered are read again too early, they might provide counter-producing results, leading to unnecessary scaling actions.

On the other hand, for scale-in actions we should also consider the length of the billing interval. The common length is one hour. So, the PP will select the instance that has almost consumed that 1-hour interval, if any. When all the existing instances have recently started its current billing interval, it is a nonsense to deallocate any of them. In those cases, such instances are maintained, although one of them is tagged for being deallocated afterwards. Such mark will be removed if the workload is increased before stopping that selected instance.

Reactive mechanisms have an implicit issue: to select a particular resource usage metric on which the SLOs directly depend. Cloud services have SLAs that specify their performance objectives. For each objective, there might be multiple relevant resource usage metrics to consider. Reactive strategies select some of those metrics and set some value thresholds on them. So, which is the minimal set of resource usage metrics to be selected in order to adequately manage a deployed service? There is no clear answer for this question. Yataghene et al [120] propose the usage of performance models in order to evaluate off-line the adequacy of different metrics for driving those reactive strategies. This makes sense but such preliminary evaluation needs to be confirmed later by a good behaviour of those schemes in the production stage.

4.4 SLA-awareness

Cloud providers following any service model (either SaaS, PaaS or IaaS) define a special kind of adaptive system with autonomic behaviour, since the customer-provider

relation is driven by a *service level agreement* (SLA). In the original proposal of autonomic computing [45] the main objective was the automation of the management tasks in a software control cycle, but SLAs were not considered at that point.

In a cloud ecosystem, SLAs partially set the goals to be achieved by a provider regarding service quality [101]. The other goals are related to its "quality of business" (QoBiz) [80]; i.e., those other goals deal with reducing service provisioning costs in order to maximise the business benefits.

This introduces a first differential factor between general autonomic computing systems and cloud computing. All non-functional quality aspects might be included in a SLA and such SLA defines the quality-related objectives for those cloud providers.

Besides those *service level objectives* (SLO), a SLA should also specify which are the penalties applied to a provider when those SLOs are not achieved, assuming the client satisfies its part of the deal.

In the (ideal) PaaS service model, PPs should manage SLAs with a rich set of objectives. Thus, SPs could select the most appropriate system for deploying and managing their applications: that one with best relation between available resources, application-level performance guarantees and renting costs.

Although several academical papers and projects assuming a PaaS model seem to manage simultaneously multiple SLOs [19, 59, 63, 73, 72, 75, 93, 97] (e.g., response time for interactive services, throughput for batch services, service availability...) such variety is lost in actual commercial systems. Most public providers only manage a few SLOs, or even only one, the most important: service availability. So, one of the existing challenges for PPs is to elastically manage the services being deployed by multiple customers providing and guaranteeing a rich set of service level objectives in their SLAs.

4.5 Composability

The services being deployed in a PaaS system should be automatically scaled. There are multiple approaches to achieve this. In the first place, those services should not be monolithic; if they were, the scaling decisions would be applied to a single element. This would mean a lack of flexibility when we try to improve service performance, since we could only add, remove, resize or migrate instances of that unique element. Moreover, that element would be larger than the regular elements of a multi-component service and this would have complicated any scaling actions. If the service is implemented by multiple small components (or subservices), their scaling actions will be faster and cheaper, since the amount of resources needed to manage them will be also smaller. Those actions should consider also the data and functional dependencies among their involved components [50].

Distributed services consist of multiple components that may be architected in layers [92]. Such composability allows that the proactive performance models consider the behaviour of each component. Thus, performance prediction models may be used to identify those components that are close to their saturation point, applying then the most appropriate scaling action on them. These components are replicated in the regular case and define elements that are smaller than in monolithic designs. Be-

ing replicated, software upgrades also become easier than in a non-replicated architecture, since the new software versions (when their interfaces are preserved) could be applied replica by replica without endangering service continuity [96].

Service composability has other implications, too. On one hand, some of the components being needed in the implementation of a new service may have been already developed by the same team, since carefully architected services may generate (or need) reusable components. On the other hand, some of the components being needed may be other services developed by other teams. In some cases, those components might have been already deployed in other platforms or in other data centres. Then, inter-service dependences should be declared in the deployment descriptors and the SLAs of those external services should be analysed in order to establish the SLA of such global service.

Careless composability might complicate the definition of the behavioural and performance models being needed in the proactive adaptivity approaches summarised in Section 4.3.1. So, distributed services to be deployed on a platform should be carefully architected, stating the dependences among related components and taking care of how such dependences have been considered in the SLAs. Indeed, Dustdar et al [29] already identified in 2011 the convenience of using self-describing components to build elastic services. With them, it would be easy to state which are the inter-component dependences. When those dependences have been documented and are known by the scaling managers, any scaling action on a given component will also raise appropriate complementary scaling actions in its dependant components. This is convenient, since these joint actions shorten the adaptivity intervals of the encompassing service, as it happens in the ADVISE framework [22].

Finally, Mohd Yusoh [77] considers inter-component dependencies in his evolutionary algorithms for solving three service management problems: component placement at deployment time, component migration and component replication. With that explicit inter-component evaluation, those algorithms reach better solutions than regular heuristic algorithms that assume that all service components are homogeneous, according to the assessments shown in [77]. These proposals are intended for SPs in [77], but they may be ported without problems to the PaaS layer, since deployment, migration and replication management tasks are usually a responsibility of the PP.

4.6 Minimal Service Disruption in Software Upgrades

Software needs to be updated due to multiple causes that have been already outlined in Section 3. This upgrading process has another constraint for cloud providers: they should take care of the SLAs. A possible set of principles and mechanisms to consider in the software upgrade stage taking into account SLAs has been described by Li in [63]. Let us describe that proposal.

When some software components need to be upgraded respecting some SLA, the upgrading system should take care of a sequence of objectives since each achieved objective provides the mechanisms needed to solve the next one. That sequence is:

1. *Global consistency*. If the element being upgraded interacts with other components using some protocols, the updating actions should be applied without abort-

ing any started action. Besides this, if the protocol is modified, all involved parties should be appropriately upgraded to maintain consistent interactions among them.

2. *Service availability*. The overall service being upgraded should remain available in the updating interval. This means that it should be active and accepting requests. To this end, the upgrade can only start in a safe system state. Kramer and Magee [57] proved that *quiescence* is a sufficient condition for ensuring upgrade-safe system states. Quiescence is achieved when all active threads complete their executions and, if any new request arrives, it will be held in an input queue. With this, the code to be upgraded is not executed by any thread at this safe state.
3. *Coexistence and service continuity*. Coexistence consists in maintaining active both software versions: the one being replaced and the new one that replaces it. Service continuity implies that every service request will not be ever blocked. Coexistence is a necessary condition for achieving service continuity.
   In order to ensure service continuity a *dynamic version management* is suggested in [63]. With this, every system component is tagged with a version number. This is also applied to software connectors (i.e., communication channels with their associated communication protocols) and to the global system. Once a new component version is deployed, the global system version is increased. The old component version is still maintained. New service requests will be tagged with the new global system version. However, requests being serviced had been tagged with the previous number. The version of each request message is checked in order to decide which component may serve it. Once all old-numbered requests have been completed, the old component version is deallocated.
   Li assumes that the service being upgraded does not use replicated components. This imposes several constraints in the next goal.
4. *State transfer*. This is the most problematic aspect to be solved. The new software version may use a different state and such state will need a translation from its previous version to the new one. Besides this, even in case of maintaining both versions in the same host, some kind of copy might be needed.
   To solve this problem, Li [63] proposes the *state-sharing* principle: both software versions are deployed on the same host and they share their state. A wrapper provides the new interface and semantics to the new version. While the old version still remains active, mutual exclusion is used for avoiding simultaneous accesses from both component versions. Once all old-version ongoing requests have been completed, mutual exclusion is disabled. No actual state transfer is needed, since the state locations need not change in this kind of upgrade.
   More evolved solutions are described by Ajmani et al [2] where *simulation objects* behave as wrappers in the server domain. An underlying upgrading layer (UL) provides support for all software upgrading mechanisms. That layer embeds a simulation object per each software version being supported, although the current version does not need any wrapper and uses a plain server proxy. In this way, the server components being used at each time do not need to be aware of any system upgrading support. The actual state transfer happens at intervals with minimal workload relying on quiescence. At that step, the role of two UL components is also changed: the server proxy becomes a simulation object (for version V-1) and one simulation object (that of version V) becomes the current

server proxy. The software upgrading architecture proposed in [2] is also able to manage non-trivial public interface updates.
If replicated components were considered, the state transfer problem could be solved using the replica recovering mechanisms embedded in the replication support. Thus, old version replicas could be progressively replaced by new version replicas without any problems [104]. If the new software version requires any state transformation, such transformation should be applied before the first client request is forwarded to that joining new-version replica.
Unfortunately, state transformation heavily depends on the application logic. This means that no system may automate such translation without any programmer intervention. Even the solution proposed by Li [63] is unable to manage all possible situations. In the end, the programmer of the new version of the component being upgraded should write some kind of state-transformation functions for completing this task [96] and the PaaS system would need to provide a mechanism for running those functions before completing the state transfer. This explains why current providers are still unable to automate software upgrades ensuring service continuity: that mechanism is not provided yet.

5. *Minimal overhead*. There are multiple managerial tasks (to be executed by reconfiguration threads) related to software upgrading: new version deployment, old version removal, dynamic version management... Those tasks should be executed on multiple nodes and should be carefully scheduled in order to minimise their effect on system performance. To this end, Li distinguishes and evaluates three different kinds of schedulers:
   - *Competition scheduler*. It assigns the same priority to both reconfiguration threads and regular threads.
   - *Pre-emptive scheduler*. It assigns the lowest priority to reconfiguration threads. Thus, reconfiguration threads will only be executed when no regular threads are ready to run. This is the recommended approach when the workload at upgrading time does not saturate the servers being upgraded. It does not endanger SLA compliance in that case.
   - *Time-sliced (or controlled competition) scheduler*. CPU time is divided in time slots. Reconfiguration threads do only receive a predefined percentage of time; e.g., 20% or 50%. This is the best approach if workload already saturates the service capacity when software upgrading is started.

A performance evaluation is also shown in [63] with different upgrading strategies and schedulers, assuming that the main SLOs are response time and throughput. Its results provide these conclusions. When the upgrade is started with an already saturated service, the shortest upgrade time is obtained using a competition scheduler with state-share and dynamic version management. The latter implies version coexistence and service continuity. Unfortunately, the SLA is not respected in that case. SLA compliance is indeed achieved when an appropriate percentage of time (less than 20% in that example) is assigned to the reconfiguration threads, using the time-sliced scheduler with global consistency, state-share and dynamic version management. Note also that Li does not consider replication. So, he is analysing a hard-to-manage scenario where the set of available HW resources is static and limited.

Even in that scenario, his proposal is able to complete the software upgrade without violating the response time and throughput objectives initially settled.

With light workloads that do not saturate the available resources, the strategies that need to be used are the same (global consistency, state-share and dynamic version management), but the best scheduler is the pre-emptive one.

These results confirm that each mechanism enumerated in the list, above, progressively reduces the distance between the target SLOs and the achieved performance in the upgrading interval. Additionally, this verifies that an appropriate scheduler should be chosen depending on the current workload level: pre-emptive for unsaturated services and time-sliced for saturated ones.

Besides [63], Gey et al [37] also propose upgrading mechanisms that should ensure service continuity. To this end, they describe an upgrading strategy in stages for multi-tenant services. Since each tenant may require a different configuration for such service, the upgrade procedure deals with each tenant (or group of tenants with similar configurations) in a different stage. That strategy assumes the coexistence and service continuity mechanisms outlined in [63]. Therefore, each tenant may see how its needed set of software components is upgraded and reconfigured without compromising service continuity.

4.7 Summary

Table 1 provides a summary of the goals that define each one of the elasticity requirements described in this section, with a short description of the mechanisms being needed to achieve those goals and some references to the papers that have proposed, surveyed or used those mechanisms.

5 Open Problems

As it has been shown in previous sections, several solutions have been presented for each elasticity requirement in recent public cloud systems and academic proposals. However, more work is still needed in some of those areas. There are some open issues. They are discussed in Section 5.1. A first set of potential solutions is described later in Section 5.2.

5.1 Challenges

The following challenges remain open and will demand additional research:

1. *Relevance of virtual machine manager types in horizontal scaling mechanisms*. Although a virtual machine is the regular unit for managing infrastructure resources based on hypervisors, other lightweight supporting approaches [115] exist. These alternatives may accelerate some of the image management tasks [105], allowing some component sharing (e.g., the guest operating system in each deployed image might be equal to the host operating system, and be shared by all the

Table 1 Goals and mechanisms for each elasticity requirement.

Requirement	Goals	Mechanisms	References
Autonomy	Self-management: Services should be automatically administered without human intervention.	MAPE-K control cycle. Adequate platform architecture for managing that control cycle. Monitoring subservices and automated provisioning.	[19, 75, 90, 97, 101, 114]
Scalability	a) Services should be able to manage increasing workloads, using an increasing amount of computing resources. b) Acceptable QoS for all possible workload levels.	1) Replication (horizontal scalability) 2) Resizing (vertical scalability) 3) Migration (vertical scalability)	1) Every PaaS system. 2) [25, 39, 99] 3) [17, 55, 56, 97, 99]
Adaptability	a) Service components should be designed for accepting reconfigurations at runtime. b) Services should be able to deallocate their instances when workload decreases.	1) Proactive: SW performance engineering, performance modelling, workload modelling. 2) Reactive: thresholds on resource usage metrics or performance metrics, adaptability rules for setting those thresholds.	1) [6, 12, 19, 27, 51, 62, 75, 93, 97, 114] 2) [19, 75, 76, 84, 90, 100, 120]
SLA-awareness	Consideration of the SLA in all elasticity management decisions.	Management of multiple service level objectives in the SLA.	[19, 59, 63, 72, 73, 75, 93, 97, 101]
Composability	Avoid services implemented in a monolithic way.	Usage of modular architectures for service design. Consideration of inter-component dependences at deployment and scaling stages.	(Supported by most PaaS systems) [22, 29, 77]
Minimal service disruption in SW upgrades	Guarantee service availability, scalability and adaptability when the service code needs to be upgraded.	Version coexistence. Dynamic version management. State transfer through sharing. Specialised upgrading schedulers. Service replication. Staged upgrading.	[37, 63]

applications deployed on that host) while still guaranteeing isolation and security in the management of other resources.

Three basic types of virtual image managers exist [115]:

- *Hypervisors*. A hypervisor (or *virtual machine monitor*, VMM) [88] is a software layer that emulates all the underlying hardware, with the help of the virtualisation support from the CPU. With this kind of manager, each *virtual machine image* (VMI) includes a guest operating system and believes that it is the exclusive owner of such (virtual) computer. In a given host, each VMI may be running a different operating system.
  Some examples of hypervisors are: Microsoft Hyper-V, Oracle VirtualBox, VMware ESX Server, VMware Player, ...
- *Containers*. Instead of providing a full virtualised image for the hardware, a container provides a logical *operating system* (OS) interface. The images being deployed may have a thin layer that slightly complements the host OS, but

all images are compelled to share that underlying OS. This means that if, e.g., the host computer is a PC and its OS is a Linux distribution all guest images should use Linux as their OS. They cannot run any other OS (e.g., Windows) in their images. Isolation and security are achieved providing a private namespace for each kind of resource in the guest image. The images being used will be smaller than in a hypervisor approach, since both the OS kernel, basic commands and many libraries will be shared by all image instances being deployed in a given host.
Some containers are: Docker, LXC, Virtuozzo, OpenVZ, Linux-VServer, ...

- *Paravirtualisation* [11, 118]. This is a mix of the two previous approaches. A part of the hardware is virtualised and the remaining hardware pieces are managed by the host OS that provides some API to the images to be deployed there. This may reduce the size of the OS kernel image to be used in the VMIs being managed with this technique. To this end, those kernels should be slightly adapted. Thus, different VMs may use different OSs in the same host and their VMIs might be smaller than with hypervisors.
Some examples of paravirtualisation are: Xen [11], VMware Workstation, ...

There are several performance comparisons among some of the widely used hypervisors [107, 47, 113]. Those evaluations should be extended, considering lightweight containers, in order to delve in the pros and cons of each alternative. Such research work will yield the platform provider a better basis to decide which kind of infrastructure resources should be used in each deployment.

2. *Identification of the best predictive adaptability strategies for each application type.* Although several predictive approaches have been described in Section 4.3.1, it is still unclear which is the best strategy in that field. Proactive adaptivity requires performance prediction, workload prediction, or a combination of both approaches. In order to evaluate the quality of those mechanisms only an afterwards evaluation is possible, comparing their predictions with the obtained real data. Further research is needed in order to find the best strategies in this field or, at least, additional advice about which is the best tool for each kind of problem.
3. *Optimal amount of metric thresholds in multi-instance reactive rules.* Reactive adaptability is commonly based on a set of scaling decision rules. In each rule a threshold for a given metric (or group of metrics) is set. When that threshold is surpassed, a fixed amount of resource instances is added or removed. Therefore, when a given threshold is surpassed and the difference between the metric and its threshold is high, it seems convenient to set other thresholds in order to add or remove a greater amount of resource instances.
Frequent evaluation (e.g., every minute) may demand a considerable computing effort if the set of metrics to be considered is large. Besides, that frequent evaluation becomes more expensive for the customer. For instance, *basic monitoring* for Amazon EC2 instances provides, free of charge, seven metrics at five-minute frequency and three metrics at one-minute frequency. On the other hand, the *detailed monitoring* for EC2 instances monitors all those ten metrics at one-minute frequency but it demands an additional charge.
Thus, in some cases, longer monitoring intervals (e.g., 5 minutes, 10 minutes,...) will be a better (or, at least, more affordable) choice. The appropriate length of

that interval depends mainly on the type of application and on the variability of the workload that such application may introduce.
If a long monitoring interval is used and some metrics have varied a lot their value in the last interval, the PP should set an appropriate number of thresholds in order to drive its scale-out or scale-in actions. It should be analysed how many thresholds should be used in those cases.

4. *Critical component upgrading*. Component upgrading needs an underlying upgrade management subsystem in the platform. There are some proposals of subsystems of this kind showing a varying degree of automation (the newer, the better) [14, 15, 106, 1]. That subsystem is also software and it might have bugs or vulnerabilities that could lead to its upgrade. Unfortunately, such an upgrade could not be dynamic since there is no support for upgrading the upgrade management system itself. So, in the end, this might lead to a stop and restart upgrade of an important part of the platform, causing its temporal unavailability. Perhaps this could be solved using a minimal and safe upgrade management system that would never need any upgrade. Which are the principles to be followed in order to design that subsystem? Additional research is needed for answering that question.
5. *Software aging*. Elastic services deployed in a PaaS system should ensure their continuity and their QoS. Since software is not perfect, the *software aging*[3] [85] problem should be considered in cloud platforms. Software aging refers to the difficulties that arise when a piece of software runs continuously for a long time; some of its used resources might not be correctly released and, in the end, this causes some malfunctions: either performance degradation or software crashes.
Some performance losses in deployed services might be caused by software-aging errors and this may endanger the levels of performance being requested in the SLA. As a result, this may violate the fourth requirement stated in Section 3 (SLA awareness and compliance). This should be considered in the monitoring and performance analysis modules of a PaaS platform, reacting in an adequate way when that software aging problem is detected.
It is worth noting that software aging problems do not depend on how a PP has built its platform. The platform might be perfect, without any software error, but it should support the deployment of applications built by its customers. A PP may not forecast how such customer applications have been built. Some of them may have resource management errors that will lead to software aging problems. Because of this, some software aging-related monitoring and management seems to be convenient in current PaaS systems.

5.2 Potential Solutions

Let us present some initial work already done in the scope of each of those challenges, discussing some of their potential solutions.

[3] Parnas coined the *software aging* concept in [85] but, actually, he referred to the resource leaking problem as a "kidney failure" (*sic*) in [85], since he was comparing the software aging problems with the human aging ones and resource leaking was only a minor part of the problem being considered.

1. *Relevance of virtual machine manager types in horizontal scaling mechanisms*. Some performance comparisons between hypervisors and containers have been published recently [31, 79]. Let us discuss their results in order to counsel on further work.
   Felter et al [31] compare three approaches: native deployment, KVM hypervisor and Docker container, using different configurations for Docker. The service being deployed in those performance comparisons is the MySQL relational database management system. Three different Docker configurations were used. The most efficient Docker deployment added only a 2% overhead on the performance of a native deployment, while the KVM hypervisor introduced 40% overhead. That overhead is caused mainly by I/O operations. Therefore, it would have been minimised if the services being deployed could do a few long I/O actions instead of many short ones. That was not the case in the tested configuration for MySQL. Regarding CPU overhead, both hypervisors and containers are very efficient, with less than 1% overhead on average.
   Morabito et al [79] compare a Linux native deployment with the KVM hypervisor (with Linux as its guest OS), LXC and Docker containers and with the OSv [54] *library OS* on top of the KVM hypervisor. Library OSes like OSv are a minimal set of operating system modules that may run on top of a hypervisor, reducing to a minimum the overhead of the guest OS being executed on such hypervisor. OSv provides a (reduced) Linux system call interface and, thus, it is able to run some types of Linux applications.
   Multiple benchmarks are used in [79] in order to evaluate CPU-, network- and disk I/O-related overheads. The results show again that both containers and hypervisors introduce a negligible overhead in CPU-related benchmarks. Regarding disk I/O operations, containers introduce an overhead from 7% to 40% while hypervisors' overhead is between 50% and 93%, depending on the benchmark being used. Regarding network-related benchmarks, the overheads are in the 0% to 43% range for containers, 28% to 54% range for hypervisors and 26% to 47% range for OSv.
   These results suggest that containers have become an interesting option for deploying applications nowadays. However, it is worth noting that containers cannot provide the same level of security and isolation among deployed applications than hypervisors do. Moreover, there are not yet efficient containers for all possible OSes, since that technology has been developed at the moment for Linux distributions.
   Thus, PPs should start to consider supporting both hypervisors/paravirtualisers and containers in their provisions, choosing the best variant for each application to be deployed. If the application has not strong isolation requirements and may be run on a Linux OS, containers may be used on top of host Linux machines. On the other hand, when an application has strong isolation requirements or it demands a non-Linux OS, either hypervisors or paravirtualisation will be the best choice. Note that most PPs have traditionally used hypervisors or paravirtualisers as their virtual machine managers and that base should be still maintained. However, multiple library OSes, containers, hypervisors, and paravirtualisation approaches exist nowadays; therefore, a thorough comparison of all of

them should be periodically made in order to choose the best alternative in each branch, since some of these technologies (especially library OSes and containers) are still evolving and improving their performance and features.

2. *Identification of the best predictive adaptability strategies for each application type.* In order to provide a valid solution for this challenge an in-depth survey on predictive adaptability strategies is needed. A survey of this kind has been written by Lorido-Botran et al [66]. That paper identifies four main classes of predictive strategies:

   - *Reinforcement learning* (RL) [110]. RL automates the scaling task without using any a-priori knowledge or model of the application. Thus, RL learns the most suitable action for each particular application state following a trial-and-error approach, dynamically. This introduces the problem of needing long initial learning stages. As a result, the time needed for converging onto an optimal policy might be infeasibly long using this strategy. This explains why we have not discussed RL in Section 4.3.1.
   - *Queuing theory* (QT) [61]. QT sets a model of the application or system being considered. To this end, each request processing element is considered a server and its received requests are modelled with an incoming queue. QT defines rigid (i.e., static) models and this introduces its main drawback: each time the service is scaled, the QT model should be remade. In a similar way, workload variations demand that per each workload level to be considered, a new QT model configuration with a different arrival rate should be evaluated.
   - *Control theory* (CT) [119]. CT also defines a model of the application. In this case, that model is based on two different "variables": a *controlled variable* (or output) that should be maintained close to a desired *control point*, and a *manipulated variable* (or input) that may be adjusted at will and is the input to the modelled system. Using CT in our target scope means that the *controlled variable* should be a SLO (e.g., response time) while the *control point* will be the acceptable range for that SLO in the SLA, and the *manipulated variable* will be one of the system metrics (e.g., amount of VMs). A CT model needs to state the relationship between its manipulated and controlled variables. To this end, a *transfer function* is needed. The ARMA function presented in [84] is an example of this kind.
   - *Time series analysis* (TS) [86]. TS encompasses a large set of strategies that identify patterns in a set of data points. Those data points are collected, at a given frequency, either from the history of registered workloads or service performance metrics. The precision in the forecast values depends on selecting the best TS strategy and on choosing its correct parameters, mainly its history window and prediction interval.

   In spite of this, Lorido-Botran et al [66] have not been able to provide an answer to this challenge; i.e., they do not identify which is the best strategy for each possible type of application to be deployed in a PaaS system. However, the analysis of each proactive strategy provides some help to reach that goal.
   To begin with, RL may be discarded due to its too long learning phase.

QT may be used for services with a linear behaviour in their serving capacity (e.g., web services consisting of a few components that do not use persistent state), but will demand the adaptation and evaluation of its QT model on each scaling action. This requires a non-negligible computing effort for the PP in its MAPE analysis (evaluation of the QT model) and execution (QT model rebuilding) stages. The surveyed papers using this strategy seem to be adequate for considering the application *response time* as its main SLO. Therefore, simple interactive services may consider QT as a potentially valid predictive strategy. Despite this, the systems and services being evaluated in those surveyed papers were not yet in a production stage.

CT is a general strategy. Thus, there are multiple types of controllers (e.g., fixed gain, adaptive, model-predictive...) and multiple types of transfer functions (e.g., auto-regressive moving average, Kalman filters, smoothing splines, Gaussian process regression, fuzzy models...). As a result of this, there are many combinations of those two CT parameters and there has been no work looking for the most appropriate combination of them for each kind of elastic service.

TS is also a general strategy. There are two subclasses of strategies [66]: those focused on the direct prediction of future values (e.g., moving average, exponential smoothing, auto-regression order, machine-learning techniques...) and those identifying a repetitive pattern that may be used later for value forecasting (e.g., pattern matching, signal processing, auto-correlation...). As a result, we find the same problems than in the CT strategy: no study has been made yet for identifying possible *(TS variant, application class)* relationships.

3. *Optimal amount of metric thresholds in multi-instance reactive rules.* If there were multiple thresholds for the same reactive action, the scaling decisions would add or remove different amounts of resources when each threshold were surpassed. Following such strategy, the PP might use larger monitoring intervals, reducing the computing efforts for that monitoring stage.

   Casalicchio and Silvestri [19] have analysed the usage of long evaluation intervals (either 1 or 5 minutes) for reactive adaptive approaches. In that paper they evaluate the usage of one or two thresholds for driving the scaling decisions, as we have seen in Section 4.3.2. Using two thresholds, the service is able to ensure the lowest value in the response time SLO. However, with one threshold, the provider was able to minimise its costs. Additionally, the experiments carried out in [19] confirmed that the obtained results from each existing reactive approach depend a lot on the workload being considered. Therefore, there is no clear winner policy in this regard.

   Hasan et al [42] also proposes the usage of two upper (*thrU* and *thrbU*) and two lower (*throL* and *thrL*) thresholds. Two of these thresholds (*thrU* and *thrL*) define the upper and bottom boundaries, while the other two (*thrbU* and *throL*) define a more relaxed frontier. For instance, if we were considering CPU utilisation, the following values would make sense: *thrL*=20%, *throL*=30%, *thrbU*=70% and *thrU*=80%. The metric value is obtained periodically, and a set of rules is considered for taking scaling decisions. There is a duration interval (by default, 5 minutes) that is started each time the *thrL* or *thrU* thresholds are surpassed. Thus, in order to scale out, the metric value should have exceeded *thrU* initially, and

remain above *thrbU* at the end of the duration interval. Otherwise, no action is taken. In a similar way, a scale-in decision is taken when the metric value was lower than the *thrL* threshold initially and below *throL* at the end of the duration interval. But there are several other rules that consider all four thresholds and the previous duration intervals in order to either release a lower amount of instances per scale-in action or remain stable without scaling in or out. All these rules allow a consistent scaling behaviour in case of workload jitter and a more relaxed way of releasing resources when the workload decreases slowly. Besides this, these rules may be applied to groups of correlated metrics, requiring that all metrics in the group surpass their thresholds in order to start a scaling action.

Another possibility consists in using a single threshold of each kind (upper and lower), but being dynamic instead of static. An example of this technique, called *proportional thresholding*, was proposed by Lim et al [64] in 2009. It consists in adjusting the thresholds depending on the number of computing resources being used. For instance, in the use case described in [64] the metric being used is CPU utilisation and the lower thershold depends on the current number of assigned VMs. Thus, the lower threshold is initially set to 0% CPU usage when only 1 VM is assigned, being increased to 14% when there are 2 VMs and to 23% with 3 VMs (while, with static thresholds, a fixed value of 20% might be used in the same scenario). The concrete values to be used in that lower threshold depend on a previous analysis on the relationship between CPU utilisation, workload, and amount of VMs. Using a regression model, a function can be found for computing the expected increase or decrease on CPU utilisation when a VM is added or released at each workload level. That function determines the appropriate value for the thresholds depending on the current number of VMs assigned to the evaluated service.

A related question in this challenge is the optimal length of the monitoring intervals. Emeakaroha et al [30] analyse that problem when the POV-Ray application is deployed in the cloud. POV-Ray is a compute-intensive application for image rendering. Three different workloads are considered in [30] depending on their frame-rendering time demands: (a) variable, (b) increasing, and (c) constant and moderate. Six different monitoring intervals are considered: 5, 10, 20, 30, 60 and 120 seconds. Considering measurement costs and missing SLA violation detection costs, the optimal interval is 10 seconds for workloads (a) and (b), and 20 seconds for workload (c). In workload (c), all intervals greater than or equal to 10 seconds provide very low costs. Therefore, it seems that constant workloads may use any monitoring interval and, in those cases, a long interval will make sense. On the other hand, workloads (a) and (b) do not tolerate intervals longer than 10 seconds for the POV-Ray application, since those long intervals introduce many undetected SLA violations and those missed violations might introduce expensive penalties for both customer and provider since, in the end, they cause an unexpected behaviour in the deployed application.

Those results do not imply that the ideal monitoring interval will be 10 seconds in all cases. As we have seen, that optimal interval depends on the variability of the workload. Besides this, it also depends on the application type and on the concrete SLA being managed. Therefore, Emeakaroha et al [30] state that similar

comparisons are still demanded for other types of applications and each SLA being considered.

All these works provide a good basis for further research in this challenge. New results are demanded considering different monitoring intervals, as it has been suggested in [30]. Dynamic thresholds may be further studied considering other policies for varying those thresholds.

4. *Critical component upgrading*. No clear solution exists yet for this challenge. Let us describe a possible solution for the easiest case: an upgrade that fixes an implementation error in one of the components of the platform, but that does not introduce any modification in the platform interfaces. Since elastic services must be scalable and scalability demands replication, all deployed elastic service components will be replicated. The same will happen with most of the platform components: they are also replicated. This allows a progressive critical component upgrading; i.e., that those platform components are upgraded in stages, following a *rolling* principle [109].

   In a rolling upgrade, the set of replicas is divided in multiple small subsets. Each subset is upgraded in a different round. In this way, while a subset is being upgraded, all the remaining ones are active, serving their incoming requests. This ensures that service availability is not lost, although overall performance may be slightly reduced.

   When the component to be upgraded belongs to the platform, Potter and Nieh [89] suggest a solution based on migration. The components being run on top of that component must be temporarily migrated to other hosts. To this end, Potter and Nieh propose the POD (Process Domain) mechanism that is a type of light-weight container. PODs have been used for solving the problem of operating-system upgrades, assuming that the OS interface does not change in such upgrade. In that case, the application image is migrated to another host while the underlying OS is being upgraded. In this way, the host instance where the OS should be upgraded remains idle while the applications initially hosted by it have been migrated to other hosts.

   However, other types of upgrades –involving either interface modifications or complex state transformations between the software versions– are not easy to implement following the rolling upgrade principle or client process migrations. If those upgrades involve a critical component of the platform or one of the components of the software upgrading subsystem, further research will be needed for managing those scenarios.

5. *Software aging and software rejuvenation*. The *software rejuvenation* fix [36] is the common solution to the software aging problem.

   When a performance degradation is detected in the analysis stage of the MAPE control cycle, a rejuvenation action should be triggered. Rejuvenation consists in a restart of those component instances with degraded performance. Since multiple instances per component exist in the regular case and those instances are carefully monitored in a periodical way, it should be easy to detect when a software-aging error is happening (i.e., when with the same request arrival rate and the same assigned resources, the performance of a given server instance is decreasing). To this end, the CT and TS predictive approaches discussed above might be extended

in order to evaluate if any performance degradation is occurring at each moment, triggering a rejuvenation action when appropriate [23]. Stopping and restarting any faulty instance will not be a problem in this scope, since many component replicas exist and each of them might have been started and would show degradation at different times, but the machinery needed by these actions should exist and should be used correctly.

This general principle has been proposed and applied in several research papers related to cloud computing, but public cloud providers do not discuss software aging and rejuvenation in their documentation. Let us describe a few of these proposals in the sequel. A thorough discussion about general software aging and rejuvenation approaches has been provided by Cotroneo et al in [23].

Araujo et al [5] discusses the software aging problems detected in the 32-bit release of the Eucalyptus' *node controller* software. Eucalyptus is a framework for private IaaS management. Those software aging problems appeared in the memory management module of the node controller component. It released memory in an incorrect way. As a result of this, the host resident memory was eventually exhausted, even when all VMs deployed on it were terminated. Although this problem appeared at the IaaS layer, it may appear in any customer application being deployed on a PaaS system; e.g., Zhao et al [122] explain software aging detection and rejuvenation approaches for the Apache HTTP Server. Therefore, those management solutions are interesting for PPs in order to know how to deal with rejuvenation of those customer applications that demand continuous execution; e.g., web services.

Langner and Andrzejak [60] go a step further. Instead of relying on the performance monitoring machinery of the cloud platform in order to detect as soon as possible any performance degradation, they look for some metrics that are independent on the workload and that can be applied at the end of the software development stage, before deploying the application in its production environment. This proposal is based on the fact that software aging errors are regularly caused by coding faults. Thus, a history of the latest versions of the application or component to be studied must exist. Each of those versions is benchmarked considering multiple metrics and those results are compared in order to detect significant differences in any of them among that collection of software metrics. If any difference appears and it suggests an incorrect resource management, that version will be returned to the debugging stage in order to detect the coding error and fix it. This will prevent some software aging errors from appearing at the production stage since they will be diagnosed and fixed at the end of the development stage.

Matias Jr et al [68] improves the solution presented in [60], applying systematic statistical processing techniques for automated metrics comparison, in order to detect in a more reliable way software aging errors at the end of the development stage. Note that the approach described in [60] required a non-automated comparison.

Liu et al [65] assume that the PP supports VM migration as one of the ways to implement scale-up actions. Note that the VM migration mechanism demands an image checkpoint in order to start a migration. Those checkpoints may be con-

Table 2 Elasticity: challenges and potential solutions.

Challenge	Potential solutions	References
VM manager type to consider in horizontal scaling	1) Comparison of the overheads introduced by hypervisors, paravirtualisers and containers. 2) Refinement of that comparison per application type. 3) Revision of the previous solutions when new generations of containers and library OSes arise.	[31, 47, 79, 105, 107, 113, 115]
Identify the best predictive adaptability approaches	1) Analysis of the existing predictive strategies and the workloads supported by the main application types.	[66]
Optimal amount of metric thresholds in multi-instance reactive rules	1) Comparison of elasticity efficiency depending on the amount of thresholds. 2) Assessment of dynamic thresholds. 3) Assessment of different threshold evaluation intervals.	[19, 30, 42, 64]
Critical component upgrading	1) First try with rolling upgrades, assuming replicated services. 2) Combination with checkpointing and migration mechanisms.	[2, 14, 15, 89, 106, 109]
Software aging	1) Correct identification of software aging problems when there is any performance degradation. 2) Usage of software rejuvenation instead of scale-out actions in those cases. 3) Careful software aging diagnosis at the end of the development stage 4) Acceleration of rejuvenation actions with checkpointing/migration mechanisms.	[5, 23, 60, 65, 68]

served and taken as a basis for the rejuvenation procedure. Instead of restarting from scratch the instances that show software aging errors, a recent checkpoint is taken for accelerating those restarts. This proposal considers the regular performance evaluation mechanisms for detecting when the performance or resource usage metrics of a given instance signals an aging error. Besides, the migration machinery ensures that if the aging error was caused by a resource leaking problem, when a checkpoint is used for restarting the instance, a "clean" image is recovered.

These papers do not show a complete picture of the software aging research area (there are many others), but they already provide arguments in favour of considering some solutions to this problem in every cloud computing platform.

5.3 Summary

Table 2 provides a summary of the challenges presented in this section, with a short description of their potential solutions and some references to the papers that have proposed those solutions.

6 Conclusions

Elasticity can be defined as the autonomic management of service scalability and adaptivity when such service deals with a dynamic workload. Such kind of elasticity is a goal in all cloud computing service models (IaaS, PaaS and SaaS) but most of its inherent issues can be found in computing platforms, since they should deal with an autonomic management of customer applications, in all their control life cycle.

This paper has identified a set of elasticity-related requirements in PaaS systems. Autonomic management, scalability and adaptivity are inherent to the elasticity definition. SLA awareness, composability and minimal service disruption in upgrading intervals are three other requirements to be managed in these systems. There are multiple mechanisms that partially comply with each requirement. They have been described, providing some pointers to recent research results in each area.

Combining all those mechanisms and techniques, elastic PaaS systems may be built nowadays. In spite of this, current solutions can still be improved. We have identified several open problems, providing some hints to deal with them. From a general point of view, those open problems are related to adaptivity. In order to be adaptive, a provider should be able to manage multiple mechanisms for solving a particular problem, knowing which of those alternatives is the best in each scenario. So, existing solutions should be carefully analysed, identifying their pros and cons in each scope and finding other variants that minimise their implementation efforts and costs. Besides those comparison-related concerns, software upgrade management when QoS levels should be guaranteed is an open problem by itself. Some solutions have been proposed but they are not yet mature enough.

When all these aspects are considered, we realise that elasticity management in the PaaS service model is an active research area amenable to improvement.

Acknowledgements This work has been partially supported by EU FEDER and Spanish MINECO under research grant TIN2012-37719-C03-01.

References

1. Ajmani S (2004) Automatic software upgrades for distributed systems. PhD thesis, Dept Electr Engin and Comput Sc, Massachusetts Institute of Technology, USA
2. Ajmani S, Liskov B, Shrira L (2006) Modular software upgrades for distributed systems. In: 20^{th} Europ Conf on Object-Oriented Progr (ECOOP), Nantes, France, pp 452–476
3. Alhamad M, Dillon TS, Chang E (2010) Conceptual SLA framework for cloud computing. In: 4^{th} Intnl Conf Digital Ecosys Techn (DEST), Dubai, pp 606–610
4. Almeida S, Leitão J, Rodrigues LET (2013) ChainReaction: a causal+ consistent datastore based on chain replication. In: 8^{th} EuroSys Conf, Prague, Czech Republic, pp 85–98
5. Araujo J, Matos R, Maciel PRM, Matias R (2011) Software aging issues on the Eucalyptus cloud computing infrastructure. In: IEEE Intnl Conf Syst Man and Cybern (SMC), Anchorage, Alaska, USA, pp 1411–1416
6. Arief LB, Speirs NA (2000) A UML tool for an automatic generation of simulation programs. In: Wshop on Softw and Perf (WOSP), Ottawa, Canada, pp 71–76
7. Armbrust M, Fox A, Griffith R, Joseph AD, Katz RH, Konwinski A, Lee G, Patterson DA, Rabkin A, Stoica I, Zaharia M (2010) A view of cloud computing. Commun ACM 53(4):50–58
8. Bailis P, Ghodsi A (2013) Eventual consistency today: limitations, extensions, and beyond. Commun ACM 56(5):55–63

9. Bailis P, Ghodsi A, Hellerstein JM, Stoica I (2013) Bolt-on causal consistency. In: Intnl Conf Mgmnt Data (SIGMOD), New York, NY, USA, pp 761–772
10. Balsamo S, Marco AD, Inverardi P, Simeoni M (2004) Model-based performance prediction in software development: A survey. IEEE Trans Software Eng 30(5):295–310
11. Barham P, Dragovic B, Fraser K, Hand S, Harris TL, Ho A, Neugebauer R, Pratt I, Warfield A (2003) Xen and the art of virtualization. In: 19^{th} ACM Symp on Oper Syst Princ (SOSP), Bolton Landing, NY, USA, pp 164–177
12. Bennani MN, Menascé DA (2005) Resource allocation for autonomic data centers using analytic performance models. In: 2^{nd} Intnl Conf Auton Comput (ICAC), Seattle, WA, USA, pp 229–240
13. Birman KP (1996) Building Secure and Reliable Network Applications. Manning Publications Co., ISBN 1-884777-29-5
14. Bloom T (1983) Dynamic module replacement in a distributed programming system. PhD thesis, Dept Electr Engin and Comput Sc, Massachusetts Institute of Technology, USA
15. Bloom T, Day M (1993) Reconfiguration and module replacement in Argus: theory and practice. Software Eng J 8(2):102–108
16. Caballer M, Segrelles Quilis JD, Moltó G, Blanquer I (2015) A platform to deploy customized scientific virtual infrastructures on the cloud. Concurr Comp-Pract E 27(16):4318–4329
17. Calatrava A, Romero E, Moltó G, Caballer M, Alonso JM (2016) Self-managed cost-efficient virtual elastic clusters on hybrid cloud infrastructures. Future Generation Comp Syst 61:13–25
18. Calcavecchia NM, Caprarescu BA, Nitto ED, Dubois DJ, Petcu D (2012) DEPAS: a decentralized probabilistic algorithm for auto-scaling. Computing 94(8-10):701–730
19. Casalicchio E, Silvestri L (2013) Mechanisms for SLA provisioning in cloud-based service providers. Computer Networks 57(3):795–810
20. Casalicchio E, Menascé DA, Aldhalaan A (2013) Autonomic resource provisioning in cloud systems with availability goals. In: ACM Cloud Autonomic Comput Conf (CAC), Miami, FL, USA, pp 1–10
21. Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE (2008) Bigtable: A distributed storage system for structured data. ACM Trans Comput Syst 26(2)
22. Copil G, Trihinas D, Truong HL, Moldovan D, Pallis G, Dustdar S, Dikaiakos MD (2014) ADVISE - A framework for evaluating cloud service elasticity behavior. In: 12^{th} Intnl Conf Serv-Orient Comput (ICSOC), Paris, France, pp 275–290
23. Cotroneo D, Natella R, Pietrantuono R, Russo S (2014) A survey of software aging and rejuvenation studies. ACM J Emerg Technol 10(1):8:1–8:34
24. Coutinho EF, de Carvalho Sousa FR, Rego PAL, Gomes DG, de Souza JN (2015) Elasticity in cloud computing: a survey. Annales Telecommun 70(15):289–309
25. Dawoud W, Takouna I, Meinel C (2011) Elastic VM for cloud resources provisioning optimization. In: 1^{st} Intnl Conf Advances Comput Commun (ACC), Kochi, India, pp 431–445
26. de Juan-Marín R, Decker H, Armendáriz-Íñigo JE, Bernabéu-Aubán JM, Muñoz-EscoíFD (2015) Scalability approaches for causal multicast: A survey. Computing (in press)
27. de Miguel M, Lambolais T, Hannouz M, Betgé-Brezetz S, Piekarec S (2000) UML extensions for the specification and evaluation of latency constraints in architectural models. In: Wshop on Softw and Perf (WOSP), Ottawa, Canada, pp 83–88
28. Demers AJ, Greene DH, Hauser C, Irish W, Larson J, Shenker S, Sturgis HE, Swinehart DC, Terry DB (1987) Epidemic algorithms for replicated database maintenance. In: 6^{th} ACM Symp on Princ of Distrib Comput (PODC), Vancouver, Canada, pp 1–12
29. Dustdar S, Guo Y, Satzger B, Truong HL (2011) Principles of elastic processes. IEEE Internet Comput 15(5):66–71
30. Emeakaroha VC, Brandic I, Maurer M, Dustdar S (2013) Cloud resource provisioning and SLA enforcement via LoM2HiS framework. Concurr Comp-Pract E 25(10):1462–1481
31. Felter W, Ferreira A, Rajamony R, Rubio J (2015) An updated performance comparison of virtual machines and Linux containers. In: IEEE Intnl Symp Perf Anal Syst Soft (ISPASS), Philadelphia, PA, USA, pp 171–172
32. Fox A, Brewer EA (1999) Harvest, yield and scalable tolerant systems. In: 7^{th} Wshop Hot Topics Operat Syst, (HotOS), Rio Rico, Arizona, USA, pp 174–178
33. Galante G, De Bona LCE (2012) A survey on cloud computing elasticity. In: 5^{th} Intnl Conf on Utility and Cloud Comput (UCC), Chicago, IL, USA, pp 263–270
34. Galante G, De Bona LCE, Mury AR, Schulze B, Righi RR (2016) An analysis of public clouds elasticity in the execution of scientific applications: a survey. J Grid Comput 14(2):193–216

35. Gambi A, Hummer W, Truong HL, Dustdar S (2013) Testing elastic computing systems. IEEE Internet Comput 17(6):76–82
36. Garg S, van Moorsel APA, Vaidyanathan K, Trivedi KS (1998) A methodology for detection and estimation of software aging. In: 9^{th} Intnl Symp Softw Reliability Eng (ISSRE), Paderborn, Germany, pp 283–292
37. Gey F, Landuyt DV, Joosen W (2015) Middleware for customizable multi-staged dynamic upgrades of multi-tenant SaaS applications. In: 8^{th} IEEE/ACM Intnl Conf Utility Cloud Comput (UCC), Limassol, Cyprus, pp 102–111
38. Gilbert S, Lynch NA (2002) Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33(2):51–59
39. Gong Z, Gu X, Wilkes J (2010) PRESS: PRedictive Elastic reSource Scaling for cloud systems. In: 6^{th} Intnl Conf Netw Service Mngmnt (CNSM), Niagara Falls, Canada, pp 9–16
40. Grozev N, Buyya R (2014) Inter-cloud architectures and application brokering: taxonomy and survey. Softw, Pract Exper 44(3):369–390
41. Hammer M (2009) How to touch a running system. reconfiguration of stateful components. PhD thesis, Facultät für Mathematik, Informatik und Statistik, Ludwig-Maximilians-Universität München, Munich, Germany
42. Hasan MZ, Magana E, Clemm A, Tucker L, Gudreddi SLD (2012) Integrated and autonomic cloud resource scaling. In: IEEE Netw Operat and Mngmnt Symp (NOMS), Maui, HI, USA, pp 1327–1334
43. Herbst NR, Kounev S, Reussner R (2013) Elasticity in cloud computing: What it is, and what it is not. In: 10^{th} Intnl Conf on Auton Comput (ICAC), San Jose, CA, USA, pp 23–27
44. Hermanns H, Herzog U, Katoen J (2002) Process algebra for performance evaluation. Theor Comput Sci 274(1-2):43–87
45. Horn P (2001) Autonomic computing: IBM's perspective on the state of information technology. Tech. rep., IBM Press
46. Huebscher MC, McCann JA (2008) A survey of autonomic computing - degrees, models, and applications. ACM Comput Surv 40(3):7
47. Hwang J, Zeng S, Wu F, Wood T (2013) A component-based performance comparison of four hypervisors. In: IFIP/IEEE Intnl Symp Integr Netw Mgmnt (IM), Ghent, Belgium, pp 269–276
48. IBM (2006) An architectural blueprint for autonomic computing. White paper, 4^{th} ed.
49. Iosup A, Ostermann S, Yigitbasi N, Prodan R, Fahringer T, Epema DHJ (2011) Performance analysis of cloud computing services for many-tasks scientific computing. IEEE Trans Parallel Distrib Syst 22(6):931–945
50. Ivanovic D, Carro M, Hermenegildo MV (2013) A sharing-based approach to supporting adaptation in service compositions. Computing 95(6):453–492
51. Jiang Y, Perng C, Li T, Chang RN (2011) ASAP: A self-adaptive prediction system for instant cloud resource demand provisioning. In: 11^{th} Intnl Conf on Data Mining (ICDM), Vancouver, Canada, pp 1104–1109
52. Johnson PR, Thomas RH (1975) The maintenance of duplicate databases. RFC 677, Network Working Group, Internet Engineering Task Force
53. Kephart JO, Chess DM (2003) The vision of autonomic computing. IEEE Computer 36(1):41–50
54. Kiviti A, Laor D, Costa G, Enberg P, Har'El N, Marti D, Zolotarov V (2014) OSv - Optimizing the operating system for virtual machines. In: USENIX Annual Techn Conf (ATC), Philadelphia, PA, USA, pp 61–72
55. Knauth T, Fetzer C (2011) Scaling non-elastic applications using virtual machines. In: IEEE Intnl Conf on Cloud Comput (CLOUD), Washington, DC, USA, pp 468–475
56. Knauth T, Fetzer C (2014) DreamServer: Truly on-demand cloud services. In: Intnl Conf on Syst and Storage (SYSTOR), Haifa, Israel, pp 1–11
57. Kramer J, Magee J (1990) The evolving philosophers problem: Dynamic change management. IEEE Trans Software Eng 16(11):1293–1306
58. Lakshman A, Malik P (2010) Cassandra: a decentralized structured storage system. Operating Systems Review 44(2):35–40
59. Lang W, Shankar S, Patel JM, Kalhan A (2014) Towards multi-tenant performance SLOs. IEEE Trans Knowl Data Eng 26(6):1447–1463
60. Langner F, Andrzejak A (2013) Detecting software aging in a cloud computing framework by comparing development versions. In: IFIP/IEEE Intnl Symp Integr Netw Mgmnt (IM), Ghent, Belgium, pp 896–899

61. Lazowska ED, Zahorjan J, Graham GS, Sevcik KC (1984) Quantitative System Performance. Computer System Analysis using Queueing Network Models. Prentice Hall, iSBN 978-0-13-746975-8
62. Leitner P, Michlmayr A, Rosenberg F, Dustdar S (2010) Monitoring, prediction and prevention of SLA violations in composite services. In: IEEE Intnl Conf Web Services (ICWS), Miami, Florida, USA, pp 369–376
63. Li W (2011) Evaluating the impacts of dynamic reconfiguration on the QoS of running systems. J Syst Software 84(12):2123–2138
64. Lim HC, Babu S, Chase JS, Parekh SS (2009) Automated control in cloud computing: Challenges and opportunities. In: 1^{st} ACM Wshop Automated Control Datacenters Clouds (ACDC), Barcelona, Spain, pp 13–18
65. Liu J, Zhou J, Buyya R (2015) Software rejuvenation based fault tolerance scheme for cloud applications. In: 8^{th} IEEE Intnl Conf Cloud Comput (CLOUD), New York City, NY, USA, pp 1115–1118
66. Lorido-Botran T, Miguel-Alonso J, Lozano JA (2014) A review of auto-scaling techniques for elastic applications in cloud environments. J Grid Comput 12(4):559–592
67. Massie M, Li B, Nicholes B, Vuksan V, Alexander R, Buchbinder J, Costa F, Dean A, Josephsen D, Phaal P, Pocock D (2012) Monitoring with Ganglia. Tracking Dynamic Host and Application Metrics at Scale. O'Reilly Media, iSBN 978-1-4493-2970-9
68. Matias Jr R, Andrzejak A, Machida F, Elias D, Trivedi KS (2014) A systematic differential analysis for fast and robust detection of software aging. In: 33^{rd} IEEE Symp on Reliable Distr Syst (SRDS), Nara, Japan, pp 311–320
69. Medina V, García JM (2014) A survey of migration mechanisms of virtual machines. ACM Comput Surv 46(3):30
70. Mell P, Grance T (2011) The NIST definition of cloud computing. Recommendations of the National Institute of Standards and Technology, Special Publication 800-145
71. Menascé DA, Bennani MN (2006) Autonomic virtualized environments. In: Intnl Conf on Autonomic and Autonomous Syst (ICAS), Silicon Valley, California, USA, p 28
72. Menascé DA, Ngo P (2009) Understanding cloud computing: Experimentation and capacity planning. In: 35^{th} Intnl Comput Measurement Group Conf, Dallas, TX, USA
73. Menascé DA, Ruan H, Gomaa H (2007) QoS management in service-oriented architectures. Perform Eval 64(7-8):646–663
74. Miedes E, Muñoz-Escoí FD (2010) Dynamic switching of total-order broadcast protocols. In: Intnl Conf Paral Distrib Proces Tech Appl (PDPTA), Las Vegas, Nevada, USA, pp 457–463
75. Mohamed M (2014) Generic monitoring and reconfiguration for service-based applications in the cloud. PhD thesis, Université d'Evry-Val d'Essonne, France
76. Mohamed M, Amziani M, Belaïd D, Tata S, Melliti T (2015) An autonomic approach to manage elasticity of business processes in the cloud. Future Gener Comp Sys 50(C):49–61
77. Mohd Yusoh ZI (2013) Composite SaaS resource management in cloud computing using evolutionary computation. PhD thesis, Sc Eng Faculty, Queensland University of Technology, Brisbane, Australia
78. Montero RS, Moreno-Vozmediano R, Llorente IM (2011) An elasticity model for high throughput computing clusters. J Parallel Distrib Comput 71(6):750–757
79. Morabito R, Kjällman J, Komu M (2015) Hypervisors vs. lightweight virtualization: A performance comparison. In: IEEE Intnl Conf Cloud Engin (IC2E), Tempe, AZ, USA, pp 386–393
80. Najjar A, Serpaggi X, Gravier C, Boissier O (2014) Survey of elasticity management solutions in cloud computing. In: Mahmood Z (ed) Continued Rise of the Cloud: Advances and Trends in Cloud Computing, Springer, pp 235–263
81. Naskos A, Gounaris A, Sioutas S (2015) Cloud elasticity: A survey. In: 1^{st} Intnl Wshop Algorithmic Aspects of Cloud Comput (ALGOCLOUD), Patras, Greece, pp 151–167
82. Neamtiu I, Dumitras T (2011) Cloud software upgrades: Challenges and opportunities. In: IEEE Intnl Wshop Mainten Evol Service-Oriented Cloud-Based Syst (MESOCA), Williamsburg, VA, USA, pp 1–10
83. Neuman BC (1994) Scale in distributed systems. In: Casavant TL, Singhal M (eds) Readings in Distributed Computing Systems, IEEE-CS Press, pp 463–489
84. Padala P, Shin KG, Zhu X, Uysal M, Wang Z, Singhal S, Merchant A, Salem K (2007) Adaptive control of virtualized resources in utility computing environments. In: EuroSys Conf, Lisbon, Portugal, pp 289–302
85. Parnas DL (1994) Software aging. In: 16^{th} Intnl Conf Softw Eng (ICSE), Sorrento, Italy, pp 279–287

86. Parzen E (1960) A survey on time series analysis. Tech. rep., n. 37, Applied Mathematics and Statistics Laboratory, Stanford University, Stanford, CA, USA
87. Pascual-Miret L, González de Mendívil JR, Bernabéu-Aubán JM, Muñoz-Escoí FD (2015) Widening CAP consistency. Tech. rep., IUMTI-SIDI-2015/003, Univ. Politècnica de València, Valencia, Spain
88. Popek GJ, Goldberg RP (1974) Formal requirements for virtualizable third generation architectures. Commun ACM 17(7):412–421
89. Potter S, Nieh J (2005) AutoPod: Unscheduled system updates with zero data loss. In: 2^{nd} Intnl Conf on Auton Comput (ICAC), Seattle, WA, USA, pp 367–368
90. Rajagopalan S (2014) System support for elasticity and high availability. PhD thesis, The University of British Columbia, Vancouver, Canada
91. Reinecke P, Wolter K, van Moorsel APA (2010) Evaluating the adaptivity of computing systems. Perform Eval 67(8):676–693
92. Rolia JA, Sevcik KC (1995) The method of layers. IEEE Trans Software Eng 21(8):689–700
93. Roy N, Dubey A, Gokhale AS (2011) Efficient autoscaling in the cloud using predictive models for workload forecasting. In: 4^{th} IEEE Intnl Conf on Cloud Comput (CLOUD), Washington, DC, USA, pp 500–507
94. Ruiz-Fuertes MI, Muñoz-Escoí FD (2009) Performance evaluation of a metaprotocol for database replication adaptability. In: 28^{th} IEEE Symp on Reliable Distr Syst (SRDS), Niagara Falls, New York, USA, pp 32–38
95. Saito Y, Shapiro M (2005) Optimistic replication. ACM Comput Surv 37(1):42–81
96. Seifzadeh H, Abolhassani H, Moshkenani MS (2013) A survey of dynamic software updating. Journal of Software: Evolution and Process 25(5):535–568
97. Sharma U, Shenoy PJ, Sahu S, Shaikh A (2011) A cost-aware elasticity provisioning system for the cloud. In: Intnl Conf on Distr Comput Syst (ICDCS), Minneapolis, Minnesota, USA, pp 559–570
98. Shen M, Kshemkalyani AD, Hsu TY (2015) Causal consistency for geo-replicated cloud storage under partial replication. In: Intnl Paral Distrib Proces Symp (IPDPS) Wshop, Hyderabad, India, pp 509–518
99. Shen Z, Subbiah S, Gu X, Wilkes J (2011) CloudScale: elastic resource scaling for multi-tenant cloud systems. In: ACM Symp on Cloud Comput (SOCC), Cascais, Portugal, p 5
100. Simoes R, Kamienski CA (2014) Elasticity management in private and hybrid clouds. In: 7^{th} IEEE Intnl Conf on Cloud Comput (CLOUD), Anchorage, AK, USA, pp 793–800
101. Singh S, Chana I (2015) QoS-aware autonomic resource management in cloud computing: A systematic review. ACM Comput Surv 48(3):42:1–42:46
102. Smith CU (1980) The prediction and evaluation of the performance of software from extended design specifications. PhD thesis, Dept. of Comput. Sc., The University of Texas at Austin, USA
103. Smith CU, Williams LG (2003) Software performance engineering. In: Lavagno L, Martin G, Selic B (eds) UML for Real. Design of Embedded Real-Time Systems, Springer, chap 16, pp 343–365
104. Solarski M (2004) Dynamic upgrade of distributed software components. PhD thesis, Fakultät IV Elektronik und Informatik, Technischen Universität Berlin, Berlin, Germany
105. Soltesz S, Pötzl H, Fiuczynski ME, Bavier AC, Peterson LL (2007) Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In: EuroSys Conf, Lisbon, Portugal, pp 275–287
106. Soules CAN, Appavoo J, Hui K, Wisniewski RW, Silva DD, Ganger GR, Krieger O, Stumm M, Auslander MA, Ostrowski M, Rosenburg BS, Xenidis J (2003) System support for online reconfiguration. In: USENIX Annual Tech Conf, San Antonio, Texas, USA, pp 141–154
107. Sridharan S (2012) A performance comparison of hypervisors for cloud computing. Master Thesis (paper 269), School of Computing, University of North Florida, USA
108. Stonebraker M (1986) The case for shared nothing. IEEE Database Eng Bull 9(1):4–9
109. Sun D, Guimarans D, Fekete A, Gramoli V, Zhu L (2015) Multi-objective optimisation of rolling upgrade allowing for failures in clouds. In: 34^{th} IEEE Symp on Reliable Distr Syst (SRDS), Montreal, QC, Canada, pp 68–73
110. Sutton RS, Barto AG (1998) Reinforcement Learning: An Introduction. The MIT Press, Cambridge, Massachusetts, USA
111. Toosi AN, Calheiros RN, Buyya R (2014) Interconnected cloud computing environments: Challenges, taxonomy, and survey. ACM Comput Surv 47(1):7:1–7:47
112. Vaquero González LM, Rodero-Merino L, Cáceres J, Lindner MA (2009) A break in the clouds: towards a cloud definition. Computer Communication Review 39(1):50–55

113. Varrette S, Guzek M, Plugaru V, Besseron X, Bouvry P (2013) HPC performance and energy-efficiency of Xen, KVM and VMware hypervisors. In: 25^{th} Intnl Symp Comput Arch High Perf Comput (SBAC-PAD), Porto de Galinhas, Pernambuco, Brazil, pp 89–96
114. Vasic N, Novakovic DM, Miucin S, Kostic D, Bianchini R (2012) DejaVu: accelerating resource allocation in virtualized environments. In: 17^{th} Intnl Conf on Arch Support for Program Lang and Oper Syst (ASPLOS), London, UK, pp 423–436
115. Vaughan-Nichols SJ (2006) New approach to virtualization is a lightweight. IEEE Computer 39(11):12–14
116. Vogels W (2009) Eventually consistent. Commun ACM 52(1):40–44
117. Wada H, Suzuki J, Yamano Y, Oba K (2011) Evolutionary deployment optimization for service-oriented clouds. Softw, Pract Exper 41(5):469–493
118. Whitaker A, Cox RS, Shaw M, Gribble SD (2005) Rethinking the design of virtual machine monitors. IEEE Computer 38(5):57–62
119. Wishart DMG (1969) A survey of control theory. J R Stat Soc Ser A-G 132(3):293–319
120. Yataghene L, Amziani M, Ioualalen M, Tata S (2014) A queuing model for business processes elasticity evaluation. In: Intnl Wshop Adv Inform Syst Enterpr (IWAISE), Tunis, Tunisia, pp 22–28
121. Zawirski M, Preguiça N, Duarte S, Bieniusa A, Balegas V, Shapiro M (2015) Write fast, read in the past: Causal consistency for client-side applications. In: 16^{th} Intnl Middleware Conf (MIDDLEWARE), Vancouver, BC, Canada
122. Zhao J, Trivedi KS, Grottke M, Alonso J, Wang Y (2014) Ensuring the performance of Apache HTTP server affected by aging. IEEE Trans Dependable Sec Comput 11(2):130–141