Experimental Analysis on Autonomic Strategies for Cloud Elasticity

Simon Dupont, Jonathan Lejeune, Frederico Alvares, Thomas Ledoux

Experimental Analysis on Autonomic Strategies for Cloud Elasticity

Simon Dupont, Jonathan Lejeune, Frederico Alvares and Thomas Ledoux
ASCOLA Research Group, Mines-Nantes, INRIA-LINA, Nantes, France.
Email: Firstname.Lastname@mines-nantes.fr
Simon Dupont is also with SIGMA Informatique, Nantes, France.

***Abstract*—In spite of the indubitable advantages of elasticity in Cloud infrastructures, some technical and conceptual limitations are still to be considered. For instance, resource start up time is generally too long to react to unexpected workload spikes. Also, the billing cycles' granularity of existing pricing models may incur consumers to suffer from partial usage waste. We advocate that the software layer can take part in the elasticity process as the overhead of software reconfigurations can be usually considered negligible if compared to infrastructure one. Thanks to this extra level of elasticity, we are able to define cloud reconfigurations that enact elasticity in both software and infrastructure layers so as to meet demand changes while tackling those limitations. This paper presents an autonomic approach to manage cloud elasticity in a cross-layered manner. First, we enhance cloud elasticity with the software elasticity model. Then, we describe how our autonomic cloud elasticity model relies on dynamic selection of elasticity tactics. We present an experimental analysis of a sub-set of those elasticity tactics under different scenarios in order to provide insights on strategies that could drive the autonomic selection of the proper tactics to be applied.**

***Keywords*-Cloud Computing; Elasticity; Autonomic Computing; SLA; QoS**

I. Introduction

According to [1], Cloud Elasticity is defined as *the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible*. For example, Software-as-a-Service (SaaS) providers relying on Infrastructure-as-a-Service (IaaS) have the capability to quickly cope with highly and unpredictable demands by finely allocating resources accordingly and therefore meeting Service Level Agreements (SLAs) previously established with their customers.

In such dynamic environments, human intervention is becoming more and more difficult or even impossible. The management of complex architectures along with the integration and calibration of all the parameters and constraints that come with to keep the system running in an optimized way (e.g., in terms of QoS and cost), is a very complex and error-prone task. Moreover, human intervention may be too slow to react to some situations such as network, hardware or software failure or sudden workload oscillation. For those reasons, Autonomic Computing [2] has been largely adopted to manage elasticity of cloud services.

Currently, Autonomic Cloud Elasticity is mainly used to scale the infrastructure resources in the IaaS layer. Nevertheless, although the recent advances in allowing rapid resource provisioning [3], infrastructure elasticity has faced technical and conceptual limitations that prevent infrastructure services to be fully and rapidly elastic. First, *resources are limited*, which means that resources cannot be scaled up infinitely and there could be times where provisioned resources would be insufficient for infrastructure services' customers to cope with increasing demands. Second, infrastructure resources *initiation time* may be too long (ranging from few seconds to several minutes [4]) and hence not very reactive. Third, customers are usually charged for using resources and, depending on the adopted resource pricing model, they may suffer of phenomenon known as *partial usage waste* [5], in which they are charged for more than what they actually consume. Last but not least, despite of the recent efforts in conceiving and managing datacenters in a more energy-efficient manner, the energy consumption due to Cloud infrastructures still remains an issue [6].

We advocate that the software at the SaaS layer can take part in the elasticity process and overcome infrastructure elasticity limitations. The reason for this resides in the fact that the overhead of software reconfigurations can usually be considered as negligible, if compared to those of infrastructure services in lower layers. Hence, software services have a tremendous potential to be dynamic and elastic so as to fit in contexts where only the infrastructure elasticity is not enough. Concretely, *Software Elasticity* can act as an extra elasticity capability that goes beyond the infrastructure elasticity when resources are scarce. For example, one can easily and dynamically replace resource-consuming software components by less resource-consuming ones. In the same sense, those less resource-consuming software components can also be used as an alternative to absorb peaks of workload instead of either adding infrastructure resources and releasing it straight away; or getting the resource way too late due to long initiation times.

This paper presents an autonomic approach to manage cloud elasticity in a cross-layered manner. First of all, we propose a model for software elasticity which draws inspiration from the two-dimensional infrastructure elasticity, that is, the elasticity based on vertical and horizontal scaling. That results in four di-

mensions of elasticity, say, software/infrastructure/vertical/horizontal, which provides Cloud providers with more options that can be used to face the previously mentioned elasticity limitations. In order to manage all those dimensions in a proper manner, we propose the orchestration of both elasticities by composing basic elasticity actions in a synchronous way, using parallel and sequence operators. From that composition, the Cloud Administrator may derive a number of extra composed actions that can applied upon different events reflecting the need for Cloud resource reconfiguration. We present an experimental analysis on the use of a sub-set of those *elasticity tactics* (i.e., event-action pairs) under different scenarios in order to provide insights on the criteria and the preferences on them that could drive the autonomic selection of the proper tactics to be applied. Those experiments were conducted on OpenStack, an open-source Cloud Infrastructure Manager, which has been deployed at Grid'5000, a French national wide grid for experimental infrastructure testbed.

The rest of the paper is organized as follows. Section II discusses with more details the limitations of infrastructure elasticity. Section III presents the concept of software elasticity and advantages as infrastructure complementarity. Section IV presents our proposal of autonomic approach based on cross-layered elasticity tactics, whereas Section V presents the analysis on the performed experiments. Sections VI and VII reviews the related work and concludes this paper.

II. Infrastructure Elasticity

In this section, we recall some basic definitions before emphasizing the limitations of the infrastructure elasticity model.

A. Definitions

Infrastructure Elasticity (IaaS layer): Infrastructure Elasticity is the ability to rapidly scale infrastructure resources on demand. Figure 1 illustrates the current infrastructure resources elasticity model which can be achieved by horizontal or vertical scaling:

- **Infrastructure Horizontal Scaling** (HS_{infra}): adjusts the VM's pool size by adding (*scale out* a.k.a. SO_{infra}) or removing (*scale in* a.k.a. SI_{infra}) VM instances.
- **Infrastructure Vertical Scaling** (VS_{infra}): resizes existing VM instances by increasing (*scale up* a.k.a. SU_{infra}) or decreasing (*scale down* a.k.a. SD_{infra}) resources allocated (e.g., CPU or RAM). This is usually done by switching from an instance offering to another (e.g., Off_{vm}(small) to Off_{vm}(large) in the case of SU_{infra}).

B. Limitations

1) Resources are limited: For the consumer, the provisioning often appears to be unlimited and available to be requested in any quantity at any time. However, in reality, there are several limitations in this

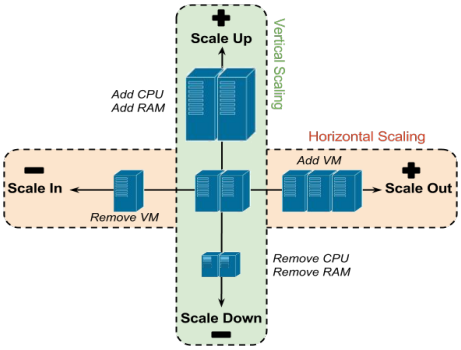
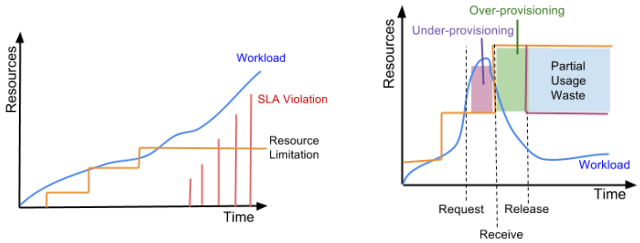


Figure 1. Infrastructure Elasticity: Horizontal and Vertical Scaling



(a) (b)

Figure 2. Resources Elasticity limitations

regard: (i) IaaS providers usually restrict the number of infrastructure resource instances (e.g., VMs) to be allocated for each user in order to better manage their datacenters, but also due to the limitations of physical resources (e.g., physical machines or energy). For example, 20 instances (VMs) per zone is the per-user maximum allowed for Amazon EC2 services; (ii) in case of failure (e.g., network), resource availability decreases; (iii) the consumer may find her/himself constrained with respect to the amount of resources to be used due to a predefined budget or design constraints.

Therefore, IaaS resources cannot be provisioned infinitely and the maximum resource utilization can be quickly reached, whether it is imposed by infrastructure providers (e.g., small private Cloud infrastructures), by consumers' restrictions (e.g., budget constraint) or by the hardware in the case of VS_{infra} (e.g., maximum RAM reached). The consequences of these limitations on the customers' applications can be harmful, as illustrated in Figure 2(a). When resources become saturated (orange line) and the demand increases (blue line), it is straightforward that QoS of demanding SaaS applications such as response time and availability are compromised, which may result in end-users dissatisfaction, financial penalties due to SLA violations (red line) and so forth.

2) Resource initiation time is significant: Although the resource elasticity capability, which enables resource provisioning to cope with dynamic environments (e.g., whose workloads varies within the same day or hour), the non-negligible resource initiation time may be way too long and hence prevent just-in-time provisioning. For example, although the recent improvements in virtualized systems, the initiation time of a VM may

vary from few seconds to several minutes depending on several factors like the hypervisor, operating system, memory size, network configurations, among others. In general, the larger the VM (in terms of memory) more time is needed to configure, deploy and launch it [4]. In this context, it might be completely useless having the required resource too long later if the concerned application needs to react immediately to cope with a change.

Figure 2(b) illustrates an example of such a limitation. In order to absorb a couple-of-minutes workload peak, the SaaS provider requests some resources (e.g., VMs). It turns out that, due to the initiation time, the requested resources are ready to be used only few minutes later, when the workload is decreasing. To sum up, the infrastructure resource provisioning was not reactive enough to cope with the change (workload increase) and thus compromising the software services' QoS and SLA fulfilment. Furthermore, by the moment resources were made available, they were for no use. Consequences can be even worse in cases of oscillating scenarios as it may suffer of "ping-pong effect" in the request/release of resources [7].

In theory, VS_{infra} allows to tackle this responsiveness limitation by almost instantly increasing (or decreasing) resources of existing instances, as long as the maximum hardware capacity is respected. However, in practice, runtime VS_{infra} (without service interruption) is rarely implemented as it imposes a number of technical constraints in terms of compatibility of the infrastructure software stack (operating systems, hypervisor, Cloud management tool) [8]. That is to say that, most of the times, VS_{infra} imposes a service downtime of existing instances while their capacity (Off_{vm}) are being increased (or decreased). For these reasons, Infrastructure Elasticity is mostly implemented using the horizontal scaling, as it is easier to implement.

3) Pricing is per instance-hour: Pay as You Go is a utility computing billing model. The time granularity of IaaS resource reservation and the implied resource billing model (e.g., hourly, daily, etc.) may also lead SaaS providers to either pay more than they actually consume (e.g., when it requests resource during a workload peak and releases it right after) or to take into consideration the reservation duration before deciding to request more resource. Figure 2(b) also illustrates the issues related to the billing time granularity. When a resource allocation becomes useless (e.g., when the workload starts decreasing), SaaS providers can just release the resource. In this case, they will suffer from a phenomenon called *partial usage waste* [5] in which they will pay the entire billing cycle even if the usage duration is inferior to the cycle.

Alternatively, in the hope that this resource will be of any help during the billing cycle, SaaS providers may employ a *use-as-you-pay* policy [9], in which, the resource will be kept until the end of the associated billing cycle (with a negative impact on energy consumption). In any case, regardless the usage policy, consumers are likely to pay more than what they actually consume.

III. Enhance cloud elasticity with software elasticity

In order to respond to the limitations of the current Cloud elasticity model, we advocate that the software at the SaaS layer can take part in the elasticity process. First, we define some software elasticity concepts, then we emphasize the synergy between SaaS and IaaS elasticities and we give an illustration.

A. Software Elasticity

We propose a definition of software elasticity and present the underlying concepts by establishing an analogy with infrastructure elasticity.

Software Elasticity is the capability of a software to adapt itself (ideally in an autonomic manner) to meet demand changes and/or infrastructure resources limitations. By analogy with the infrastructure elasticity where IaaS resources (e.g., VMs) are dynamically adjusted to fulfil SLA contracts, *Software Elasticity* provides means for adjusting SaaS resources (e.g., software components) in a seamless and instantaneous way to better achieve SLA expectations. As the same principle of infrastructure elasticity we distinguish horizontal and vertical scaling :

- **Software Horizontal Scaling** (HS_{soft}): in the same way as for the HS_{infra}, HS_{soft} can be achieved on the SaaS layer. Then, we introduce SO_{soft} and SI_{soft}, that respectively refers to add (*scale out*) or remove (*scale in*) software components on the fly.
- **Software Vertical Scaling** (VS_{soft}): we draw inspiration from Infrastructure Vertical Scaling to extend the Software elasticity capability by adding an extra scalability dimension called VS_{soft} that refers to change the offering of existing component. In this case, the different Off_{comp} correspond to variegate the functionalities or non-functional aspects (e.g., security or logging). Then, we introduce SU_{soft} and SD_{soft}, that respectively refers to *scale up* or *scale down* functionalities and/or non-functionalities to the existing component.

B. Benefits of IaaS and SaaS elasticities complementarity

There are three main benefits in resulting from this synergy.

Alleviate the use of infrastructure resources: resources cannot be scaled up infinitely and the maximum resource utilization can be attained. Changing the software offering at runtime (e.g., SD_{soft}) may help relieving the resources (e.g., CPU, RAM, etc.) and may allow to absorb few-minutes workload peaks. As a consequence the starting of new resources in the infrastructure layer can be avoided, which could be long and costly [4].

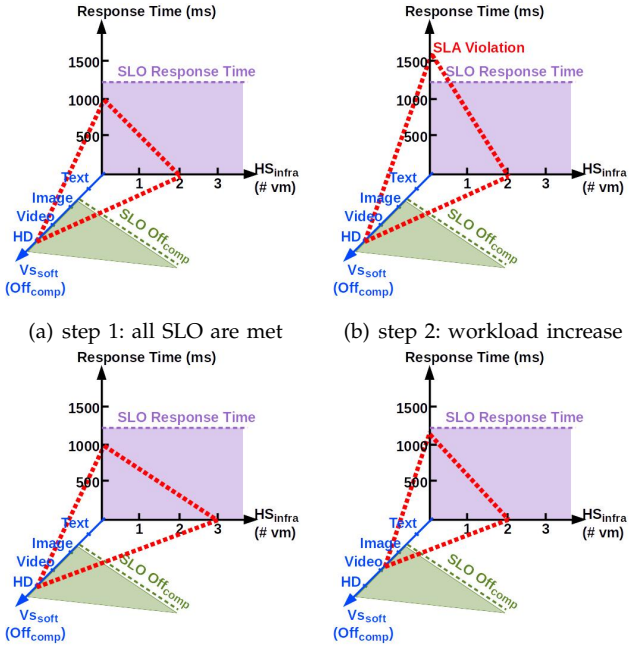
Improve responsiveness of scaling: in order to deal with highly dynamic environment (e.g., network fluctuations, unpredictable workload), the system must be

modular and flexible enough but also provide rapid reconfiguration capabilities. The later requirement is compromised due to non-negligible resource initiation times, which may take several minutes. Software elasticity, on the other hand, can be more reactive, since, in general, it involves lightweight components whose deployment and initiation take less time than infrastructure resource ones.

Improve expression capability of elasticity: Table I summarizes the four scaling dimensions previously outlined along with the eight underlying APIs. In this context, Cloud administrator can create more sophisticated reconfiguration plans by orchestrating actions of different dimensions and then manage finely and effectively his/her cloud resources.

C. Illustrative Example

In order to illustrate the concept of Software Elasticity, and more specifically the new VS_{soft} dimension, we give an example which consists of a SaaS application from the domain of internet advertisement. The application behaves in an autonomous manner by choosing the appropriate Off_{comp} for the advertising component in accordance with the current workload, the runtime context and some constraints on the advertisement quality, the performance and the energy compliance. In this example, the advertising component provides four different offerings which are Video HD, Video, Image or Text. The Video HD and Video offerings provide a better Quality of Experience (QoE) to end-users than Image or Text ones but more computation time to operate the service.



(a) step 1: all SLO are met (b) step 2: workload increase

(c) step 3: infrastructure scaling (d) step 3′: software scaling

Figure 3. Impact of HS_{infra} and VS_{soft} on the response time

Figure 3 shows the impact of HS_{infra} and VS_{soft} on the response time metric. We voluntarily fixed the value of the dimensions HS_{soft} and VS_{infra} for a better understanding. We can see HS_{infra}, with the amount of resources (e.g., number of VMs), on the horizontal axis and response time expectations on the vertical axis for a specific workload and request type. The blue axis represents the VS_{soft} with the different Off_{comp} for the advertising component.

We consider a SLA established between the end-users and the SaaS provider. This SLA contains two Service Level Objectives (SLO) which describe a commitment to maintain a particular state of the service in a given period. The first one corresponds to a SLO Response Time indicating that the service must respond under 1200 ms (purple area). The second one formally specifies the Off_{comp} usage expected by the end-user and guaranteed by the SaaS provider. In our example, it is stated that the advertising component cannot be provided with the Off_{comp} Text (green area). We can imagine SLOs specifying the percentage of utilization of each Off_{comp} (e.g., 75% Video HD, 20% Video, 5% Image) for a certain time window (e.g., 1 day).

In Figure 3(a), we consider a given workload and an initial application configuration (infra: 2 VMs ; soft: Off_{comp} HD) which meets all SLOs. Let's consider, in a second step (see Figure 3(b)), an increasing workload which leads to a response time SLO violation with the actual configuration. In order to retrieve a suitable configuration (i.e., respecting the SLOs), it becomes necessary to reconfigure the application to absorb the new demand. If we consider each dimension independently for simplicity, there are two possible reconfigurations:

- (i) increase the number of VMs by executing a SO_{infra} and then balance the load on more workers (see Figure 3(c));
- (ii) change the Off_{comp} by executing a SD_{soft} and then reduce the request computation time (see Figure 3(d)). In this context, it should be noted that the execution of SD_{soft} must be done in accordance with the SLO in terms of Off_{comp} usage.

In this simple example, we have considered only two scaling dimensions instead of the four usable. Moreover, the suggested reconfigurations involve only one or the other dimension. As we will see later, it is possible to make use of different scaling dimensions within the same reconfiguration plan and then benefit from their joint use.

IV. Elasticity Strategies for Autonomic Cloud Services

In this section, we describe our autonomic cloud elasticity model which relies on dynamic selection of elasticity strategies.

A. PaaS Autoscaling Service

Our autonomic cloud elasticity model can be likened to PaaS which aims to manage the elasticity of re-

Table I
CLOUD ELASTICITY SCALING ACTIONS

Scaling Dimension	API Name	Description
Infrastructure Horizontal Scaling (HS_{infra})	Scale Out Infrastructure (SO_{infra})	Add VM(s) to the pool
	Scale In Infrastructure (SI_{infra})	Remove VM(s) from the pool
Infrastructure Vertical Scaling (VS_{infra})	Scale Up Infrastructure (SU_{infra})	Increase offering (Off_{vm}) of existing VM(s)
	Scale Down Infrastructure (SD_{infra})	Decrease offering (Off_{vm}) of existing VM(s)
Software Horizontal Scaling (HS_{soft})	Scale Out Software (SO_{soft})	Add software component(s) to the application
	Scale In Software (SI_{soft})	Remove software component(s) to the application
Software Vertical Scaling (VS_{soft})	Scale Up Software (SU_{soft})	Increase offering (Off_{comp}) of existing component(s)
	Scale Down Software (SD_{soft})	Decrease offering (Off_{comp}) of existing component(s)

sources (e.g., software component, virtual machines, physical infrastructure). This kind of service is usually called autoscaling service. In the rest of this section, we call Cloud Administrator (CA for short) the person in charge of the management of the autoscaling service. In order to manage the Cloud system, CA needs a way to monitor and act on both IaaS and SaaS resources. These resources must provide the necessary interfaces, that is to say, the APIs enabling the PaaS Autoscaling Service to interact with them.

B. Autonomic Cloud Elasticity Model overview

The autonomous management of our model results in an implementation of the MAPE-K loop reference model [2], as shown in Figure 4.

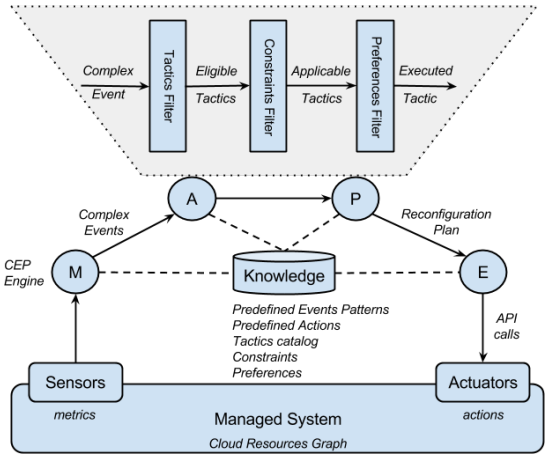


Figure 4. Cloud Resources Model Overview

The MAPE-K loop is applied to a cloud resources graph. The *autonomic manager*, that is to say the MAPE-K loop, interacts with the managed system through two interfaces that are the *sensors* and *actuators* to supervise and act on the system respectively.

The loop is fed by the sensors following a push mode which means that the managed system (i.e. cloud resources) pushes metrics to the *M* of the autonomic manager. Based on the received data, *M* uses Complex Event Processing (CEP [10]) to aggregate metrics and generate complex high level events which denote pertinent information about the health status of the managed system requiring reconfiguration decision.

Events are triggered by *M* and send to decision part of the loop for analyse. The decision, which corresponds to *A* and *P* phases in our work, comprises three steps in the form of consecutive filters that we will detail later in this section. The decision part will output a reconfiguration plan which to be applied on the managed system by the *E* phase through actuator calls and then meet the problem represented by the incoming event.

The *K* comprises data shared among all MAPE phases such as current cloud resources graph with associated metrics values, runtime *constraints* and *preferences* expressed on the system. It also contains predefined symptoms and reconfiguration plans in the form of *event patterns* and *predefined actions* respectively. Last but not least, *K* includes a mapping between predefined events and actions that results in a catalogue of event-action pair called *tactics*. As we shall see later, decision part aims to filter these tactics by confronting them to monitoring, runtime constraints and preferences expressed by the Cloud Administrator.

C. Resources Model: managed system

We focus on n-tier web applications and we model a resource hierarchy as follows: each tier of the application is composed of virtual machines (VM for short) accommodating software components. VMs are hosted on an infrastructure corresponding to a set of physical machines (PM for short).

Figure 5 illustrates a simple cloud resources model instance (i.e. managed system) which is a 2-tiers application deployed on a 2 PMs' infrastructure. Each tier consists of VMs hosted on PM (dash lines). Each VM contains the software components associated with its tier. Thus, one can see an instance of the model as a resource graph composed of two sub-trees. The left one represents the physical architecture view of the application (considering the PM-VM mapping) whereas the right one gives the logical architecture view (with the mapping tier-VM).

D. Monitor: events

In our model, each cloud resource has a set of metrics reflecting its state of health. The managed system exhibits these metrics through sensors making possible to capture their values at runtime. The monitor phase can collect and aggregate multiple sensors values over time in order to make timely relevant information about the system health status in the form of *event*.

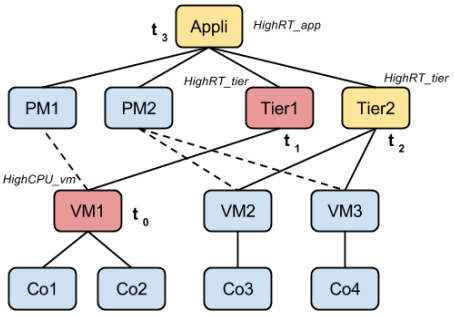


Figure 5. Instance of cloud resources model (managed system)

We distinguish two kinds of events: Basic Event and Complex Event. Basic Events denote pertinent information for one metric of one resource. For example, if CPU consumption of one VM has exceeded a maximum threshold, generate a *HighCPU_vm* event. Complex Events are composite events and may involve one or few metrics of one or several resources with different time scales. We rely on CEP [10] to generate new complex events through aggregation and composition of basic ones. Then, it becomes possible to detect complex relationships between events overtime by defining correlation rules also known as Event Patterns.

CA can take advantage of the tree structure of the cloud resources model to define its events patterns specifying hierarchical-based relationships. Then, he could define a CEP rule that looks at some nodes' states according to the parent node's state relying on the father-son relationship as shown in red in Figure 5: if the response times observed on a tier exceeds a maximum threshold (*HighRT_tier* on Tier1) and the underlying VM(s) have excessive CPU consumption (*HighCPU_vm* on VM1). This example shows an event pattern which could generate new kind of complex event named *HighRT_tier_Overloaded_vms* by aggregating basic events. The hierarchical structure also enables to track the temporal evolution of a symptom watching the propagation of correlated metrics in the tree overtime as shown in yellow on the same figure: a response time problem began on Tier1 at t_1, and then appeared on Tier2 at t_2 before impacting the application itself at t_3. This example can also be designed as event pattern to generate complex events called *HighRT_upward*. Moreover, hierarchical structure allows to identify trends (upward or downward) or causal relationships between symptoms.

Based on his knowledge of the system and the potential help of test team members who performed scalability tests, CA can define situations or symptoms that need to pay attention by specifying some Event Patterns at design time. These patterns are stored in the Knowledge and evaluated at runtime.

E. Execute: predefined actions

Cloud resources are associated with a set of actions that can be operated through actuators, offering reconfiguration capability to the system. The set of actuators exhibited by the system constitutes its reconfiguration API.

As for events, we distinguish two kinds of actions: Basic Action and Complex Action. Basic actions result in a single actuator call. The basic actions considered in our work have been listed and detailed previously in Table I. With software elasticity, it becomes possible to benefit from the advantages of the different scaling dimensions by using their related actions in a smart and coordinated way. We propose the concept of Complex Action which denotes an orchestration of basic actions constituting an executable self-contained reconfiguration plan.

Figure 6 shows an example of complex action using the BPMN 2.0 formalism. The example meets a need for additional resources. It involves two scaling dimensions from two layers (HS_{infra} and VS_{soft}) through 3 different basic actions which are composed in parallel and sequence.

The goal of this complex action is to absorb the resource initiation time when a SO_{infra} is called and therefore face the reactivity limitation previously mentioned. Concretely, when a need for additional resources occurs (e.g., HighRT_tier - step 1), we execute in parallel a SO_{infra} (e.g., adding one VM to the pool - step 2) and a SD_{soft} (e.g., switching the Off_{comp} from Video HD to Image - step 2') to relieve infrastructure resources during the initiation of the VM. We wait for a specific duration (e.g., timer) or until resources are ready (step 3), then we execute a SU_{soft} (e.g., switching back to Video HD - step 4) to retrieve a nominal state.

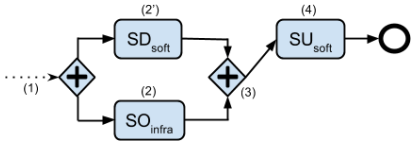


Figure 6. Complex action example: $(SD_{soft} \parallel SO_{infra})\ ;\ SU_{soft}$

Another example of complex action could be the *migrate* action allowing migration of stateless VM(s) from a source PM to a target PM. This means execute a SO_{infra} on target PM and wait until VM is ready, then execute SI_{infra} on source PM.

As for events patterns, CA is able to provide a set of predefined scaling actions (basic or complex) at design-time relying on its knowledge of the systems (e.g., following calibration tests) and possibly with the assistance of architects (system or software). These actions are stored in the Knowledge and will be applied at runtime by the MAPE-K loop. We will discuss later the manner in which the actions are selected.

It should be noted that the number of predefined symptoms and actions remains reasonable. CA does not specify thousands of event patterns or actions. This is an iterative and incremental approach in which the CA is able to refine its model by adding/removing symptoms/actions progressively by relying for example on calibration tests or the behaviour of the application

at runtime.

F. Reconfiguration Decision

This section aims to present the decision part of our autonomic platform. We will detail the three different steps, which take the form of filters, making the bridge between the M and E phases of the MAPE-K loop.

1) Tactics Filter – Event-Action Mapping: CA is able to describe situations requiring reconfiguration of the system, in the form of event patterns, but also possible reconfiguration plans that it wants to apply to the system, in the form of predefined actions. Besides, he needs to make the link between events patterns and predefined actions. This link is done through the concept of *Tactics*, defining a reconfiguration solution in response to a provisioning issue.

A tactic is actually an event-action pair defined by CA and representing a well-known *IF-THEN* statement (e.g., if(event) then action). Thereby, the CA is able to provide the mapping between events and actions which corresponds to establish a catalogue of tactics stored in the Knowledge. These tactics will be automatically filtered at runtime depending on the triggered events. When an event occurs at runtime, the *Tactics Filter* takes a look at the event-action mapping stored in the Knowledge, that is to say select *eligible tactics* previously defined as solutions to the problem represented by the incoming event. Figure 7 illustrates the event-action mapping. We can see a set of predefined Events which have been designed in the form of event patterns and a set of predefined Actions. Each pair $(e, a) \in Events \times Actions$ defines a tactic. The tactics catalogue is the set composed of all (e, a) pairs.

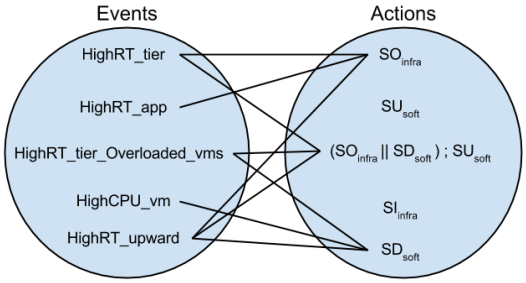


Figure 7. Example of Event-Action Mapping

2) Constraints Filter – Context: The runtime context can induce some constraints that may prevent to execute eligible tactics. Although the monitoring phase provides a view of the current context within the various probes values, the system is also faced with runtime constraints defined on resources. These constraints can be numerous and various (e.g., budget constraint, technical restriction, etc.) and are stored in the Knowledge to be evaluated at runtime. This is the purpose of the second filter named *Constraints Filter* which takes as input the eligible tactics (provided by the first filter) and the run-time constraints in order to identify unsuitable tactics and thus outputting only applicable tactic(s) (see Figure 4).

An example of constraint could be the maximum capacity of PM in terms of CPU or RAM. In the same way, an SLO regarding the use of a Off_{comp} can be considered as constraint (e.g., Off_{comp} *Text* can not be used from 8:00 p.m. to 11:00 p.m.). Such constraints may limit the scope for reconfiguration by preventing to apply certain tactics. For example, in the case of PM capacity constraint, if the maximum capacity of the PM is reached, adding a new VM (SO_{infra}) on it will be impossible. With regard to the second constraint, the SLO may limit the use of VS_{soft} dimension. This filtering step may be sufficient to identify the tactic to execute if only one tactic remains applicable, but if it is not the case, we need a last filtering step.

3) Preferences Filter – Strategy: By specifying the event-action mapping, Cloud administrator can give n possible actions to solve a same problem, which is formalized by n eligible tactics for an event. When an event occurs at runtime, after applying the first two filters, it can remain few applicable tactics. Which one to choose? Are they equivalent? What is the best from the provider point of view? And for the customers? In order to answer these questions and decide among the remaining tactics, we need a last filter based on CA *preferences*.

Although the execution of these tactics provide an answer to the problem, they are not equivalent. Indeed, there may be several criteria to take into account, as follows:

- Cost: the financial impact including the price of the infrastructure and licensing;
- QoS: the effects in terms of QoS;
- QoE: the impact on VS_{soft} dimension;
- Elasticity Time: the total time required to execute the tactic;
- Responsiveness: the ability of the tactic to react quickly. It differs from the elasticity time that considers the full execution of the tactic.

The list of criteria is not exhaustive and may be changed or completed depending on the system, the kind of application, the type of workload, etc. Relying on its system knowledge and calibration tests, the CA will give rates on tactics by assigning values to criteria in the form of *strategies* and stores them in the *K*.

In the next section, we evaluate several tactics by following the calibration process that could be achieved by the CA. Then, we analyse and discuss the results in order to present advantages and disadvantages of each tactic depending on workload scenario and runtime context. This approach allows the identification of criteria for comparing the predefined tactics and setting preferences in the form of strategies. The objective of our experiments is to provide insights on the criteria driving the autonomic selection of elasticity tactics.

V. Experiments

In this section, we present an experimental study on the impact of different tactics over a cloud application

in terms of reconfiguration actions and performance (e.g. response time and availability). The final goal of this is to show how a cloud administrator can define the strategy filter among a set of tactics.

A. Experimental testbed and configuration

1) Application configuration: We consider a cloud application developed in a SaaS-fashion. The application is architecturally organized in 2-tiers: the first one is the load balancing tier (*lbt*), whereas the second one is the business tier (*bt*). For the *lbt*, we rely on Nginx 1.6.2 to distribute workload across our multiple workers of the *bt*. The second tier consists of Java application and a web server Jetty. The software component that implements the *lbt* is equipped with software elasticity capabilities, meaning that it has several offering levels (Off_{comp}). We implement them by gradually increasing the degree of CPU intensiveness at each level. Finally, each component instance (of the *lbt* or *bt*) is deployed at a virtual machine, which is offered by the infrastructure provider.

2) Infrastructure configuration: The experiments were conducted on Grid'5000 Nancy, by using 7 physical machines linked by a 20 Gbit/s Ethernet switch. Each machine has two 2.8GHz Xeon processors (4 cores per CPU) and 15GB of RAM, running Linux 2.6 Machines. The used cloud computing platform is Openstack Grizzly 1.0.0, which requires one dedicated physical machine, the cloud controller, to manage the system. Consequently, 6 physical machines are dedicated to host virtual machines, which in turn, are pre-configured to run Ubuntu 12.04.

3) Autonomic Manager: Our autonomic manager process is hosted on the cloud controller machine. It monitors the average response time of received and processed requests by aggregating the *lbt*'s Nginx logs in a given time window (in our case 15 seconds). According to a comparison of this value with two threshold values, two event patterns can be issued :

- ***"High Response Time"*** occurs when the monitored value is higher than an upper threshold. It reflects an under-provisioned system leading eventually to a SLA violation;
- ***"Low Response Time"*** occurs when the monitored value is lower than a lower threshold. It reflects an over-provisioned system leading eventually to useless operating costs.

In response to ***High Response Time***, the autonomic manager may execute either SO_{infra}, so as to increase the system's capacity in terms compute resources (VMs); SD_{soft}, in order to absorb more workload with the same amount of compute resources; or a parallel composition of these two basic actions, composed in sequence with SU_{soft}. Regarding the event ***Low Response Time***, the autonomic manager may execute either SI_{infra} so as to free allocated compute resources or SU_{soft} to upgrade the QoE by increasing the Off_{comp} (see Table II). For sake of simplicity we do not consider the basic actions associated with the VS_{infra} and HS_{soft}.

Table II
DEFINITION OF TACTICS USED IN THE EXPERIMENT

Event	Action
High Response Time	SO_{infra} SD_{soft} $(SO_{infra} \parallel SD_{soft}); SU_{soft}$
Low Response Time	SI_{infra} SU_{soft}

B. Experimental evaluation

1) Experimental setup: We fixed the upper and lower threshold firing the *High/Low Response Time* events to 400*ms* and 20*ms*, respectively. In the case of the upper threshold, that corresponds to a threat threshold small enough to avoid that the response time reaches 1000*ms*. With respect to the lower one, it is big enough to identify an over-provisioning case.

In our evaluation analysis, we considered three experiments. Each experiment directly applies for each event pattern, i.e. *High Response Time* and *Low Response Time* a unique action (cf. Table II) over the business tier:

- **Experiment 1**: SO_{infra} and SI_{infra}, where SO_{infra} (resp. SI_{infra}) action adds (resp. removes) one virtual machine;
- **Experiment 2**: SU_{soft} and SD_{soft}, where SU_{soft} (resp. SD_{soft}) action degrades (resp. upgrades) one level of Off_{comp} for each running component instance/virtual machine in the tier;
- **Experiment 3**: $(SO_{infra} \parallel SD_{soft}); SU_{soft}$ and SI_{infra}, where $(SD_{soft} \parallel SO_{infra}); SU_{soft}$ is a composite action (see Figure 6) which degrades 4 levels of Off_{comp} for each running component instance/virtual machine in the tier and waits until the virtual machine resulting from the SO_{infra} starts, then it upgrades back 4 levels of Off_{comp}.

Figure 8 shows, through those experiments, the evolution of the state and performance of the business tier over the time with a given input workload scenario. A workload value is a number of requests per second sent to the application. In all sub-figures, this value is represented by the black curve of the right y axis. The considered workload scenario lasts 40 minutes and has three phases:

- **First phase**: the workload increases during 10 minutes with a medium speed-up (0.4 request/sec^2) and then decreases during 3 minutes up to the starting point;
- **Second phase**: the workload increases during 10 minutes with a half speed-up of the first phase (0.2 request/sec^2) during 10 minutes and then decreases during 3 minutes up to the stating point;
- **third phase**: three peak loads are injected at regular interval characterizing a high speed-up (1.6 request/sec^2) for a short period of time.

These three phases enable us to evaluate the consequences of the different considered tactics over the application with different speed-up value.

During the experiments, we gather and observed the following metrics:

- **Infrastructure's Size**: the number of running virtual machines (Figures 8(a), 8(b) and 8(c));
- **Software Offering (Off_{comp})**: the offering of the application, which have 5 different values. The higher the value, the higher the QoE (Figures 8(d), 8(e) and 8(f));
- **Number of Failed Requests**: the number of requests which are not able access to the service in one second (Figures 8(g), 8(h) and 8(i)). This metric shows the unavailability of the application;
- **Average Response Time**: the average response time of processed requests in one second (figures 8(j), 8(k) and 8(l))

The initial configuration, that is, the system configuration before receiving any request, is one component instance (which corresponds to one virtual machine) for each tier and the maximum level of Off_{comp} for the component in the *bt*.

Finally, the reconfiguration execution time is represented by a green area in Figure 8. Since this primitive is costly in terms of reconfiguration time, it is interesting to show the behaviour of the application during the action execution.

2) Analysis and discussion: We can notice that Experiments 1 and 3 follow the same behaviour in terms of infrastructure's size (Figures 8(a) and 8(c)): the number of VMs increases whenever a *High Response Time* event occurs during a workload increase. Indeed, as we can see in Figures 8(j) and 8(l), the action corresponding to this event starts its execution whenever the average response time exceeds the upper threshold. Since Experiment 2 is deprived of action SO_{infra}, the number of VMs remains constant (Figure 8(b)) all the time.

Experiment 1 keeps the same level of Off_{comp} (Figure 8(d)) since the SD_{soft} action is disabled, contrary to Experiment 3, which decreases the Off_{comp} at each occurrence of event *High Response Time* (Figure 8(f)). This difference has an impact over the average response time and the unavailability of the application. Indeed, in the first phase of our scenario, there is a huge degradation of performance during the reconfiguration execution of Experiment 1 (Figures 8(g) and 8(j)), whereas for Experiment 3, we remark a full availability and an excellent average response time (Figures 8(i) and 8(l)). This shows that the parallelization of Off_{comp} degradation and the SO_{infra} allows to absorb the workload increasing during the reconfiguration period, which allows the application to keep a constant full availability and thus meet the SLAs. However in the second phase, as can be noticed in Figures 8(g), 8(i), 8(j) and 8(l), Experiments 1 and 3 do not induce a worsening of performance. Consequently, in Experiment 3, the Off_{comp} has been uselessly degraded. In the third phase of our scenario, Figures 8(g), 8(i), 8(j) and 8(l) show that a SO_{infra} is inefficient: the performance is degraded during the peak load in both experiments. Moreover, the long reconfiguration period (VM initiation time) makes the new VM useless since it becomes effective only after the bursts.

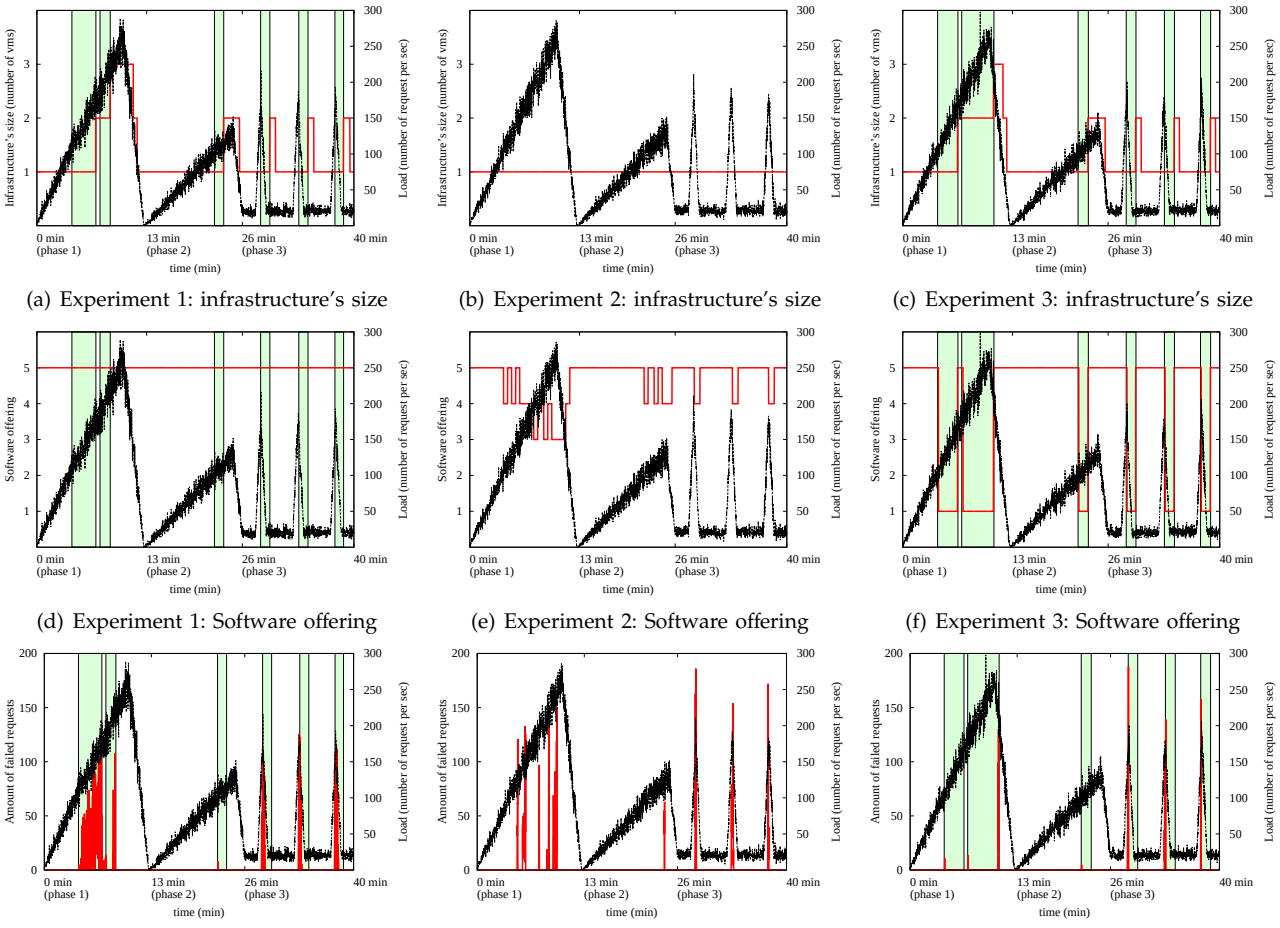
Concerning the Experiment 2, Figure 8(e) shows that the modification of Off_{comp} level induces a short execution time in comparison with the SO_{infra} action. Nevertheless, we can remark an oscillation period between two successive values when the workload increases (especially in the phases 1 and 2) implying that *High Response Time* event and *Low Response Time* are triggered alternatively. This phenomenon can be explained thanks to two characteristics of the software elasticity:

- High Reactivity: the new configuration is quickly operational implying that workload variation during reconfiguration execution has no impact over performance
- Off_{comp} levels are too coarse grained: the difference of performance between two successive Off_{comp} levels is too large, which makes the autonomic manager oscillates. In other words, upon a *High Response Time*, the manager is induced to degrade the level of Off_{comp}. At the degraded level, the performance is too high so the lower threshold is exceeded and a *Low Response Time* is fired. This behaviour is repeated while the load increases.

Figures 8(h) and 8(k) show that this oscillation is costly in terms of performance. In the third phase, we can observe that the performance of Experiment 2 is the same than the other two: we can notice an unavoidable period of unavailability and response time increase due to the sharp increase of load. However, Experiment 2 prevents the overhead of a SO_{infra}, i.e., it does not increase the cost of the infrastructure.

According to our autonomic model and the tactics defined in this evaluation, for the same *High Response Time* event, three actions are possible. This mapping corresponds to the *tactic filter* of our autonomic model (cf. Figure 4). Following the filter scheme, in this example, the three actions would also pass the *context filter*. This filter could be useful if we had a SLA forbidding the degradation of Off_{comp}: in this case only the SO_{infra} action would be applicable. If still, there are several possible eligible actions, the autonomic manager should pick one to be performed. Our experimental analysis shows that each action has its advantages and disadvantages depending on the workload as detailed below:

- SO_{infra} action fosters QoE at the expense of reactivity and infrastructure's cost. It is interesting to apply it in the case of slight workload increase because there is no need for the application to be reactive. Instead, there is a need to continously increase the amout of compute resource (VMs).
- SD_{soft} action fosters infrastructure's cost and reactivity at the expense of QoE. This action makes sense in the case of bursting workload where adding infrastructure resources may take too long to meet sudden changes.
- $(SD_{soft} \parallel SO_{infra}); SU_{soft}$ action fosters reactivity at the expense of infrastructure's cost and punctually



(a) Experiment 1: infrastructure's size (b) Experiment 2: infrastructure's size (c) Experiment 3: infrastructure's size

(d) Experiment 1: Software offering (e) Experiment 2: Software offering (f) Experiment 3: Software offering

(g) Experiment 1: Amount of failed requests (h) Experiment 2: Amount of failed requests (i) Experiment 3: Amount of failed requests

(j) Experiment 1: Average response time (k) Experiment 2: Average response time (l) Experiment 3: Average response time

Figure 8. Experimental results

the QoE. This action should be applied in the case of a workload that increases moderately because there is a need for both reactivity and continuous scaling out of infrastructure resources.

Table III summarizes the three identified criteria: reactivity, QoE and infrastructure's cost. In this table + and − mean positive and negative influence on criteria respectively. The value 0 means both positive and negative influence. In conclusion, the Cloud Administrator could define a strategy by specifying its preferences on the criteria. For instance, (reactivity, QoE, infrastructure's cost) for an application whose workload is similar to the first phase of our scenario.

Table III
(Dis)Advantages of tactics according to criteria

	Reactivity	QoE	Infrast. Cost
SO_{infra}	−	+	−
SD_{soft}	+	−	+
$(SD_{soft} \| SO_{infra}); SU_{soft}$	+	0	−

VI. Related Work

A. Cloud Elasticity Management

Cloud computing promises to completely revolutionize the way to manage resources. Thanks to infrastructure elasticity, resources can be provisioning within minutes to satisfy a required level of QoS formalized by SLAs between different Cloud actors. This can be achieved in a completely adhoc manner by dynamically adjusting resources according to a set of predefined *auto-scaling rules*. Examples of such a kind of approach are implemented by industrials such as Amazon Auto Scaling, Microsoft Azure Auto-scaling Application Block, but also in research work [11] [12].

However, setting up auto-scaling rules in order to respect SLAs while minimizing service cost is not a trivial task since many parameters must be taken into account. For this purpose, an effective Capacity Planning may require the combination of several solutions out of different domains. For example, in order to model systems' performance, Queuing Theory [13] [14] can be applied. Alternatively, Reinforcement Learning [15] [16] and Game Theory [17] [18] might be used respectively to give a more effective performance profiles or to deal with economy equilibrium and thus improve resources requirements estimations. Those techniques can be combined along with operational research techniques such as Constraint and Linear Programming in order to find solutions that respect the constraints (e.g., imposed by SLAs) or even solution that optimizes a specific criterion such as QoS or cost.

With respect to these works, most of them focus only on method accuracy and ignore Cloud technical and conceptual limitations. Some initiatives are interested in solving technical limitations, such as [19] and [20], while conceptual limitations (e.g., economic model) are not addressed. [21] proposes an approach based on vertical scaling to prevent duplicating time for data-base tier. However, the vertical scaling is not provided by all IaaS providers or not as hot scaling, which means the VM needs to be rebooted with potential downtime and significant initialization time. [19] presents a predictive model to absorb the initialization time. The accuracy of prediction method depends on the input window size and the prediction interval but the authors do not detail these values. [20] utilizes fuzzy logic to define more elastic auto-scaling rules but they only focus on qualitative specification of thresholds. Furthermore, most of the auto-scaling solutions consider only low-level performance metrics (e.g., CPU utilization at IaaS level) which is a good indicator for system utilization information, but it cannot truly reflect the QoS provided by SaaS application and neither show if application performances meet user's requirements [22]. Finally, most of the existing works only deal with the infrastructure layer whereas only infrastructure elasticity may not be enough to meet technical or conceptual limitations. Our work, instead, advocates that the software layer must take part in the elasticity process to overcome infrastructure elasticity limitations.

B. Cross-layer Elasticity Management

According to our knowledge, only few studies have focused on expanding system's elasticity capability to the software layer. [23] is interested in architecture-based self-adaptation. They developed *Rainbow* [24], a framework which is similar to a MAPE-K model [2], using software architectures and a reusable infrastructure to support runtime self-adaptation of software systems. For illustration, they rely on the *Znn.com* case study (also known as *ZAP.com* or *Z.com*), a web-based system that serves multimedia news content. The objective is to serve news contents within a reasonable response time while keeping the resources' associated cost under a predefined threshold. In the case of sudden and unanticipated workload increase, to prevent service unavailability due to resource elasticity limitations, *Znn.com* opts to serve minimal textual content instead of a multimedia one. In others words, the adaptation objective is to maximize the QoS and minimize the infrastructure cost by changing the offering of the existing component. The *Znn.com* system's adaptation capability allows to act both on the infrastructure and the software layers. Similarly, our work relies on autonomic computing but we address more concretely the domain of Cloud computing. In particular, our approach goes further since we propose a catalogue of reconfiguration strategies driven by the Cloud administrator preferences.

Similar to our work, [25] introduces a self-adaptation programming paradigm, called *brownout*, allowing applications to robustly face unpredictable runtime variations without over-provisioning. The paradigm is based on optional code that can be dynamically deactivated through decisions based on control theory. We propose a broader approach since we rely on a synergy between infrastructure elasticity and software elasticity.

VII. Conclusion

In this paper, we highlighted the main shortcomings of the standard Cloud elasticity model. We claimed that the software layer (SaaS) can take part in the elasticity process, which could be evidenced with the help of different examples that demonstrate the advantages of using infrastructure and software elasticities together, in a coordinated way, to overcome the current limitations and improve reconfiguration decisions.

First of all, we propose a model for software elasticity which draws inspiration from the vertical/horizontal infrastructure elasticity. It results in four dimensions of elasticity which allow Cloud providers to finely and effectively manage their cloud resources.

Based on a set of experiments performed on a real infrastructure testbed, we provided an experimental analysis and discussion on the use of three elasticity *tactics* (i.e., event-action pairs) under different scenarios in order to provide insights on criteria and preferences

that could drive the autonomic selection of the proper tactic(s) to be applied.

We are currently working on the design of a Domain Specific Language (DSL) for Cloud Elasticity. This language, called *Elascript*, is a scripting language (with an associated graphical formalism) which offers a way to define complex and safe reconfiguration plans simply and concisely. The expressiveness of DSLs can offer more guarantees and extensive static analysis than general languages. In this way, *Elascript* can detect unbearable code sequences that have no meaning for a reconfiguration plan (e.g., trying to execute SU_{soft} and SD_{soft} on the same component in parallel) and then help the Cloud Administrator in its management task.

In the near future, we plan to extend our experiments to numerous tactics involving the four elasticity dimensions mentioned above with more complex architectures and under various types of workload. We thus hope to refine the list of criteria leading autonomic selection of tactics and model the underlying preferences as elasticity strategies reflecting high levels reconfiguration goals.

Acknowledgment

This work is supported by SIGMA Informatique (*http://www.sigma.fr*).

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (*https://www.grid5000.fr*).

References

[1] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in cloud computing: What it is, and what it is not." in *Int. Conf. on Autonomic Computing (ICAC)*, 2013, pp. 23–27.

[2] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[3] G. Galante and L. C. E. d. Bona, "A survey on cloud computing elasticity," in *IEEE Int. Conf. on Utility and Cloud Computing (UCC)*, 2012, pp. 263–270.

[4] M. Mao and M. Humphrey, "A performance study on the vm startup time in the cloud," in *IEEE Int. Conf. on Cloud Computing (CLOUD)*, 2012, pp. 423–430.

[5] H. Jin, X. Wang, S. Wu, S. Di, and X. Shi, "Towards optimized fine-grained pricing of iaas cloud platform," *IEEE Transactions on Cloud Computing*, 2014.

[6] A. Beloglazov, J. Abawajy, and R. Buyya, "Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing," *Future Generation Computer Systems*, vol. 28, no. 5, pp. 755–768, 2012.

[7] A. Kejariwal, "Techniques for optimizing cloud footprint," in *IEEE Int. Conf. on Cloud Engineering (IC2E)*, 2013, pp. 258–268.

[8] A. Lenk and M. Turowski, "Vertical scaling capability of openstack - survey of guest operating systems, hypervisors, and the cloud management platform," in *Service-Oriented Computing ICSOC 2014*, 2014.

[9] Y. Kouki and T. Ledoux, "Rightcapacity: Sla-driven cross-layer cloud elasticity management," *International Journal of Next-Generation Computing*, vol. 4, no. 3, 2013.

[10] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 15:1–15:62, 2012.

[11] R. Han, L. Guo, M. M. Ghanem, and Y. Guo, "Lightweight resource scaling for cloud applications," in *IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGrid)*, 2012, pp. 644–651.

[12] M. Hasan, E. Magana, A. Clemm, L. Tucker, and S. Gudreddi, "Integrated and autonomic cloud resource scaling," in *Network Operations and Management Symposium (NOMS), 2012 IEEE*, 2012, pp. 1327–1334.

[13] M. Mao, J. Li, and M. Humphrey, "Cloud auto-scaling with deadline and budget constraints," in *IEEE/ACM Int. Conf. on Grid Computing (GRID)*, Oct 2010, pp. 41–48.

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

[15] E. Barrett, E. Howley, and J. Duggan, "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 12, pp. 1656–1674, 2013.

[16] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck, "Using Reinforcement Learning for Autonomic Resource Allocation in Clouds: towards a fully automated workflow," in *Int. Conf. on Autonomic and Autonomous Systems*, 2011, pp. 67–74.

[17] F. Teng and F. Magoulès, "A new game theoretical resource allocation algorithm for cloud computing," in *Int. Conf. on Advances in Grid and Pervasive Computing (GPC)*, 2010, pp. 321–330.

[18] D. Ardagna, B. Panicucci, and M. Passacantando, "A game theoretic formulation of the service provisioning problem in cloud systems," in *Int. Conf. on World Wide Web*. ACM, 2011, pp. 177–186.

[19] S. Islam, J. Keung, K. Lee, and A. Liu, "Empirical prediction models for adaptive resource provisioning in the cloud," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 155–162, 2012.

[20] P. Jamshidi, A. Ahmad, and C. Pahl, "Autonomic resource provisioning for cloud-based software," in *Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2014, pp. 95–104.

[21] L. Wang, J. Xu, M. Zhao, Y. Tu, and J. A. B. Fortes, "Fuzzy modeling based resource management for virtualized database systems," in *Int. Symp. on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2011, pp. 32–42.

[22] V. C. Emeakaroha, I. Brandic, M. Maurer, and S. Dustdar, "Low level metrics to high level slas-lom2his framework: Bridging the gap between monitored metrics and sla parameters in cloud environments," in *Int. Conf. on High Performance Computing and Simulation (HPCS)*. IEEE, 2010, pp. 48–54.

[23] D. Garlan, B. Schmerl, and S.-W. Cheng, "Software architecture-based self-adaptation," in *Autonomic computing and networking*. Springer, 2009, pp. 31–55.

[24] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[25] C. Klein, M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez, "Brownout: Building more robust cloud applications," in *Int. Conf. on Software Engineering (ICSE)*. ACM, 2014, pp. 700–711.