HAL open science

Automated Workload Generation for Testing Elastic Web Applications

Michel Albonico, Jean-Marie Mottu, Gerson Sunyé

▶ **To cite this version:**

Michel Albonico, Jean-Marie Mottu, Gerson Sunyé. Automated Workload Generation for Testing Elastic Web Applications. 2016. hal-01317723

HAL Id: hal-01317723
https://hal.science/hal-01317723

Preprint submitted on 18 May 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Automated Workload Generation for Testing Elastic Web Applications

Michel Albonico[†*], Jean-Marie Mottu[‡], and Gerson Sunyé[†]

[†]AtlanMod Team, Inria, Mines Nantes, LINA, Nantes, France, {michel.albonico,gerson.sunye}@inria.fr
[‡]LINA - Université de Nantes, Nantes, France, jean-marie.mottu@univ-nantes.fr
[*]Universidade Tecnológica Federal do Paraná, Francisco Beltrão, Brazil, michelalbonico@utfpr.edu.br

***Abstract*—Web applications are often exposed to unpredictable workloads, which make infrastructure resource management difficult. Resource may be overused when the workload is high and underused when the workload is low. A solution to deal with unpredictable workloads is to migrate web applications to cloud computing infrastructures, where resources vary according to demand. Since resource variations happen during the application life cycle, adaptation tasks must be performed at runtime. The resource variation and the adaptation tasks lead web applications to different states that do not exist in non-elastic infrastructure, which we call elasticity states. We claim that elasticity states may reveal supplementary application errors and that web applications must be tested accordingly. For that, web applications must be lead through elasticity states throughout the test. The natural way to lead web applications through elasticity states is to expose them to workload variations. However, generating the correct workload to induce infrastructure adaptation in a minimal time, without overloading the application, is a difficult task. Aiming to make the workload generation more efficient, we propose a two phases approach that first analyzes the application adaptation behavior and then generates the appropriate workload. We validated our approach by conducting several experiments on Google and Amazon cloud infrastructures. In these experiments, a web application was successfully conducted through the predefined resource variations.**

***Index Terms*—Automated load generation, cloud computing, elasticity, elastic web application, software testing.**

I. Introduction

Web application workloads vary in an unpredictable way. This variation may lead to both, resource overload or underuse, when the workload is high or low. A way to improve the workload variations processing is to migrate web applications to cloud computing infrastructures, which provide elasticity, i. e., resource changes according to demand.

Since resource changes happen during the web application execution, adaptation tasks such as resource addition or load balancer reconfiguration are performed at runtime. Both, resource changes and adaptation tasks, expose applications to states that do not exist in non-elastic infrastructures, which we call *elasticity states*. We claim that elasticity states may introduce or reveal supplementary application errors. First, an application may fail if it migrates from a non-elastic infrastructure to an elastic one, and it is not designed for that. Second, even if the application is designed for executing on elastic infrastructures, errors may be introduced in code that is added to deal with the elasticity states. Finally, errors may also be introduced by failures of infrastructure adaptation tasks.

Since the elasticity states may introduce supplementary errors, we claim that applications must be tested during them. A common way to lead applications through these states is to expose them to workload variations. However, generating the correct workload to induce applications to pass through elasticity states is a difficult task. For instance, a wrong workload variation may overload the application, raise the infrastructure adaptation time, or result on undesired elasticity states. Therefore, we need an approach that generates correct workload variations.

In this paper, we propose an approach that generates appropriated workload variations for leading web applications through predefined elasticity states. This approach has two phases: warm-up and workload generation. In the warm-up phase, we experiment the application and analyze its behavior, gathering preliminary data. Then, the gathered data is used as input in workload generation phase, where the workload variations are generated. Our approach is entirely automated, making the workload generation less laborious and less error-prone.

We conduct two experiments to demonstrate that workload level is the critical point of leading web applications though elasticity states. We also conduct a third experiment, where our approach generates workload variations for leading web applications on Amazon EC2[1], and Google[2]. In both cloud providers, we are able to lead the web applications according to predefined elasticity states.

The rest of the paper is organized as follows. In the next section, we present the major aspects of elastic web applications. Section III introduces our approach. The experiments and their results are described in Section IV. Section V discusses related work. Finally, Section VI discusses the experiments results and concludes.

II. Background

A. Cloud Computing Elasticity Definitions

Different authors [1]–[5] have a common definition for cloud computing elasticity, it is the ability of a cloud in-

[1]https://aws.amazon.com
[2]http://cloud.google.com

frastructure to modify its resource configuration as quickly as possible, according to application demand.

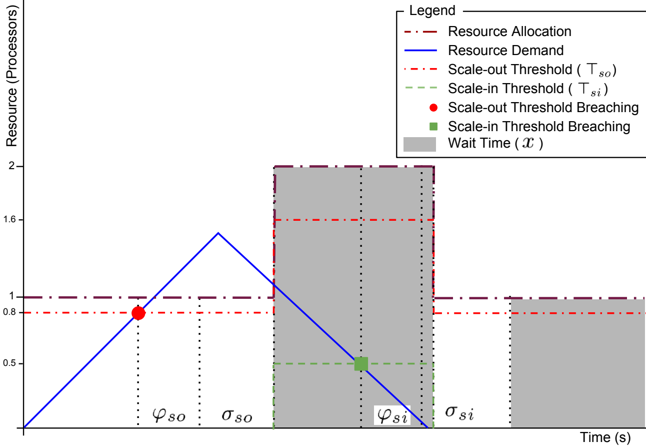


Fig. 1: Representation of cloud computing elasticity.

Figure 1 represents the typical behavior of elastic cloud computing system. In this figure, the resource demand (continuous line) varies over time, first increasing from 0 to 1.5 (demanding 50% more resources than the current allocated processors) and then decreasing to 0. When the resource demand exceeds the *scale-out threshold* ($\top_{so}$), and remains higher during the *scale-out reaction time* (φ_{so}), the cloud elasticity mechanisms assign a new resource to the system. The new resource is available after a *scale-out time* (σ_{so}), the time the cloud infrastructure spends to allocate the new resource. Once the resource is available, the threshold values are updated accordingly.

When the resource demand starts to decrease, breaches the *scale-in threshold* ($\top_{si}$), and remains lower during the *scale-in reaction time* (φ_{si}), the cloud elasticity mechanisms release a resource. However, in the figure the resource release does not start immediately. This because the scale-in threshold is breached in less than the *scale-out wait time* (x_{so}) period, which some cloud providers call *cooldown period*. After this period, the infrastructure needs a *scale-in time* (σ_{si}) to release the resource. As soon as the scale-in begins, the threshold values are updated.

Table I describes the variables presented in Figure 1, which will be used as input for the automatic workload generation presented in Section III-B.

B. Web Application Pressure and Elasticity States

Workload fluctuations drive web applications to different states, related to workload pressure and to infrastructure reconfiguration (elasticity). In the next sections, we introduce these states.

1) Pressure States: Figure 2 presents the states of a web application when exposed to different workload levels.

When the application starts, it is in the *steady* state: the web application is not saturated and answers all requests. During this state, the workload requests (ωr) are proportional to the amount of answered requests (αr).

TABLE I: Cloud Computing Elasticity Variables

Symbol	Name	Unit of Measure
$\top_{si}$	*scale-in threshold*	*% of resource usage*
$\top_{so}$	*scale-out threshold*	*% of resource usage*
ω_{si}	*scale-in workload level*	*transactions per second*
ω_{so}	*scale-out workload level*	*transactions per second*
φ_{si}	*scale-in reaction time*	*seconds*
φ_{so}	*scale-out reaction time*	*seconds*
σ_{si}	*scale-in time*	*seconds*
σ_{so}	*scale-out time*	*seconds*
x_{si}	*scale-in wait time*	*seconds*
x_{so}	*scale-out wait time*	*seconds*

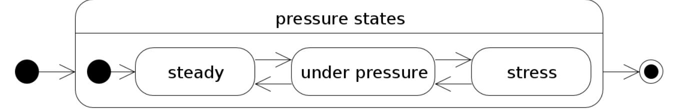


Fig. 2: Web application pressure states.

If the workload increases to a level that drives resource usage close to resource capacity, the application enters the *under-pressure* state. In this state, the web application is on its performance limits and some requests are not answered: ωr becomes bigger than αr.

If the workload decreases during the *under-pressure* state, the application returns to the *steady* state. However, if the workload keeps increasing, the application goes beyond its performance limits and enters the *stress* state. During this state, the application cannot answer most requests: αr tends to 0 transactions per second.

Table II resumes the different pressure states.

TABLE II: Pressure States Notation

State	Notation
steady (S)	$\omega r \cong \alpha r$
under-pressure (UP)	$\omega r > \alpha r$
stress (ST)	$\omega r > \alpha r \wedge \alpha r \rightarrow 0$

It is important to mention that when the application is pressured (*under-pressure* or *stress*), some requests are delayed. Then, the application may remain in the *under-pressure* state even if the workload is drastically reduced. This because the application processes all delayed requests before returning to the *steady* state.

2) Elasticity States: Figure 3 represents the elasticity states: states related to resource changes, which web application is exposed.

At the beginning the web application is exposed to *ready* state: resource configuration is steady. Then, if the web application is exposed for a certain time (φ_{so}) to pressure states that breach the scale-out threshold, the cloud elasticity mechanism starts adding new resource. At this point, the web application is exposed to the *scaling-out* state: period while the resource is added.

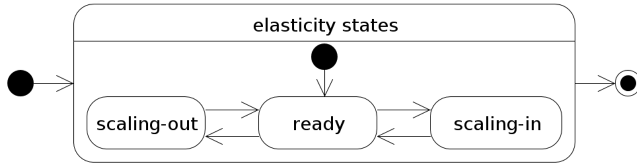


Fig. 3: Elasticity states.

After a *scaling-out*, the web application returns to the *ready* state. Then, if it is exposed for a certain time (φ_{si}) to pressure states that breach the scale-in threshold, the cloud elasticity mechanism start releasing resource. This puts the web application in the *scaling-in* state: period while the resource is released. After that, the web application returns to *ready* state again.

III. Proposed Load Generation Approach

In this section, we describe our approach to automatically generate workload variations for leading web applications through elasticity states. Figure 4 presents the approach workflow, which is divided into two parts: *warm-up* and *workload generation*.

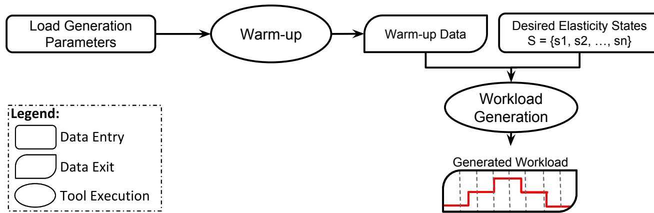


Fig. 4: Workload Generation Workflow.

At the beginning, the user sets the following required parameters for workload generation: application, type of work, desired elasticity states, and *elasticity variables* (see Table I). However, some variables related to elasticity are only known after a first execution. To discover these variables, we use the *warm-up* phase, where the application is stimulated with type of work previously set by user. Then, the *workload generation* phase uses all parameters to generate necessary workload variations.

A. Warm-up

In this section, we describe the warm-up strategies to discover elasticity variables.

1) Thresholds: Thresholds define limits within which an application remains in the *ready* state. Since they can be set for different resources (processor, memory, bandwidth, etc.), all of them must be monitored. While the workload generation phase does not use thresholds as parameters directly, thresholds simplify the discovery of other elasticity variables.

2) Scale-out Workload Level: Discovering scale-out workload level depends on whether scale-out threshold is known, or not. When the threshold is known, application is stimulated with a gradual workload level increasing until resource usage breaches the threshold. Then, current workload level is used as scale-out workload level.

When the threshold is unknown, application is also stimulated with a gradual workload level increasing. However, in this case the workload level is increased for more time, lasting until a scale-out begins. At this moment, we calculate the supposed time of threshold breaching by subtracting the scale-out reaction time (see Section III-A3) from the time the scale-out begins. Then, the workload level at the threshold breaching time is used as scale-out workload level.

3) Scale-out Reaction Time: Scale-out reaction time is discovered during the same execution used to discover the scale-out workload level. It also depends on whether the thresholds are known or not.

When the scale-out threshold is known, we increase the workload until the threshold is breached and then we keep the workload level at the same level until a scale-out begins. Then, the time from the threshold breaching until the scale-out begins is used as scale-out reaction time.

When the threshold is unknown, we set the reaction time to 60 s, the minimal reaction time of most cloud providers. Then, we observe whether some performance degradation happens previously to 60 s before the scale-out begins. If there exists some performance degradation during this period, the time from first degradation until scale-out begins is used as scale-out reaction time. Otherwise, the scale-out reaction time remains at 60 s. In this case, we consider that the performance degradation may be originated by resource exhaustion, which characterizes a threshold breaching.

4) Scale-out Wait Time: We discover the scale-out wait time right after discovering the scale-out workload level and the scale-out reaction time. When the scale-out is completed, we immediately request a new scale-out (doubling the scale-out workload level). Then, after the new scale-out completion, we check the new reaction time.

If it is bigger than the previous one, we consider that there is a wait time that delays the reaction time and use the value of the last reaction time. If it is not, we consider that it is not observable.

5) Scale-in Workload Level: To discover the scale-in workload level we also use two strategies: for known and unknown scale-in thresholds. When the threshold is known, we base our calculation on Liu [6]. The author considers that computing capacity necessary to process a workload is linearly proportional to workload variation. Thus, we calculate the scale-in workload level as follows:

$$\omega_{si} = \left(\frac{\omega_{so}}{\top_{so}}\right) \top_{si} \tag{1}$$

When the scale-in threshold is unknown, we gather the scale-in workload level by experimenting the application. Since this experimentation consists in leading the application to a resource decreasing (scale-in), if the resource is at minimal level, one scale-out is required before the experimentation. In the experimentation, the application is stimulated with a workload level that is decreased from scale-out workload level

multiplied by 2 (current amount of machines) to 0 transactions per second. The workload decreasing rate is calculated as follows: $decreasing\ rate = \omega_{so}/\varphi_{si}$, where the workload variation is divided by the scale-in reaction time. If the workload level reaches 0 transactions per second before scale-in begins, it is kept at this level until the scale-in begins. Then, we calculate the time that supposedly the threshold is breached by subtracting φ_{si} from the time the scale-in begins. After, the scale-in workload level is set with the workload level at calculated threshold breaching time.

6) Scale-in Reaction Time: To discover scale-in reaction time, we need to conduct a new experiment in both of situations, when the scale-in is known or unknown. For that, likewise in scale-in workload level discovering, if the resource is at minimal level, before experiment a scale-out is required.

When the scale-in threshold is known, we stimulate the application with the scale-in workload level (that breaches the threshold) until a scale-in begins. Then, the interval from the beginning of experimentation until scale-in begins is used as scale-in reaction time.

When the scale-in threshold is unknown, the scale-in threshold breaching is not observable. Then, to be sure that the scale-in threshold is breached, we reduce the scale-in workload level to 0 transactions per second, since this is the minimal workload level allowed. When the scale-in begins, we use the interval from the experimentation beginning until scale-in begins as scale-in reaction time.

7) Scale-in Wait Time: To discover the scale-in wait time, we need two sequential scale-in. Therefore, the resource amount must be two times bigger than minimal allocation. In this case, if the resource amount is minimal, two previous scale-out are required to discover the scale-in wait time.

In the scale-in wait time discovery, the application is first stimulated with a workload level that requests one scale-in, and when this scale-in finishes, we immediately reduce the workload level in a way to request a new scale-in. Then, we use the same strategy of scale-out wait time discovering, i. e., if the second reaction time suffers some increasing related to first one, it is used as scale-in wait time.

8) Scale-out and Scale-in Times: We discover the scale-out and scale-in times during the first experiments, which perform resource scaling, i. e., scale-out and scale-in, respectively. These two variables correspond to the time spent by cloud infrastructure to perform each resource scaling. It is important to mention that in the case of web applications, the scale-out time comprehends the resource addition and the inclusion of the new resource on load balancer.

B. Workload Generation

To lead a web application through elasticity states, we generate a *stage-by-stage workload*. A *stage* consists in varying the workload to a *workload stage level* (ωsl), then keep it at this level during a *workload stage time* (ωst). For each desired elasticity state, we calculate a new stage.

For scale-out and scale-in states a stage is calculated as the following formulas:

$$\omega sl_{so} = m\omega_{so} \tag{2a}$$

$$\omega sl_{si} = m\omega_{si} \tag{2b}$$

$$\omega st_{so} = \varphi_{so} + \sigma_{so} + x_{so} \tag{2c}$$

$$\omega st_{si} = \varphi_{si} + \sigma_{si} + x_{si} \tag{2d}$$

where m is the amount of resource required. Indeed, a scale-out allocates one new resource, increasing m of one, while a scale-in releases a resource, decreasing m of one. For instance, if the resource has been scaled out once and a new scale-out is required, the m value is 3. Otherwise, if it has been scaled out once, then a scale-in is required, the m value becomes 0.

Workload stage levels calculation (Equations 2a and 2b) consists in multiplying the workload level (ω_{so}, or ω_{si}) by m. The Equations 2c and 2d are used to calculate the workload stage times, where we sum the reaction time (φ_{so}, or φ_{si}), the scaling time (σ_{so}, or σ_{si}), and the wait time (x_{so}, or x_{si}). In both, workload stage level and workload stage time, the type of variables is chosen according to the state nature, i. e., scale-out, or scale-in.

For ready state, we calculate the stage as follows:

$$\omega sl_r = \neg\Delta\omega sl \tag{3a}$$

$$\omega st_r = manually\ set \tag{3b}$$

As discussed in Section II-B, a ready state happens naturally after scale-out, and scale-in states. Then, to keep this state, the workload stage level remains the same of previous stage (Equation 3a). Since in the ready state we do not need to wait for resource changes, the workload stage time is set by the user, according to his requirements.

C. Load Generator Architecture

Some cloud providers have policies to avoid attacks, such as denial-of-service, for example. In these cases, high number of requests coming from the same source are usually blocked. Since in our approach we generate up to thousand requests per second, this may be interpreted as an attack. To avoid that, we distribute our tool components, following recommendations of Amazon tutorial [7]. Distributing our tool components allows to split the total amount of requests among several components. This distribution prevents false-positive attacks, allows the generation of higher workload levels, in addition to be more realistic than having a unique client.

Figure 5 illustrates our tool architecture, which is composed by a *coordinator*, and several *generators*.

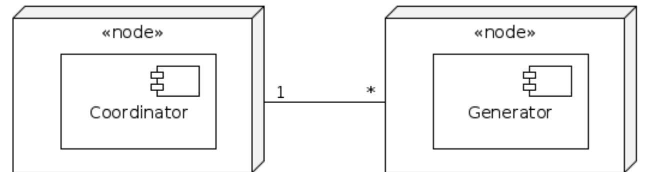


Fig. 5: Our tool architecture.

The coordinator has several roles: front-end, generation of workload variations, synchronization of load tasks executions. Each generator performs exactly the same load tasks received from coordinator. These tasks mimic requests from real-world web applications, such as a page reading, a log-in form sending, etc.

In the current version, our tool is implemented in Java. The workload generation parameters are set using a property file. And, all synchronization is made using *remote method invocation* (RMI). If an *application programming interface* (API) is available, the monitoring tasks are performed using it. Otherwise, a virtual machine used to host the web application is accessed remotely, and common Linux tools are used for monitoring tasks. The remote access is also used if frequent monitoring (e. g., every second) is necessary, since this is usually not allowed by the cloud providers APIs. In the cases where remote access is not allowed, our tool also allows to monitor cloud infrastructures by reading resource status on cloud provider status pages.

IV. Experiments and Results

In this section, we present the experiments and their results. Experiment 1 and 2 verify the influence of different workload levels for scaling resources in and out, respectively. Experiment 3 aims at validating our approach, using it to lead web applications through different elasticity states. Experiments 1 and 2 are only executed on Amazon EC2 cloud provider, and the experiment 3 is executed on both, Amazon EC2 and Google cloud providers.

In both cloud providers, we use small machines from a single geographic region. Amazon machines (t2.small) have 1 virtual CPU (2.5 GHz), 2 GB of memory, and 10 GB of disk. Google machines (g1.small) have 1 virtual CPU (1.38 GHz), 1.7 GB of memory, and 10 GB of disk. The default auto-scaling web hosting services of each cloud provider are used as elastic infrastructure. We consider the default web pages from these services as web applications.[3]

A. Warm-up

Most elasticity variables from Amazon EC2 are previously known, thus only the workload levels, and scaling times must be discovered using the warm-up phase. Some elasticity variables from Google are also previously known, although in our experiments we discover all of them, simulating an infrastructure where no variables are known.

Figure 7 illustrates the workload variation used in warm-up phase to discover the unknown Amazon EC2 elasticity variables. The workload is incremented until the resource usage breaches the scale-out threshold, when the scale-out workload level is gathered. Then, it is kept at same level until a scale-out is performed, and the scale-out scaling time is calculated. After the scale-out, the workload level is immediately reduced to scale-in workload level (calculated according to Section III-A5), and it is kept until a scale-in is performed. At this moment the scale-in scaling time is calculated.

[3]The infrastructure configurations can be reproduced by following the procedures we describe on Google Docs document available at this link: https://goo.gl/k7ZkUI.

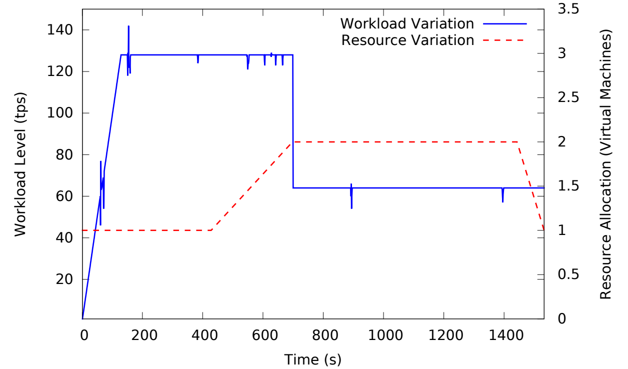


Fig. 7: Warm-up workload variation (Amazon EC2).

Table III presents all Amazon EC2 elasticity variables.

TABLE III: Amazon EC2 Elasticity Variables

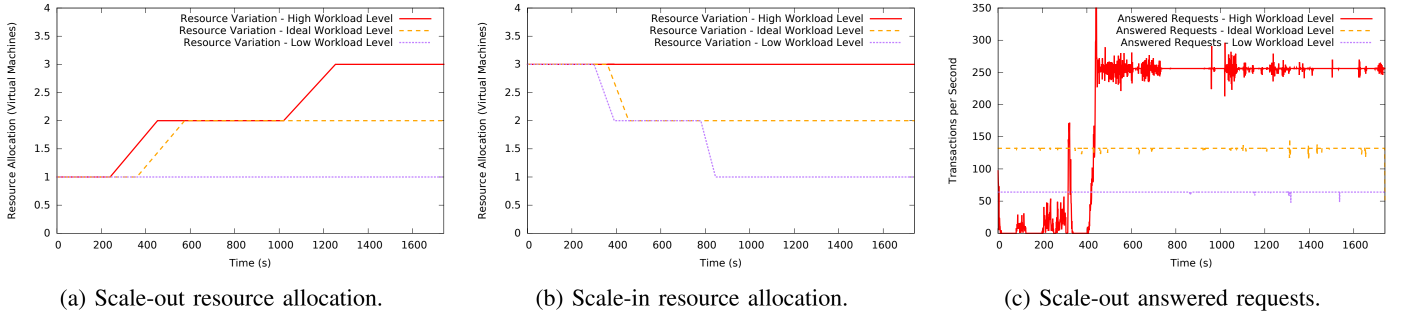
Variable	Value	Known how?
$\top_{si}$	$20\%\ CPU$	set up
$\top_{so}$	$60\%\ CPU$	set up
σ_{si}	$90\ seconds$	set up
σ_{so}	$270\ seconds$	set up
x_{si}	$300\ seconds$	set up
x_{so}	$300\ seconds$	set up
ω_{so}	$128\ tps\ (65\%\ CPU)$	discovered
φ_{so}	$300\ seconds$	discovered
ω_{si}	$30\ tps\ (15\%\ CPU)$	discovered
φ_{si}	$600\ seconds$	discovered

Figure 8 shows the workload variations used to discover Google elasticity variables. Since the thresholds are not known, the workload is incremented until a scale-out began. Then, the reaction time is calculated, and the scale-out workload level is gathered. The workload level is then reduced to scale-out workload level value and kept at this level until the scale-out completes, when we also gather the scale-out scaling time. To know the scale-in reaction and scaling times, the workload level is reduced to zero after the scale-out completion and kept at this level until a scale-in is performed. The next workload variations (from time 1000 s) correspond to scale-in workload level discovering. The workload is increased in a way a new scale-out is triggered, then it is decreased gradually and when reaches zero it is kept at this level until a scale-in begins. Then, the scale-in workload value is gathered. After, the workload is increased in two steps, aiming two sequential scale-out, where is gathered the scale-out wait time. Two sequential scale-in are then requested, to gather the scale-in wait time.

All the values gathered in Google warm-up phase are presented in Table IV.

B. Experiment 1

This experiment attempts to demonstrate that wrong workload levels induce unexpected resource changes. For that, we



(a) Scale-out resource allocation.

(b) Scale-in resource allocation.

(c) Scale-out answered requests.

Fig. 6: Effects of different pressure states on elasticity (Amazon EC2).

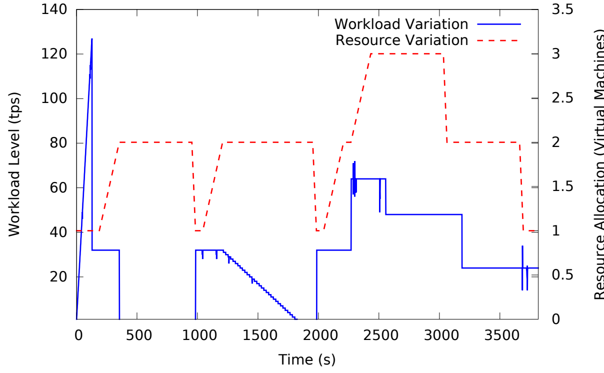


Fig. 8: Warm-up workload variation (Google Cloud).

TABLE IV: Elasticity Variables (Google Cloud)

Variable	Value	Known how?
ω_{so}	$30\ tps\ (resource\ amount = 1)$	Discovered
φ_{so}	$60\ seconds$	Discovered
σ_{so}	$165\ seconds$	Discovered
φ_{si}	$600\ seconds$	Discovered
σ_{si}	$30\ seconds$	Discovered
ω_{si}	$24\ tps\ (resource\ amount = 2)$	Discovered
x_{so}	$60\ seconds$	Discovered
x_{si}	$60\ seconds$	Discovered

use three different workload levels: low ($\omega_{so}/2 = 64\ tps$), ideal ($\omega_{so} = 128\ tps$), and high ($2\omega_{so} = 256\ tps$). The web application is exposed to each workload level on different executions, where the workload level is kept steady during the whole execution. Each execution lasts the double of workload stage time (as discussed in Section III-B) to verify whether the workload level is able to keep the resource amount after the first scale-out.

Figure 6a illustrates the resource changes derived from the different workload levels used in this experiment. When the web application is exposed to the low workload level (64 tps) there is no resource change. With the ideal workload level (128 tps) one resource change is triggered, then the resource remained unchanged until the end of web application execution. The high workload level triggers two resource changes instead of one. At the beginning of Figure 6c, we can see that the high workload level also exposes the web application to a period of stress.

C. Experiment 2

Experiment 2 attempts to demonstrate that correct workload levels are also important for leading web applications through resource scale-in. In this experiment, we also expose the web application to three different workload levels in distinct executions, accordingly to Experiment 1. Although, before each execution we lead the web application to two resource scale-out, wherein for that we consider the ideal workload level of previous Experiment 1. This strategy is used to verify whether some workload levels lead the web application through more than one resource scale-in. Since in this experiment the amount of resource is equal to 3, to request one resource scale-in the m value is equal to 2 (see Section III-B) . Then, the workload levels are calculated as follows: low ($2\omega_{si}/2 = \omega_{si} = 30\ tps$), ideal ($2\omega_{si} = 60\ tps$), and high ($2 * 2\omega_{si} = 120\ tps$).

Figure 6b illustrates all the resource changes triggered for each workload level. When the workload level is high the resource is not scaled in, with the ideal workload level the resource is scaled once, and the low workload level triggers two resource scale-in.

D. Experiment 3

This experiment is conducted to verify whether our approach is able to lead web applications through predefined elasticity states. It is conducted in both, Amazon EC2 and Google Cloud. Whole experiment was conducted according to strategies presented in Section III-B.

In this experiment we try to lead the web application through following sequence of states:

$$S1 = \{SO, SO, SO, SI, SI, SI\} \quad (4)$$

where SO is a scaling-out and SI is a scaling-in state.

Figure 9 shows the relation between the generated workload and resource variations. According to this figure, all resource variations happen according to predefined desired states. The comparison of workload variations to answered requests (Figure 10) shows that the web application is not pressured (*under-pressure* or *stress* pressure states) at any moment.

Figure 11 shows that all desired states are triggered accordingly to workload variations. Comparing workload variation

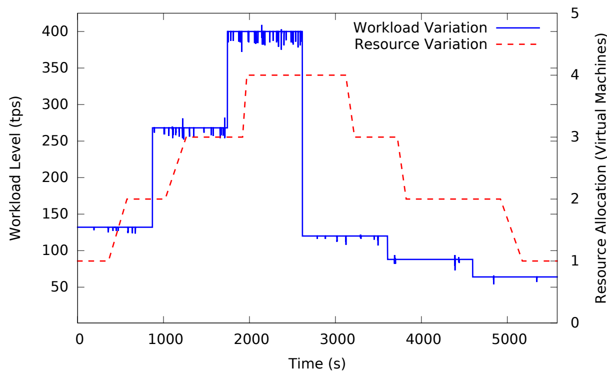


Fig. 9: Leading a web application on Amazon EC2.

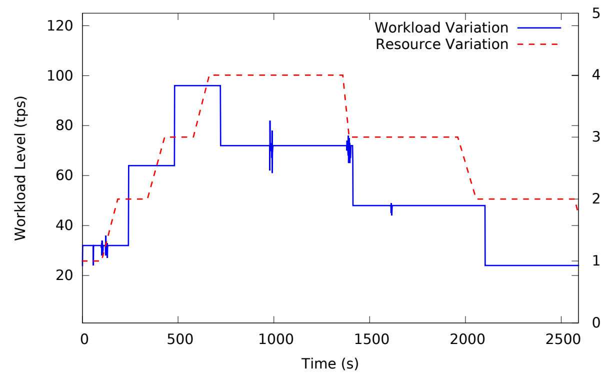


Fig. 11: Leading a web application on Google cloud.

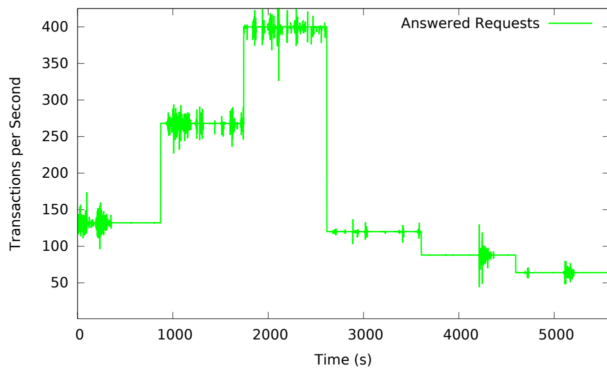


Fig. 10: Answered requests (Amazon EC2).

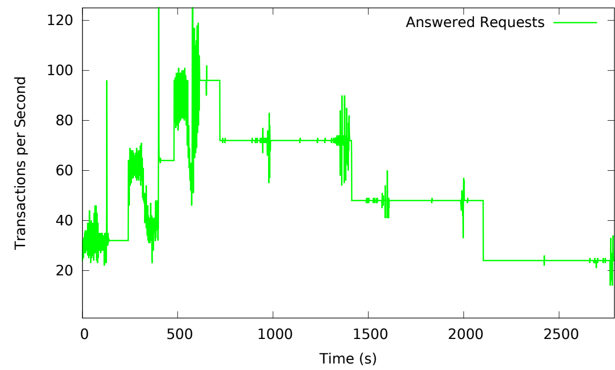


Fig. 12: Answered requests (Google).

and resource variation (Figure 12), we see that each workload variation used to request a resource scale-out puts the web application in an *under-pressure* state. This happens because scale-out threshold is close to resource limits. In this case, when the threshold is breached, the resource is consumed close or beyond its limit, because of this some requests are delayed or rejected. The second and third scale-outs put the web application in *stress* states. Analysing the cloud infrastructure logs we noted the stress states occurred due to cloud elasticity mechanisms inserted a new machine (resource scaled-out) into load balancer before all of its web hosting services be up. As a result, part of requests (those sent to referred machine) could not be answered. This problem was also reported to cloud provider by proper support form[4].

V. Related Work

Meira et al. [8] propose a state machine to represent the behavior of a Database Management Systems (DBMS) exposed to workload variations. Additionally to steady, under pressure, and stress, they also consider warm-up and thrashing states. Warm-up comprehends the application start-up, which is a natural process of any application, then we do not consider it is related to elasticity. The trashing state is never reached in web applications, therefore it is not considered either. We based our pressure states definition on, although we consider web applications as study case.

Gambi et al. have two works that address elasticity testing [9], [10]. In the first work, the authors predict elasticity state transition based on workload variations, and test whether cloud infrastructures react accordingly. Second work presents a tool for automatic testing of cloud-based elastic systems, however this tool does not take into account elasticity states. Our approach is similar to first work in two points: both trigger resource changes without stress, and consider elasticity states and elasticity state transitions. However, the authors classify the elasticity states according to allocated amount of resource, while we also consider as elasticity state the periods where the resource is being changed. We also use elasticity state transitions for a distinct purpose than them, varying the workload in a way to reach these state transitions.

Malkowski et al. [11] focus on controlling elasticity of n-tier applications, such as web applications. This work is similar to ours in two points: it takes care of n-tier applications, and bases its models on previous application runs. Although, it is carried out on cloud infrastructure side, while our approach interacts directly with application (opposite side).

Manasce [12] and Draheim et al. [13] take care of web applications load testing. Manasce only describes the quality of service (QoS) factors and how to conduct load testing for web sites. Draheim et al. propose a simulation of user behavior

[4] https://code.google.com/p/google-compute-engine/issues/detail?id=203

in load testing of web applications. Therefore, neither of them takes care about elasticity.

Other work are related to web performance measurement [14]–[16]. These work only generate web traffic and measure performance variables.

VI. Discussion and Conclusion

In this paper we proposed an approach that automates the workload generation for leading web applications through elasticity states. Our approach gathers the unknown elasticity variables by experimentation, then automatically generates the workload variations necessary to lead the web application through a list of predefined elasticity states.

Results of Experiments 1 and 2 indicates that wrong workload levels lead web applications through unwanted elasticity states. Another important issue to consider is that high workload level put web applications under stress (Figure 6c). Applications under stress are error prone, and errors related to stress are present even in non-elastic infrastructures. Since these errors are not related to elasticity, we claim the elasticity test should avoid them. Otherwise, when any error is found we must discover whether it was originated by elasticity or stress. Same experiment results show that the ideal workload levels drives web applications to desired elasticity state and keeps them at this state. These results confirm our theory that to generate correct workload variation is a critical task on leading web applications through elasticity states.

Experiment 3 results show that in both, Amazon EC2 and Google, our approach generates workload variations that lead the web applications through the predefined elasticity states. In the both cases, the web applications is lead without stress. Considering that, we can say that our approach is able to lead web applications through elasticity states, according to this work objectives.

This paper is the first step towards an automated approach for testing elastic systems. At the moment we lead web applications through predefined elasticity states. Although, we are also interested on aspects directly related to test of elastic applications.

As future work, we intend to adapt our tool to support any cloud infrastructure and to use a workload profiling tool, such as JMeter, for simulating requests. We will also explore functional and non-functional tests of elastic applications. Finally, we intend to apply our approach on more complex web applications architectures.

Acknowledgment

This work is supported by CAPES Fundation (Science Without Borders process 9070–13–3), Ministry of Education of Brazil.

References

[1] D. Agrawal, A. El Abbadi, S. Das, and A. J. Elmore, “Database scalability, elasticity, and autonomy in the cloud,” *DASFAA’11 Proceedings of the 16th international conference on Database systems for advanced applications*, pp. 2–15, Apr. 2011.

[2] L. Badger, T. Grance, R. Patt-Comer, and J. Voas, *Draft Cloud Computing Synopsis and Recommendations*. Nist Special Publication 800-146, 2011.

[3] N. R. Herbst, S. Kounev, and R. Reussner, “Elasticity in Cloud Computing: What It Is, and What It Is Not,” *ICAC*, pp. 23–27, 2013.

[4] M. M. Bersani, D. Bianculli, S. Dustdar, A. Gambi, C. Ghezzi, and S. Krsti, “Towards the Formalization of Properties of Cloud-based Elastic Systems,” in *Proceedings of the 6th International Workshop on Principles of Engineering Service-Oriented and Cloud Systems*, ser. PESOS 2014. New York, NY, USA: ACM, 2014, pp. 38–47.

[5] M. Kuperberg, N. Herbst, J. von Kistowski, and R. Reussner, “Defining and Quantifying Elasticity of Resources in Cloud Computing and Scalable Platforms,” Fakultt fr Informatik (INFORMATIK) Institut fr Programmstrukturen und Datenorganisation (IPD), Tech. Rep., 2011.

[6] H. H. Liu, *Software Performance and Scalability: A Quantitative Approach*, 1st ed. Hoboken, N.J: Wiley, May 2009.

[7] “Best Practices in Evaluating Elastic Load Balancing.” [Online]. Available: https://aws.amazon.com/articles/1636185810492479

[8] J. A. Meira, E. C. de Almeida, and Y. Le Traon, “A State Machine for Database Non-functional Testing,” in *Proceedings of the 18th International Database Engineering & Applications Symposium*, ser. IDEAS ’14. New York, NY, USA: ACM, 2014, pp. 378–379.

[9] A. Gambi, W. Hummer, and S. Dustdar, “Testing elastic systems with surrogate models,” in *2013 1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE)*. IEEE, May 2013, pp. 8–11.

[10] ——, “Automated testing of cloud-based elastic systems with AUToCLES.” IEEE, Nov. 2013, pp. 714–717.

[11] S. J. Malkowski, M. Hedwig, J. Li, C. Pu, and D. Neumann, “Automated control for elastic n-tier workloads based on empirical modeling,” in *Proceedings of the 8th ACM international conference on Autonomic computing - ICAC ’11*. New York, New York, USA: ACM Press, Jun. 2011, p. 131.

[12] D. Menasce, “Load testing of Web sites,” *IEEE Internet Computing*, vol. 6, no. 4, pp. 70–74, Jul. 2002.

[13] D. Draheim, J. Grundy, J. Hosking, C. Lutteroth, and G. Weber, “Realistic load testing of Web applications,” in *Proceedings of the 10th European Conference on Software Maintenance and Reengineering, 2006. CSMR 2006*, Mar. 2006, pp. 11 pp.–70.

[14] D. Mosberger and T. Jin, “httperf - A Tool for Measuring Web Server Performance,” *SIGMETRICS Perform. Eval. Rev.*, vol. 26, no. 3, pp. 31–37, Dec. 1998.

[15] “Apache JMeter.” [Online]. Available: http://jmeter.apache.org/

[16] “Blazemeter: Test automation for devops.” [Online]. Available: https://blazemeter.com/