# A Survey on Elasticity Management in the PaaS Service Model

Francesc D. Muñoz-Escoí, José M. Bernabeu-Aubán

Instituto Universitario Mixto Tecnológico de Informática
Universitat Politècnica de València
46022 Valencia (SPAIN)

{fmunyoz,josep}@iti.upv.es

Technical Report ITI-SIDI-2015/002

# A Survey on Elasticity Management in the PaaS Service Model

Francesc D. Muñoz-Escoí, José M. Bernabeu-Aubán

Instituto Universitario Mixto Tecnológico de Informática
Universitat Politècnica de València
46022 Valencia (SPAIN)

Technical Report ITI-SIDI-2015/002

e-mail: {fmunyoz,josep}@iti.upv.es

**Abstract**

Elasticity is one of the goals of cloud computing. An elastic system should be able to manage in an autonomic way its resources, minimising its costs and being adaptive to dynamic and variable workloads, scaling out when workload is increased and deallocating resources when workload decreases. PaaS providers should manage customer application reconfigurations with the aim of converting those applications into elastic services. This paper identifies the requirements that such a 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 each one of those concerns, providers' main goal is maximization 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 (or near-optimal) adaptivity strategies. Those open issues are also discussed, composing thus quite a complete image of what is elasticity management in PaaS systems.

KEYWORDS: Cloud Computing, Scalability, Adaptability, Elasticity, Service Level Agreement, PaaS, Workload Prediction, Reactive Management

## 1 Introduction

A requirement for a distributed service is availability: it should remain regularly open to be used by as many customers as needed with acceptable performance, and scalability is convenient in that case. Distributed systems and services have always been required to be scalable [45]. and replication of service components in order to balance the workload arriving to each replica has been the approach of choice. Services needed to be carefully designed and architected, minimising the synchronisation needs among their components with the aim of not blocking their execution. Evidently, unlimited scalability could not be achieved since it requires infinite infrastructure resources. So, traditionally, the scalability of a service depended on its software architecture being limited by the amount of hardware resources secured in the data centre where it was deployed.

The advent of cloud computing [5] partially broke those limits. IaaS providers may supply a large shared and flexible infrastructure for distributed service deployment. For most services, this is almost equivalent to an unlimited source of resources for deploying the distributed applications being needed by small and medium companies. However, those resources are not free of charge. So, infrastructure consumers should take care of minimising resource usage when the workload being managed by their applications is being processed. This introduces the need for an elastic management.

Elasticity management is not trivial. Ideally, managing actions should be taken by specialized companies: the PaaS providers. According to the NIST definitions [37], 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 gives freedom to the customer regarding application reconfiguration, since a customer should only deal with the general "*configuration settings for the application-hosting environment*" but does not need to worry about the mechanisms being needed for implementing those reconfigurations. Reconfiguration management is the responsibility of PaaS providers. Since every cloud service should be elastic [21], this means that PaaS providers should deal with many mechanisms that automate service scalability and adaptability.

However, the level of automation needed to approach a cost-optimal exploitation of a service is still challenging nowadays because of the many aspects that should be considered. On one hand, scalability decisions must match what has been stated in the *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 (and cannot be) 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 take appropriate actions. Those actions may consist in adding service instances or migrating those instances to better VMs, when the service capacity should be increased, or in the opposite case, in releasing instances when service capacity needs to be decreased.

One of the regular dimensions in SLAs is availability. Successful distributed services are concurrently used by many customers. Service providers should guarantee service continuity. Otherwise, service clients will not rely on that provider and they will look for another (more reliable) one. Unfortunately, since complex software systems are always in need of modification, both platform and service components need to be eventually upgraded [55] in order to fix bugs, remove security vulnerabilities or enhance their functionality. This software upgrading process might endanger the availability levels specified in a SLA. So, this is another source of trouble for service providers.

The goal of this document is to survey the current state of the art in the Cloud elasticity research area, particularly in how it impacts PaaS providers. To this end, this paper is structured as follows. Section 2 describes the provider requirements found in cloud computing systems that follow a PaaS service model. Section 3 explains the mechanisms being used in order to deal with such requirements. Section 4 discusses pending problems in this area. Section 5 presents some related work. Finally, Section 6 concludes this document.

## 2 Elasticity Requirements

Quoting Nikolas Herbst's definition [21], 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 is, basically, the combination of two complementary dimensions: scalability and adaptivity. 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 such workload decreases. Additionally, such resource management should be autonomous; i.e., driven by the system itself, without help from any human operator or administrator.

Considering this definition, some initial requirements may be identified:

*RQ1:* *Autonomy*. Elasticity management should be embedded in the service provisioning system. To this end, the principles of *autonomic computing* [23, 25, 27] might be followed. As a result of this, an autonomous manager should exist, with sensors and effectors on its managed elements; i.e., on the server instances. Thus, an autonomous manager should be able to *monitor* a managed element and its environment, *analyse* the collected metrics, *plan* its adapting actions and *execute* them, considering an appropriate *knowledge* base. This conforms the MAPE-K [25] reference control model that could be taken as a basis for building an autonomic platform.

*RQ2:* *Scalability*. The services being provided 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). There are multiple scaling mechanisms. They will be described in Sect. 3.

*RQ3:* *Adaptivity*. The amount of resources being used should be adapted to the current demands, avoiding overcommitting unnecessary resources.

Resource overprovisioning does not compromise the achievable service performance, but incurs in excessive costs, since those resources must be paid, thus compromising profitability. On the other hand, resource underprovisioning is equally bad, and, depending on the terms of the SLA, potentially worse, since the incoming workload will overwhelm the existing servers ruining the QoS compromised by the SLA, with explicit economic consequences, or implicit punishment by customers abandoning the service.

Those are the general requirements to be considered in any kind of elastic service. This paper is focused on the PaaS cloud computing service model. Other specific requirements arise in this case. They are the following:

*RQ4:* *SLA-awareness*. SaaS providers develop and integrate the software of their service, and expect their PaaS providers to automate their deployment, reconfiguration and management. Such automation must be necessarily based on some *service level objectives* 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 within their SLAs, 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.

*RQ5:* *Composability*. A distributed service may consist of multiple modules or components. Those components wait for some inputs and provide some outputs. So, they are related by a flow of data and processing. The scaling unit of action will be, in most cases, the various components forming a service application. Thus, if only one of those components has become the current performance bottleneck, we might scale out only that component in a first stage. This means that we should focus on each component at the time of taking elasticity actions. In spite of this, we should still consider that those components are inter-related, carefully analysing which are the behaviour dependencies among them.

*RQ6:* *Service continuity in software upgrades*. Distributed services should guarantee their service continuity. This means that clients expect a continuous availability. Real world software systems are never static. Clearly, software defects will drive change. Additionally, market pressures and customer demands for new functionality also drive change, forcing service providers to continuously come up with newer versions for the software that runs their services. A careful software upgrading procedure should be designed [34] taking into consideration the terms of the service's SLA, which in many cases will require some degree of service continuity and availability.

## 3 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. 2.

## 3.1 Dealing with Autonomy

The platform being provided in a PaaS service model should deal itself (ideally, without any human intervention) with the elastic behaviour of the services deployed by its customers. This means that such platform should be a piece of *autonomic software* [27].

In order to have an autonomic behaviour, a software system should be carefully architected. In the IBM blueprint for autonomic computing [25] a software system is considered autonomic when it is able to self-manage with a minimum of human interference. In that case, such 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 where four stages are identified:

1. *Monitor*. In this stage 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 from the monitor 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 such 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 execute stage applies those actions to the managed elements through the corresponding effectors.

Each stage is continuously being applied and its output is considered by the next one in that control loop. To this end, the stages may access a shared *knowledge base* where the global policies can be found and the interesting information being generated in each stage is stored. In most cases, such logged data needs to be considered in the *analyse* or *plan* stages.

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 [25] become now a combination of:

- A general economic goal: to minimize 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.

Multiple academic proposals (e.g., [13, 42, 49, 56, 66]) follow an architecture that complies with many of the recommendations from the IBM's 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.

However, when cloud elasticity is considered, autonomy only refers to the ability of automatically adapting the service being provided to its current workload and its agreed QoS levels (SLA). This means that the autonomy being looked for in this context is a rather limited subset of what is being understood as general autonomic computing. Indeed, such limited degree of autonomy is being provided by the internal platform components that are managing the application components installed by the PaaS customers. So, many autonomic computing aspects (e.g., self-protection [25], self-managing distributed architectures [69, 31], ...) are not key features in this scope. The interested reader may refer to [24] in order to delve deeper in the autonomic computing research field.

## 3.2 Managing Scalability

In order to have an elastic system, the software services it provides should 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 PaaS provider has access to an underlying computing infrastructure (in some cases, rented to external IaaS providers) and that it may use as many computing nodes as needed.

There are different scalability mechanisms [19]:

- *Replication* (i.e., *horizontal scalability*). Horizontal scalability consists in using additional computing nodes for executing additional server instances when such serving capacity needs to be increased. This may be implemented adding new VMs to the current set, deploying there some instances of the required service components.

  Replication is a mechanism traditionally used [3] for ensuring service availability. Two classical replication models exist: *state-machine* (or active) *replication* [54], where client requests are forwarded to all server replicas and every replica executes directly each request, and *primary-backup* (or passive) *replication* [11], where client requests are sent to a special replica (the primary one) that serves each request and forwards any state updates to the remaining replicas (the backup ones).

  Both approaches try to ensure a strong consistency among all involved replicas [20]. To this end, both require long synchronisation delays in their communication steps. Active replication demands a reliable total order broadcast to deliver client requests. On the other hand, passive replication only requires direct communication between clients and the primary replica, but a reliable FIFO broadcast is needed to deliver update checkpoints onto backup replicas. Those communication delays might compromise scalability.

  In order to increase scalability, inter-replica consistency should be relaxed. To this end, *eventual consistency* is needed. Up to our knowledge, the first mentions of eventual consistency in distributed services were given in the Clearing house (a replicated directory) service at Xerox and published in 1987 [17]. Eventual consistency is based on a lazy propagation of updates from the replica that has served each request. Different clients may forward their requests to different replicas. Eventually all those updates are delivered and all replicas converge to the same state. In order to avoid update-ordering conflicts, such lazily propagated updates may be expressed as commutative operations [65].

  Another alternative to avoid consistency problems is to delegate state sharing. If there is a need of sharing data, such data may be maintained by a specialised external component. An example is a scalable datastore (e.g., Google Bigtable [14], Apache Cassandra [32], Microsoft Azure SQL Database,...), that transparently uses replication and *sharding* [64] in order to ensure a fast management of updates without compromising scalability. Another alternative is the usage of a distributed file system from the IaaS or PaaS provider (e.g., Amazon S3, Google GFS,...). Those file systems have embedded replication at a lower level (disk blocks or other encompassing storage units).

- *Redimensioning* (i.e., *node capacity upgrades*). Vertical scalability consists in improving the capacity of the serving nodes. A way for implementing vertical scalability is node upgrading. This is supported when virtual machines are used and those virtual machines get additional resources from their hosting physical computers; e.g., a larger share of CPU time [38], a larger amount of physical memory, a higher bandwidth in their access to the network, etc.

  Unfortunately, many IaaS providers do not allow VM redimensioning at run-time. Note that VM redimensioning requires that either the host computer has free resources at redimensioning time or the other guest VMs should also adapt their provisioned set of resources to accommodate such redimensioning. Instead of those dynamic redimensioning actions, most IaaS providers facilitate a static set of VM types and the consumer is compelled to select which kind of VM should be used in each image allocation.

- *Migration*. When the underlying infrastructure does not allow VM redimensioning, vertical scalability may be implemented migrating the server instances from their current VM to another more powerful one in a different host computer.

Depending on the size of the instance image, the network bandwidth and the location of the host computers, migration may need a *long* time to be concluded (several seconds). Note that a migration is a three-step procedure:

1. The current instance image should be stopped. Such stop action should be initiated when such instance is not serving any request; i.e., when it remains suspended waiting for a new request.
2. Its image should be transferred to the new container.
3. Its activity should be restarted there. If the instance identity and network address should be preserved, some routing reconfiguration will be needed in this step.

Since every distributed service should be fault-tolerant, we could also assume that service components could be replicated, even when migration is the main scalability mechanism. Considering that replication management needs some replica recovery (i.e., state transfer) mechanism, the migration procedure may be slightly modified, following this alternative sequence:

1. A new service instance is deployed in a given target host or container. At this point, it is considered as a "new replica" that demands state recovery.
2. Its state is transferred from the original instance to that new container, using to this end the regular replica-recovery mechanisms.
3. The old instance is stopped once all its received client requests have been served and the new instance has been activated. This ensures service continuity.

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

Sharma *et al.* [56] describe the Kingfisher system where all scaling mechanisms (replication, resizing and migration) can be used and combined in order to improve service performance. Kingfisher is able to federate private and public infrastructures. VM redimensioning may be implemented in private infrastructures. OpenNebula is used in such a case. On the other hand, VM migration is the mechanism needed in public IaaSes and also when VMs should be moved between different IaaS providers. The goal in [56] is to propose an efficient resource provisioning algorithm. The best results are obtained combining two kinds of algorithms. The first one is centred in minimising the amount and cost of infrastructure resources. The second one minimises the provisioning latency, considering all scalability approaches: VM replication, VM resizing and VM migration. A third algorithm, based on integer linear programming, is needed for combining all previous algorithms.

Knauth and Fetzer [28] 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.

In a second paper [29], 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 techniques. Indeed, it could be considered as a hybrid between horizontal and vertical scalability, since a VM stop is a scale-in action and a VM restart is a scale-out action. VM restarting actions can be completed in less than a second in this case. To this end, VM restarting is implemented in a lazy way, 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 support provided by the host.

CloudScale [57] uses a workload predictor in order to adjust the resources being assigned to each VM in a given host. So, it is able to implement vertical scalability using the VM redimensioning technique described above. However, when the overall forecast workload in a given host exceeds its capacity, VM migration is also used in order to comply with the infrastructure 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 action.

Casalicchio *et al.* [12] analyse the negative impact that VM migration may have on an IaaS provider when VM allocations have exhausted the available resources and VMs need to be migrated to external infrastructure providers. 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.

Finally, Medina and García [36] provide a complete survey on general VM migration techniques, without limiting their scope to vertical scalability in cloud systems. Two main approaches can be distinguished in that scope: *live migration*, where both the process activity and the communication resources and current connections are migrated, and *suspend/resume migration*, where communication resources and connections are dropped and need to be restarted (with new addresses and identities) once the migration has been concluded.

In any case, the temporal costs of those scalability actions depend on the image container being used. Different alternatives exist. In the regular and more general case, virtual machines are considered, but other alternatives exist with a lighter weight (e.g., LXC or c-groups in Linux-based hosts for Linux-based images. There are similar mechanisms within other OSes), allowing faster migrations or resource reallocations.

## 3.3 Being Adaptive

Adaptivity has been considered as a light version of autonomic computing in some works [24]. In spite of this, in the elasticity literature [21], we refer to adaptivity as the possibility of "optimally" adjusting the computing capacity of a given service to the current workload, at run-time. So, besides being scalable, an elastic service should also be adjustable to the workload being received at each moment.

In [50] two different dimensions of "being adaptive" are considered:

- *Adaptability*. It refers to the potential for adapting a system. This means that the system components have been designed considering that sooner or later 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 [41] and Ruiz-Fuertes [53] propose adaptable metaprotocols 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 reconfiguration actions has any immediate effect on the number of system nodes being needed.

- *Adaptivity*. This goes a step further. Adaptivity is the system's 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 3.1), 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 kind of workload analysis or workload modelling is done in order to forecast the workload 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, quality of service might be easily guaranteed when those predictions have an acceptable level of accuracy. 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.

So, proactive systems predict the forthcoming workload and apply any needed reconfigurations beforehand, while reactive systems do not use any predictive approach. Instead, they monitor the current real workload and reconfigure themselves when that current workload reaches a level that compromises their quality of service.

Let us discuss in the sequel some of the existing solutions in each one of these two types of adaptivity approaches.

### 3.3.1 Proactive Mechanisms

Software performance prediction has become an interesting topic in the cloud computing field in the last years. *Software performance engineering* (SPE) [59], one of the existing performance prediction approaches, is a research area that has received attention since the 80s [60]. SPE states that performance is one of the most important goals of reliable software products and that adequate levels of 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 such an 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.* [6] surveyed the existing techniques in the performance modelling and prediction field. Such SPE techniques should be applied at the design stage, considering the proposed software architecture. At that 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.

Such scenario becomes even worse when other aspects are considered. In this scope, PaaS providers try to use proactive mechanisms for adapting customer software to their forecast workloads. To this end, the behavioural model that should be taken as the input for those proactive mechanisms need to be provided by the PaaS customers (i.e., the designers and developers of the distributed applications to be deployed onto the platform). This would compel PaaS providers to recommend a set of design tools to their customers, and those customers would be forced to use such tools in their software design stages. Many software companies are not yet convinced about the need to use such tools and about the benefits those tools provide. So, those behavioural models would become non-existent in most cases.

Most of the solutions surveyed in [6] were based on UML diagrams as their behavioural model and generated some variant of queuing network 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 provide a compoundable performance model and composability is a regular requirement when a distributed application is being considered. 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. On the other hand, Petri nets and semi-Markov processes were seldom used.

Generally, the approaches based on stochastic process algebras [22] 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 [4, 16] 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 [16] or with additional input requested by the converter [4].

All those performance prediction techniques have been conceived for their consideration and 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 designers or architects of those applications; i.e., this is a help for the platform customers. On the other hand, PaaS providers may also use those approaches in order to predict the performance of the underlying platform components. However, PaaS providers should also consider an additional concern: how to predict the performance of the software components being deployed on those platforms. As it refers to this latter concern, the PaaS provider receives a set of components to be deployed. Those components have already been designed and developed, by other people. That provider 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 adequate performance levels that match what has been stated in their SLA. In the end, this might be a very different problem, but this will depend on the information being provided for deploying services.

Considering a given service, its deployment descriptor should provide information about each one of the service components and their dependences. In some cases, it may also report how many instances of each component should be initially deployed. Thus, the *traditional* performance prediction approaches described up to now could be useful for filling that information. So, that initial deployment will not be a problem for the PaaS provider since that requested information will be provided by the PaaS customer. Once the service is deployed, the platform should consider what has been specified in the SLA in order to drive its adaptivity.

In this second phase (i.e., once the customer application has already passed its initial deployment), we may still rely on queuing network models in order to identify soon the potential bottlenecks, given the current service workload. But, once that service has been running for a while, we could analyse the existing history of workload levels in order to forecast the forthcoming workload trends, trying to adjust beforehand the amount of resources demanded by those deployed applications. So, performance predictors could be combined with workload predictors.

Some of the solutions being used in different academic proposals are discussed hereafter.

To begin with, *Bennani et al.* [8] 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, two kinds of controllers are used: local and global. There are multiple *local controllers*, one for each deployed application. 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 from that database 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 (either real or virtual; the latter is provided by the global controller in some cases, see next paragraph for details) 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, on the other hand, 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.

As we can see, 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 also 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 existing utility function. Besides this, a provider needs to evaluate all the resources being used or demanded by all existing 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 being considered and on the existing environment, the workload being supported may show variability trends that could be hard to forecast.

*Casalicchio and Silvestri* [13] 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.

Although the ASAP [26] subsystem only deals with VM provisioning and its SLO is to minimise the VM provisioning time (i.e., that of an IaaS provider), it shows an example of advanced predictive techniques. The main contribution of this proposal is to manage and combine 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 [6], Petri nets were identified as a valid tool for performance modelling, but only used in a few systems. *Mohamed* [42] 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 scaling out or scaling 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 later condition the reactive actions.

*Roy et al.* [52] 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 (i.e., one instance per iteration) 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 the involved services. The experimental results presented in the paper show that the combination of a performance model and a workload predictor provides optimal results for managing an adaptive resource allocator, as it was already shown in [8].

Kingfisher [56] is a platform that combines a workload analyser with a compound capacity planner. Such a capacity planner uses three different strategies for managing the platform resources:

- Infrastructure cost-aware provisioning. Given the estimated peak workload $\lambda_1$, $\lambda_2$... $\lambda_k$ 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 cost-aware 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 $\lambda_i$.
- 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, an integer linear programming algorithm that mixes both of the two previous strategies is proposed.

DejaVu [66] may be used as the Kingfisher's workload analyser. All client submitted 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 constantly 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.

### 3.3.2 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 a set of rules is generally static, but the thresholds being used may be updated at run-time. The input being considered by those rules is the current set of measurements taken by the monitoring subsystem being used in the platform. 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. [42, 43, 46, 49, 58, 70].

Let us describe some examples that present any evolution on this basic description of reactive mechanisms.

As it has been presented above, *Casalicchio and Silvestri* [13] compared multiple adaptive strategies. Only one of them is predictive. The other four are strictly reactive. Those reactive policies are based on two different metrics: CPU utilisation and response latency. On each metric, two different threshold strategies are considered: to use one or two thresholds for each kind of decision. Let us describe two of those policies in order to explain the threshold usage. In both cases, the metric being considered is CPU utilisation and it is monitored once per minute (or, instead, every 5 minutes). Those policies are:

- *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. On the other hand, when the CPU utilisation is below 50%, one of the existing instances is removed.

- *UT-2al*: It uses two thresholds for each kind of decision. Thus, when the CPU utilisation is greater than 62% a new instance is added, but if it is greater than 70% two instances are added. On the other hand, when the CPU utilisation is below 50%, one of the existing instances is removed, but when it is lower than 25% two instances are deallocated.

The performance results from [13] 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).

Such 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 scaling in actions (i.e., for deallocating an existing instance) we should also consider the length of the billing interval being used by the underlying IaaS provider, in case of using a public IaaS. The common length is one hour. So, the PaaS provider 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 chosen and tagged for being deallocated afterwards. Such mark will be removed if the workload is increased before stopping that selected instance.

At a glance, when such metric evaluation intervals are long, it makes sense to use multiple thresholds. Thus, if the workload has varied a lot from the previous evaluation, the platform may add or stop multiple server instances in a single reactive action. We have only found one paper managing multiple thresholds per metric [13], and its results have been partially explained above. With a metric evaluation interval of five minutes, the response time (called request latency in that paper) metric is the best for minimising overall provisioning costs and for optimising the benchmark completion time. To obtain those results, only a threshold was used. With two thresholds the results are slightly worse.

On the other hand, when the metric being considered is CPU utilisation, the usage of two thresholds provides better results than using only one. But in all cases, CPU utilisation seems to be a metric worse than response time in order to optimise the SLOs being considered in that paper (overall provisioning cost and benchmark completion time). This is reasonable since there is no direct relationship between CPU utilisation and the SLOs defined in that scenario. Thus, no clear conclusion can be extracted about the optimal number of metric thresholds to be set in reactive strategies.

## 3.4 Satisfying SLAs

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 [23] 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. The other goals are related to its ”quality of business” (QoBiz) [44]; 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, cloud providers should manage SLAs with a rich set of objectives. Thus PaaS customers 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 [13, 33, 34, 40, 39, 42, 52, 56] (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 PaaS providers is to elastically manage the services being deployed by multiple customers providing and guaranteeing a rich set of service level objectives in their SLAs.

## 3.5 Managing Compound Services

As it has been explained in Section 3.2, the services being deployed in a PaaS system should be automatically scaled by the PaaS. 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 the single element that implements that service. This would mean a lack of flexibility when we try to improve such service performance, since we could only add, remove, redimension or migrate instances of that single kind of element. Moreover, that single element would be larger than the regular elements of a multi-component service and this would have complicated any scaling actions to be applied onto it. If the service is implemented by multiple small components, their scaling actions will be faster and cheaper, since the amount of resources needed to manage them will be also smaller.

Distributed services, as suggested in [51], consist of multiple components that may be architected in layers. Such composability makes possible 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 (i.e., when they become performance bottlenecks), 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. So, scale-in and scale-out actions can be easily and quickly applied to these components. Being replicated, software upgrades also become easier than in a non-replicated architecture, since the new software versions (when component interfaces are maintained) could be applied replica by replica without endangering service continuity [55].

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.

Finally, careless composability may also complicate the definition of the behavioural and performance models being needed in the proactive adaptivity approaches summarised in Section 3.3.1. 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 respective SLAs of each component. When those dependences have been appropriately 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.

## 3.6 Software Upgrades

Software needs to be updated due to multiple causes that have been already outlined in Section 2. This upgrading process has another constraint for cloud providers: they should take care of what has been stated in the SLAs. A possible set of principles and mechanisms to consider in the software upgrade stage taking into account SLAs has been described by Wei Li in [34]. Let us describe that proposal.

According to Li [34] when some software components (and this may be applied to both the applications deployed onto the platform and to the components that belong to the platform itself) need to be upgraded respecting some SLA, the upgrading system should take care of a set of objectives. Those objectives should be accomplished in a specific sequence since each achieved objective provides the mechanisms needed to solve the next one; i.e., it does not make sense to accomplish objective $i$ when objective $i-1$ is still pending. If we want to guarantee a good level of quality of service in a software upgrade, all of the objectives in that sequence should be adequately dealt with. That goal sequence is the following:

1. *Global consistency*. If the element being upgraded interacts with other components using some protocols, the updating actions should be applied without aborting 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 proved in [30] that *quiescence* is a sufficient condition for ensuring upgrade-safe system states. Quiescence is achieved when all active threads have completed their operation executions and if any new operation request arrives it is held in an input queue. With this, the code to be upgraded is not executed by any thread or process at this safe state.

   In spite of guaranteeing service availability, quiescence is not the best approach to follow since it blocks service activity. So other complementary mechanisms are needed for improving availability when the next goals in this list are considered.

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 being received 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 [34]. 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.

   The detection of when all old-numbered requests are completed is managed by an admission control module that assumes that each request always generates a reply. It filters the reply messages in order to know when the old software version can be removed.

   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 physical transference or copy might be needed.

   In order to partially solve this problem, Li [34] proposes the *state-sharing* principle. In that case, both the old and the new version are deployed on the same host and they share their state. A wrapper provides the new interface and the new semantics to the new version. While the old version still remains active, a mutual exclusion primitive is used for avoiding simultaneous accesses from both component versions. Once all ongoing requests with the old version have been completed, such mutual exclusion tool can be disabled. No actual state transfer is needed in this case, 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. Such layer embeds a simulation object per each software version being supported, although the current version does not need any wrapping interference 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. 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.

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. This means that 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 presented in [34], using different upgrading strategies and schedulers, assuming that the main SLOs are response time and throughput. Its results provide the following 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 the example being considered in that paper) 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 in his evaluations. 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. However, in this case a second strategy may be used for achieving the shortest upgrading time: one that considers only the global consistency and service availability aspects. This means that the upgrading strategy should be based only on quiescence. Dynamic version management and state sharing do not shorten the upgrading time in this second scenario. But this is not for free: the response time being obtained with quiescence is noticeably longer than when all the other upgrade management mechanisms are combined.

These results confirm that each one of the mechanisms 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.

## 4 Open Problems

As it has been shown in the previous sections, several preliminary solutions have been presented for each elasticity requirement in recent academic proposals. However, further work is still needed in those areas. There are some open issues that need to be mentioned. Let us describe them:

- 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 [67] exist. These alternatives may accelerate some of the image management tasks [61], 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 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 [67]:

  - *Hypervisors*. A hypervisor (or *virtual machine monitor*, VMM) 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.
  - *Containers*. Instead of providing a full virtualised image for the hardware, a container provides a logical 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.
  - *Paravirtualisation* [7, 68]. 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.

  A container-based approach may reduce the sizes of the component images being managed for deploying a given customer application. This might accelerate several image life cycle actions: image cloning, image start, image migration...

  There are several performance comparisons among some of the widely used hypervisors (e.g., [63]), but there are not similar works on light-weight containers. A careful evaluation of these light-weight approaches is interesting, and a benchmarking comparison of those containers with paravirtualisation approaches and current hypervisors, too, in order to delve in the pros and cons of each alternative. Such research work will provide to the platform provider a better basis to decide which kind of infrastructure resources should be used in each application deployment.

- Analysis of the best migration mechanisms when the infrastructure being used does not admit image redimensioning. Some papers (e.g., [28]) have presented different mechanisms for optimising the VM migration time in cloud infrastructures. This is particularly interesting for applications that have not been designed considering scaling out approaches and that do not easily tolerate that their components were replicated (e.g., stateful services).

  These migration mechanisms could be surveyed in order to find the best approaches. This issue has been partially fixed in [36] but that survey does not describe some possible optimisations (e.g., those described in [28]) nor their applicability to different kinds of light-weight containers. These aspects need to be analysed and explained in forthcoming papers.

- Although many proactive adaptivity mechanisms have been described in Section 3.3.1, it is still unclear which is the best strategy in that field. Proactive adaptivity requires a 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.

- In case of using long metric evaluation intervals, how many thresholds should be used for taking multi-instance scaling actions?

  *Casalicchio and Silvestri* [13] have already analysed the usage of long evaluation intervals for reactive adaptive approaches. In that paper they evaluate the usage of one or two thresholds for driving the scaling out or scaling in decisions, as we have seen in Section 3.3.2. Surprisingly, a single threshold was enough in many cases, but it seems convenient to further investigate in this research line, specially in case of highly variant workloads.

- The rules to be applied in order to derive from the SLOs a set of appropriate metrics to be evaluated in reactive adaptive approaches are still unclear. Although that problem has been partially solved in some systems, there is an additional issue: how to consider utility functions in reactive adaptive strategies. Since there is no direct correspondence between utility functions and reactive strategies, this problem is usually delegated to other managerial components with a broader platform view, able to collect information from all utility-related platform components. Other simpler solutions could be found, but further research is needed to this end.

- Software upgrading considering SLOs has been only studied in depth by Wei Li [34]. His solution is based on version coexistence through state sharing [34] since this removes the need of requiring *quiescence* [30]. Quiescence demands old-version inactivity in order to proceed with a software upgrade and this usually compromises service continuity.

  State sharing means that the new version of the component code should be copied on the node where such previous version is running. This might cause problems if the virtual image being used in the previous version did not forecast the need for additional space and during the upgrade there is not enough space in the rented VM for holding both service versions simultaneously. Although software upgrading is not a frequent task, an analysis is needed for evaluating the additional renting costs introduced by this state-share principle in the upgrading approaches.

- Component upgrading needs an underlying upgrade management subsystem in the platform. This upgrade management subsystem should be carefully designed. There are some proposals of subsystems of this kind showing a varying degree of automation (the newer, the better) [9, 10, 62, 1]. Such 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 such a subsystem? Light-weight containers may also help in this concern, as suggested in [48] with the POD (Process Domain) mechanism. PODs have been used for solving the problem of operating-system upgrades, assuming that the OS interface does not change in such upgrade. To this end, the application image is migrated to another host while the underlying OS is being upgraded.

Even when a stable upgrade management subsystem exists, from time to time some critical components of the provider's platform need an update. How may we guarantee service continuity for all deployed applications when those underlying platform components are being upgraded? No complete solution exists yet. The regular approach consists in replicating those critical components and using an upgrading sequence adapted to the replication model being assumed. However, quiescence is needed in this approach in order to complete the state transfer between old and new replicas. As a result, that solution still impedes service continuity. So, further work is needed in this area.

- Services deployed in a PaaS system should generally be highly available. So, one of the goals of platform providers is to ensure service continuity. Since software is not perfect, the *software aging* problem and its *software rejuvenation* fix [71] should be considered in cloud platforms. Software aging refers to the regular problems 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 might cause some malfunctions.

  Some performance losses in already deployed services might be caused by software-aging errors. This should be considered in the monitoring and performance analysis modules of a PaaS platform, triggering a rejuvenation action when needed. 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). Stopping and restarting any faulty instance will not be a problem in this scope. Platform designers do need to consider this fact when a PaaS system is being architected.

## 5 Related Work

There are some other surveys on cloud computing elasticity. Let us briefly summarise their contributions in a chronological order.

Galante and De Bona [19] 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, redimensioning or migration). Such 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.* [44] provide a more extended survey on cloud elasticity. Its first contribution on what was already presented in [19] is its deepest analysis of elasticity in the PaaS service model, although infrastructure-related approaches are also included in that work. A second contribution is its consideration of multiple kinds of SLOs. Besides regular QoS SLOs, Najjar *et al.* also considers *quality of experience* (QoE) and *quality of business* (QoBiz) goals.

Finally, Coutinho *et al.* [15] have written the most recent survey on cloud elasticity we are aware of. They centre their attention in evaluation metrics, generating the best coverage we have found on that subject (i.e., the metrics to be defined and collected by the monitoring subsystem in order to build an elastic system). It studies all service models: IaaS, PaaS and also SaaS. Besides, that survey starts with a long description of the literature review procedure that was followed by its authors, presenting some statistics about the journals, conferences, countries, 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 adaptivity mechanisms) have not been thoroughly discussed in that paper.

Our goal in this document is to complement the results presented in those previous works. We have focused mainly in the automation branch of elasticity limiting our scope to the PaaS service model since, in our opinion, it is the field where there are still several open problems related to adaptive scalability when it should be confronted using autonomic principles.

## 6 Conclusions

Computing service 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 been the first in identifying a complete set of elasticity-related requirements in PaaS systems. Autonomic management, scalability and adaptivity are inherent to the elasticity definition. SLA awareness, composability and service continuity even in upgrading intervals are three other requirements to be carefully managed in these systems. There are multiple mechanisms that partially comply with each requirement. They have been thoroughly described and related in this document, 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, and may be for years, 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] Sameer Ajmani. *Automatic Software Upgrades for Distributed Systems*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, USA, September 2004.

[2] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Modular software upgrades for distributed systems. In Dave Thomas, editor, *20th European Conference on Object-Oriented Programming (ECOOP), Nantes, France, July 3-7*, volume 4067 of *Lecture Notes in Computer Science*, pages 452–476. Springer, 2006.

[3] Peter Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In Raymond T. Yeh and C. V. Ramamoorthy, editors, *2nd International Conference on Software Engineering (ICSE), San Francisco, California, USA, October 13-15*, pages 562–570. IEEE Computer Society, 1976.

[4] Leonardus B. Arief and Neil A. Speirs. A UML tool for an automatic generation of simulation programs. In *Workshop on Software and Performance (WOSP)*, pages 71–76, Ottawa, Canada, September 2000. ACM Press.

[5] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andy Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010.

[6] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Trans. Software Eng.*, 30(5):295–310, 2004.

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Timothy L. Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In Michael L. Scott and Larry L. Peterson, editors, *19th ACM Symposium on Operating Systems Principles (SOSP), Bolton Landing, NY, USA, October 19-22*, pages 164–177. ACM, 2003.

[8] Mohamed N. Bennani and Daniel A. Menascé. Resource allocation for autonomic data centers using analytic performance models. In Parashar and Kephart [47], pages 229–240.

[9] Toby Bloom. *Dynamic Module Replacement in a Distributed Programming System*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, USA, March 1983.

[10] Toby Bloom and Mark Day. Reconfiguration in Argus. In *First International Workshop on Configurable Distributed Systems, London, UK, 25-27 May*, pages 176–187. IEEE, 1992.

[11] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Optimal primary-backup protocols. In Adrian Segall and Shmuel Zaks, editors, *6th International Workshop on Distributed Algorithms (WDAG), Haifa, Israel, November 2-4*, volume 647 of *Lecture Notes in Computer Science*, pages 362–378. Springer, 1992.

[12] Emiliano Casalicchio, Daniel A. Menascé, and Arwa Aldhalaan. Autonomic resource provisioning in cloud systems with availability goals. In Salim Hariri and Alan Sill, editors, *ACM Cloud and Autonomic Computing Conference (CAC), Miami, FL, USA, August 5-9*, pages 1–10. ACM, 2013.

[13] Emiliano Casalicchio and Luca Silvestri. Mechanisms for SLA provisioning in cloud-based service providers. *Computer Networks*, 57(3):795–810, 2013.

[14] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[15] Emanuel Ferreira Coutinho, Flávio Rubens de Carvalho Sousa, Paulo Antonio Leal Rego, Danielo Gonçalves Gomes, and José Neuman de Souza. Elasticity in cloud computing: a survey. *Annals of Telecommunications*, 2015. (in press).

[16] Miguel de Miguel, Thomas Lambolais, Mehdi Hannouz, Stéphane Betgé-Brezetz, and Sophie Piekarec. UML extensions for the specification and evaluation of latency constraints in architectural models. In *Workshop on Software and Performance*, pages 83–88, Ottawa, Canada, September 2000. ACM Press.

[17] Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard E. Sturgis, Daniel C. Swinehart, and Douglas B. Terry. Epidemic algorithms for replicated database maintenance. In Fred B. Schneider, editor, *Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC), Vancouver, British Columbia, Canada, August 10-12*, pages 1–12. ACM, 1987.

[18] Paulo Ferreira, Thomas R. Gross, and Luís Veiga, editors. *EuroSys Conference, Lisbon, Portugal, March 21-23*. ACM, 2007.

[19] Guilherme Galante and Luis Carlos Erpen De Bona. A survey on cloud computing elasticity. In *IEEE Fifth International Conference on Utility and Cloud Computing (UCC), Chicago, IL, USA, November 5-8*, pages 263–270. IEEE, 2012.

[20] Rachid Guerraoui and André Schiper. Fault-tolerance by replication in distributed systems. In Alfred Strohmeier, editor, *International Conference on Reliable Software Technologies (Ada-Europe), Montreux, Switzerland, June 10-14*, volume 1088 of *Lecture Notes in Computer Science*, pages 38–57. Springer, 1996.

[21] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. In *10th International Conference on Autonomic Computing (ICAC)*, pages 23–27, San Jose, CA, USA, June 2013.

[22] Holger Hermanns, Ulrich Herzog, and Joost-Pieter Katoen. Process algebra for performance evaluation. *Theor. Comput. Sci.*, 274(1-2):43–87, 2002.

[23] Paul Horn. Autonomic computing: IBM's perspective on the state of information technology. Technical report, IBM Press, 2001.

[24] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing - degrees, models, and applications. *ACM Comput. Surv.*, 40(3), 2008.

[25] IBM. An architectural blueprint for autonomic computing. White paper, $4^{th}$ ed., June 2006. First edition available in April 2003.

[26] Yexi Jiang, Chang-Shing Perng, Tao Li, and Rong N. Chang. ASAP: A self-adaptive prediction system for instant cloud resource demand provisioning. In Diane J. Cook, Jian Pei, Wei Wang, Osmar R. Zaïane, and Xindong Wu, editors, *11th IEEE International Conference on Data Mining (ICDM), Vancouver, BC, Canada, December 11-14*, pages 1104–1109. IEEE Computer Society, 2011.

[27] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[28] Thomas Knauth and Christof Fetzer. Scaling non-elastic applications using virtual machines. In Liu and Parashar [35], pages 468–475.

[29] Thomas Knauth and Christof Fetzer. DreamServer: Truly on-demand cloud services. In Eliezer Dekel, Randal Burns, and Roy Friedman, editors, *International Conference on Systems and Storage (SYSTOR), Haifa, Israel, June 30 - July 02*, pages 1–11. ACM, 2014.

[30] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Software Eng.*, 16(11):1293–1306, 1990.

[31] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *International Conference on Software Engineering (ISCE), Workshop on the Future of Software Engineering (FOSE), May 23-25, Minneapolis, MN, USA*, pages 259–268, 2007.

[32] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44(2):35–40, 2010.

[33] Willis Lang, Srinath Shankar, Jignesh M. Patel, and Ajay Kalhan. Towards multi-tenant performance SLOs. *IEEE Trans. Knowl. Data Eng.*, 26(6):1447–1463, 2014.

[34] Wei Li. Evaluating the impacts of dynamic reconfiguration on the QoS of running systems. *Journal of Systems and Software*, 84(12):2123–2138, 2011.

[35] Ling Liu and Manish Parashar, editors. *IEEE International Conference on Cloud Computing (CLOUD), Washington, DC, USA, 4-9 July*. IEEE, 2011.

[36] Violeta Medina and Juan Manuel García. A survey of migration mechanisms of virtual machines. *ACM Comput. Surv.*, 46(3):30, 2014.

[37] Peter Mell and Timothy Grance. The NIST definition of cloud computing. Recommendations of the National Institute of Standards and Technology, Special Publication 800-145, September 2011.

[38] Daniel A. Menascé and Mohamed N. Bennani. Autonomic virtualized environments. In *International Conference on Autonomic and Autonomous Systems (ICAS), 16-21 July, Silicon Valley, California, USA*, page 28. IEEE Computer Society, 2006.

[39] Daniel A. Menascé and Paul Ngo. Understanding cloud computing: Experimentation and capacity planning. In *35th International Computer Measurement Group Conference, Dallas, TX, USA, December 6-11*. Computer Measurement Group, 2009.

[40] Daniel A. Menascé, Honglei Ruan, and Hassan Gomaa. QoS management in service-oriented architectures. *Perform. Eval.*, 64(7-8):646–663, 2007.

[41] Emili Miedes and Francesc D. Muñoz-Escoí. Dynamic switching of total-order broadcast protocols. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 457–463, Las Vegas, Nevada, USA, July 2010.

[42] Mohamed Mohamed. *Generic Monitoring and Reconfiguration for Service-Based Applications in the Cloud*. PhD thesis, Université d'Evry-Val d'Essonne, France, November 2014.

[43] Mohamed Mohamed, Mourad Amziani, Djamel Belaïd, Samir Tata, and Tarek Melliti. An autonomic approach to manage elasticity of business processes in the cloud. *Future Generation Computer Systems*, pages –, 2015. (in press).

[44] Amro Najjar, Xavier Serpaggi, Christophe Gravier, and Olivier Boissier. Survey of elasticity management solutions in cloud computing. In Zaigham Mahmood, editor, *Continued Rise of the Cloud: Advances and Trends in Cloud Computing*, pages 235–263. Springer New York, July 2014.

[45] B. Clifford Neuman. Scale in distributed systems. In Thomas L. Casavant and Mukesh Singhal, editors, *Readings in Distributed Computing Systems*, pages 463–489. IEEE-CS Press, 1994. ISBN 978-0-8186-3032-3.

[46] Pradeep Padala, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. Adaptive control of virtualized resources in utility computing environments. In Ferreira et al. [18], pages 289–302.

[47] Manish Parashar and Jeffrey Kephart, editors. *Second International Conference on Autonomic Computing (ICAC), 13-16 June, Seattle, WA, USA*. IEEE Computer Society, 2005.

[48] Shaya Potter and Jason Nieh. AutoPod: Unscheduled system updates with zero data loss. In Parashar and Kephart [47], pages 367–368.

[49] Shriram Rajagopalan. *System Support for Elasticity and High Availability*. PhD thesis, The University Of British Columbia, Vancouver, Canada, March 2014.

[50] Philipp Reinecke, Katinka Wolter, and Aad P. A. van Moorsel. Evaluating the adaptivity of computing systems. *Perform. Eval.*, 67(8):676–693, 2010.

[51] Jerome A. Rolia and Kenneth C. Sevcik. The method of layers. *IEEE Trans. Software Eng.*, 21(8):689–700, 1995.

[52] Nilabja Roy, Abhishek Dubey, and Aniruddha S. Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In Liu and Parashar [35], pages 500–507.

[53] María Idoia Ruiz-Fuertes and Francesc D. Muñoz-Escoí. Performance evaluation of a metaprotocol for database replication adaptability. In *28th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 32–38, Niagara Falls, New York, USA, September 2009.

[54] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[55] Habib Seifzadeh, Hassan Abolhassani, and Mohsen Sadighi Moshkenani. A survey of dynamic software updating. *Journal of Software: Evolution and Process*, 25(5):535–568, 2013.

[56] Upendra Sharma, Prashant J. Shenoy, Sambit Sahu, and Anees Shaikh. A cost-aware elasticity provisioning system for the cloud. In *International Conference on Distributed Computing Systems (ICDCS), Minneapolis, Minnesota, USA, June 20-24*, pages 559–570, 2011.

[57] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. CloudScale: elastic resource scaling for multi-tenant cloud systems. In Jeffrey S. Chase and Amr El Abbadi, editors, *ACM Symposium on Cloud Computing (SOCC), Cascais, Portugal, October 26-28*, page 5. ACM, 2011.

[58] Rhodney Simoes and Carlos Alberto Kamienski. Elasticity management in private and hybrid clouds. In *IEEE 7th International Conference on Cloud Computing, Anchorage, AK, USA, June 27 - July 2*, pages 793–800. IEEE, 2014.

[59] Connie U. Smith and Lloyd G. Williams. Software performance engineering. In Luciano Lavagno, Grant Martin, and Bran Selic, editors, *UML for Real. Design of Embedded Real-Time Systems*, chapter 16, pages 343–365. Springer US, 2003.

[60] Connie Umland Smith. *The Prediction and Evaluation of the Performance of Software from Extended Design Specifications*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, USA, August 1980.

[61] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy C. Bavier, and Larry L. Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In Ferreira et al. [18], pages 275–287.

[62] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma Da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Marc A. Auslander, Michal Ostrowski, Bryan S. Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *Proceedings of the General Track: USENIX Annual Technical Conference, June 9-14, San Antonio, Texas, USA*, pages 141–154. USENIX, 2003.

[63] Suganya Sridharan. A performance comparison of hypervisors for cloud computing, August 2012. Master Thesis (paper 269), School of Computing, University of North Florida, USA.

[64] Michael Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[65] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Third International Conference on Parallel and Distributed Information Systems (PDIS), Austin, Texas, September 28-30*, pages 140–149. IEEE Computer Society, 1994.

[66] Nedeljko Vasic, Dejan M. Novakovic, Svetozar Miucin, Dejan Kostic, and Ricardo Bianchini. DejaVu: accelerating resource allocation in virtualized environments. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), London, UK, March 3-7*, pages 423–436, 2012.

[67] Steven J. Vaughan-Nichols. New approach to virtualization is a lightweight. *IEEE Computer*, 39(11):12–14, 2006.

[68] Andrew Whitaker, Richard S. Cox, Marianne Shaw, and Steven D. Gribble. Rethinking the design of virtual machine monitors. *IEEE Computer*, 38(5):57–62, 2005.

[69] Steve R. White, James E. Hanson, Ian Whalley, David M. Chess, and Jeffrey O. Kephart. An architectural approach to autonomic computing. In *1st International Conference on Autonomic Computing (ICAC), 17-19 May 2004, New York, NY, USA*, pages 2–9, 2004.

[70] Lydia Yataghene, Mourad Amziani, Malika Ioualalen, and Samir Tata. A queuing model for business processes elasticity evaluation. In *International Workshop on Advanced Information Systems for Enterprises (IWAISE)*, pages 22–28, Tunis, Tunisia, November 2014. IEEE-CS.

[71] Jing Zhao, Yanbin Wang, GaoRong Ning, Kishor S. Trivedi, Rivalino Matias Jr., and Kai-Yuan Cai. A comprehensive approach to optimal software rejuvenation. *Perform. Eval.*, 70(11):917–933, 2013.