Controlling Cloud-Based Systems for Elasticity Test Reproduction

Michel Albonico[1(✉)], Jean-Marie Mottu[2], Gerson Sunyé[2], and Frederico Alvares[3]

[1] Federal University of Technology - Paraná, Francisco Beltrão, Brazil
michelalbonico@utfpr.edu.br
[2] Inria/IMT-Atlantique/LS2N, AtlanModels, Nantes, France
{jean-marie.mottu,gerson.sunye}@inria.fr
[3] Inria/IMT-Atlantique/LS2N, Ascola Teams, Nantes, France
frederico.alvares@inria.fr

Abstract. Systems deployed on elastic infrastructures deal with resource variations by adapting themselves, which is error-prone. Therefore, we must test Cloud-Based Systems (CBS) throughout elasticity. Such tests may be re-executed regularly to diagnose and fix CBS bugs, which requires to design tests to execute in a deterministic manner. In this paper, we identify three main challenges that testers face when reproducing elasticity tests: to control the elasticity behaviour, to select specific resources to be deallocated, and to coordinate events parallel to elasticity. Since elasticity tests can last long, we consider the test execution time as a secondary challenge. In this paper, we propose an approach that meets such challenges. Experimental results show that the proposed approach successfully reproduces elasticity-related bugs that face the listed challenges while reducing the execution time.

Keywords: Cloud computing · Elasticity · Elasticity testing
Test reproduction · Speediness

1 Introduction

Elasticity is one of the main reasons that make cloud computing an emerging trend. It allows to allocate or deallocate system resources according to demand [1,2]. Therefore, *Cloud-Based Systems* (CBS) must adapt themselves according to resource variations. These adaptations are not trivial and may affect the CBS execution. According to Bersani et al. [2]:

> "Scaling resources may incur in non-trivial operations inside the system. Component synchronization, registration, and data migration and data replication are just the most widely known examples[...], which may degrade system QoS."

Therefore, to guarantee their quality, we must test CBSs in the presence of elasticity, i.e., *elasticity testing*.

During CBSs development, tests may be regularly re-executed [3] to detect, diagnose, and correct bugs, where each execution must reproduce the same behaviour. This requires to design elasticity tests to be deterministic, which raises four challenges that we have identified: three functional and one non-functional. The *first challenge* (functional) is to repeat the CBS elastic behaviour by managing sequences of resource allocations and deallocations. In this case, the same elastic behaviour leads the CBS to repeat its adaptations over the multiple test executions. As a consequence, this reproduces the issues related to those adaptations, in case such issues have not been corrected. Looking into two CBSs bug tracking, i.e., MongoDB[1] and ZooKeeper[2], we measure that as soon as bugs are related to elasticity, all of them require to be able to repeat the CBS elastic behaviour.

By analysing further MongoDB and ZooKeeper bug tracking, we realize that other elasticity-related bug reproductions require to combine the elastic behaviour along with two further conditions, which we consider as second and third challenges. At least one of them is required 70% and 67% of the MongoDB and ZooKeeper bugs respectively.

The *second challenge* (functional) is to repeat time-based events, where elasticity tests may require to repeat an elastic behaviour, and to synchronize time-based events with specific CBS states. This is required when testing ≈40% of MongoDB and ≈33% of ZooKeeper elasticity-related bugs. An example is the MongoDB NoSQL database bug 7974 [4], where two time-based events are required to reproduce the bug: (1) to create a unique index before one of the MongoDB nodes is removed by a resource deallocation, and (2) to upload a document after a new node is added by a resource allocation.

The *third challenge* (functional) is to repeat a specific CBS components variation, what we call *selective elasticity*. Elasticity tests may require repeat an elastic behaviour, and to remove a specific CBS component during a resource deallocation. This is the case when testing ≈44% of both MongoDB and ZooKeeper elasticity-related bugs. An example is the Apache ZooKeeper bug 2164 [5], which only occurs when the ZooKeeper leader component is removed by a resource deallocation.

Finally, reproducing elasticity tests has a *fourth challenge* (non-functional), to reduce elasticity test execution time. Reducing the execution time can also save money since in cloud computing the billing model is pay-as-you-go, where customers are charged by the time they use resources. One way to do this is to anticipate the reaction to resource demands. Indeed, driving CBSs is time consuming since elastic controllers take a while (at least 60 s) to react to a resource demand. This, summed to the time to allocate or deallocated a resource, result in test executions that last hours, or even days, depending on the length of the elasticity states sequence.

[1] https://www.mongodb.com/.
[2] https://zookeeper.apache.org/.

In this paper, we present an approach and a prototype to address the three functional listed challenges in reproducing elasticity tests: *the reproduction of an elastic behaviour*, *the scheduling of time-based events*, and *the reproduction of CBS components variation*. The approach also addresses the non-functional challenge by anticipating the reaction to resource demand, and as a consequence, accelerating the test reproduction.

To support our claims, we conduct five experiments with two different CBS case studies. The first two experiments aim at measuring the test execution time reduction when using the proposed approach. The other three experiments aim at reproducing three existing elasticity-related bugs by controlling the test reproduction with the proposed approach.

The remainder of this paper is organized as follows. In the next section, we remind the major aspects of cloud computing elasticity, and a previous work of part of the authors in driving CBSs throughout elasticity. Section 3 details the challenges in elasticity test reproduction and introduces the proposed approach. The experiments and their results are described in Sect. 4. Section 5 discusses the related work. Finally, Sect. 6 concludes.

2 Cloud Computing Elasticity

This section defines the main concepts related to Cloud Computing Elasticity, which will help the understanding of our approach.

2.1 Typical Elastic Behavior

Figure 1 presents the typical behavior of elastic cloud computing applications.

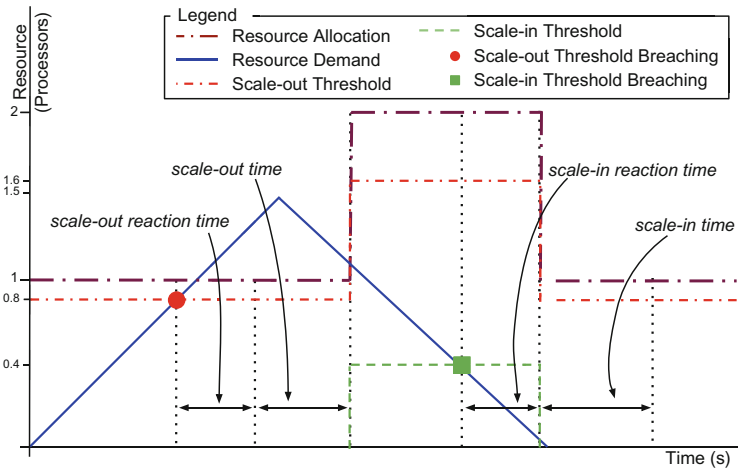


Fig. 1. Typical elastic behavior [6].

In this figure, the *resource demand* (continuous line) varies over time, increasing from 0 to 1.5 then decreasing back to 0. A resource demand of 1.5 means that the application demands 50% more resources than the current allocated ones.

When the resource demand exceeds the *scale-out threshold* and remains higher during the *scale-out reaction time*, the cloud elasticity controller assigns a new resource. The new resource becomes available after a *scale-out time*, the time the cloud infrastructure spends to allocate it. Once the resource is available, the threshold values are updated accordingly. This behavior is similar when considering a resource *scale-in*, respectively. Except that, as soon as the scale-in begins, the threshold values are updated and the resource is becomes unavailable. Nonetheless, the infrastructure needs a *scale-in time* to release the resource.

2.2 Elasticity States

Workload fluctuations lead to resource variations (elasticity) that drive the CBS throughout elasticity-related states. Figure 2 depicts the possible transitions between elasticity states.

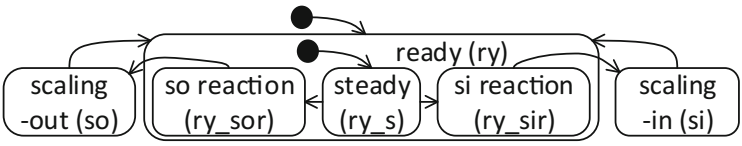


Fig. 2. Elasticity states [6].

At the beginning the CBS is at the *ready* state (*ry*), when the resource configuration is steady (*ry_s* substate). Then, if the CBS is exposed for a certain time (*scale-out reaction time*, *ry_sor* substate) to a pressure that breaches the scale-out threshold, the cloud elasticity controller starts adding a new resource. At this point, the CBS moves to the *scaling-out* state (*so*) and remains in this state while the resource is added. After a *scaling-out*, the CBS returns to the *ready* state, and can move either back to a *scaling-out* state or to a *scaling-in* state (*si*).

2.3 Elasticity Control

When testing CBSs throughout elasticity, testers should be able to drive the CBS in a deterministic way, controlling its elastic behaviour. Thus, they can be more specific and model situations they judge as critical. Furthermore, this can also reduce testing execution time since the elasticity behaviour is specific. In cloud computing, this also means reduction of cost since most of cloud providers use the policy of *pay-as-you-go*, where consumers pay for the time they use resources.

We can categorize *CBS driving* into two groups: (i) direct resource management, and (ii) generation of adequate workload. The first and simplest one (i) interacts directly with the cloud infrastructure, asking for resource allocation and deallocation. The second one (ii) consists in generating adequate workload variations that drive CBS throughout elasticity states, as previously explained in Sect. 1, which reproduces a realistic scenario. The second group is more complex since requires a preliminary step for profiling the CBS resource usage, and calculating the workload variations that trigger the elasticity states.

In a previous work [7], we propose a CBS driving approach that fits in the second group. That approach is based in the assumption that elasticity state transitions occur due to workload variations that eventually breach the thresholds, as illustrated in Fig. 1. We provide further details about this approach in the following paragraphs.

An input workload has three characteristics [8]: *workload type*, *request mix*, and *request intensity*. The *workload type* is the type of requests sent to the CBS, such as *read* and *write* operations. The *mix of requests* is the set of requests associated to a *workload type*. Finally, the *request intensity* is the amount of requests sent to the CBS in a period. Then, given a workload type, the CBS driving approach calculates the *requests intensity variation* that should drive the CBS throughout a pre-set list of elasticity states.

Figure 3 depicts the approach workflow, which has three execution phases: *workload profiling*, *workload calculation*, and *application leading*.

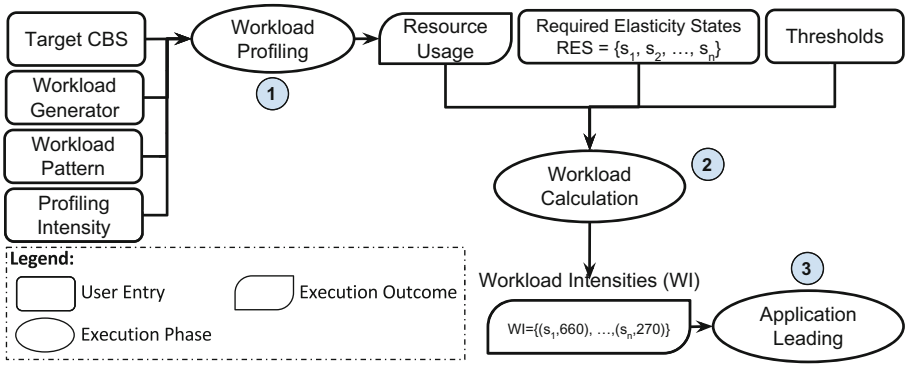


Fig. 3. CBS driving procedure workflow [7].

The *workload profiling* phase has four parameters: the *target CBS*, the *workload generator*, the *workload type*, and the Profiling Intensity (PI). The *target CBS* is the CBS driven throughout elasticity. The *workload generator* is the tool that generates the workload. The *workload type* describes the type or set of requests sent to the CBS. Finally, the PI defines the number of requests per second sent to the CBS during the *workload profiling* phase.

To profile the effect of the *workload* on the CBS, the approach generates the workload according to the workload profiling parameters. Then, it calculates the Average Resource Usage (ARU) for the period, which is the input of the *workload calculation* phase.

In the *workload calculation* phase, the CBS driving approach calculates the *request intensity(ies)* that drive the CBS throughout the *Required Elasticity States (RES)*, which we call *workload intensity(ies)*.

Therefore, to drive a CBS throughout elasticity states we must know which are the workload intensities that breach the scale-out, and the scale-in thresholds, which we call *multipliers*. The scale-out multiplier, denoted by M_{so}, is the workload intensity that breaches the scale-out threshold. The scale-in multiplier, denoted by M_{si}, is the workload intensity that breaches the scale-in threshold.

After discovering the multipliers, the CBS driving approach calculates the workload intensities for each elasticity state in the RES. For scaling states, i.e., scaling-out and scaling-in, the workload intensity must breach a threshold, while for the ready state it must not breach any threshold. However, since scaling states change the amount of resources over time, the amount of allocated resource (AR) is a key parameter. The approach calculates the workload intensities by multiplying it by M_{so} and M_{si}. We call the product of this multiplication as *current multiplier* (CM), where $CM_{so} = M_{so} \cdot AR$ and $CM_{si} = M_{si} \cdot AR$. Such multipliers correspond to the workload intensities that drive the CBS through the scaling-out and the scaling-in states, denoted by WI^{so} and WI^{si}. The workload intensity for ready states ($WI^{ry\text{-}s}$) is calculated as ι percent of CM_{so} ($WI^{ry\text{-}s} = \frac{\iota}{100} \cdot CM_{so}$), where ι is a configurable parameter. Such intensity must lead the resource usage to a level close to CM_{so}, a significant amount of work, but without breaching any threshold.

In the *application leading* phase, we lead the CBS using the calculated workload intensities (WI), which is presented in Algorithm 1. We expose the CBS to each workload intensity until the related elasticity state ends. To identify the elasticity state transitions, the approach monitors the cloud infrastructure periodically.

Algorithm 1. Application Leading.

```
Data: workload intensities WI
monitorElasticity();
foreach p < s, i > ∈ WI do
    while s.isUp do
        generateWorkload(i);
    end
end
```

3 Elasticity Testing Approach

In this section, we first present the challenges in elasticity test reproduction, then we present the overall architecture of our approach and aspects of the prototype implementation.

3.1 Challenges in Elasticity Test Reproduction

Elasticity test reproduction consists in exposing the CBS to the same conditions as previous executions, which should stimulate it to repeat the same behaviour. Then, testers can find CBS bugs, correct them, and then check whether they have been fixed, requiring several runs of failed tests. Another use is to check if changes in the CBS, such as new features, affect its behaviour, or introduce bugs.

To discover which are the conditions that CBSs face, we analyse elasticity-related bugs reported in the bug tracking of two popular CBSs: MongoDB and ZooKeeper. Bug reports have rich information since developers use them to implement tests reproducing bugs. Therefore, these reports reveal the conditions necessary to reproduce elasticity-related bugs, and as a consequence, elasticity tests. The search for elasticity-related bugs has two steps:

1. Select bug reports that contain in their description words that may refer to elasticity, such as: *elasticity*, *scaling*, *adding*, *removing*, *node*, *sync* (for synchronization), and *replic* (for replication).
2. Gather the bug reports whose description refers to resource changes, excluding bug reports where the resource changes do not reflect an elastic behaviour, such as the ones that restart a Virtual Machine (VM) rather than remove or add one.

The two CBS projects use *JIRA*[3] issue tracking to report their bugs. Therefore, for the Step 1, we use the query in Listing 1 to select bug reports related to elasticity. In the query, we change $PROJECT by the project name that corresponds to the CBS, where for MongoDB the project name is *SERVER*, while for ZooKeeper, it is *ZOOKEEPER*. We exclude bug reports whose resolution is *Cannot Reproduce* or *Duplicate*. The first resolution refers to bugs that developers could not reproduce due to either wrong or insufficient information, while the second resolution refers to duplicate bug reports.

Listing 1. Query Used at Step 1.

```
project = "$PROJECT" AND issuetype = Bug AND resolution not
in ("Cannot Reproduce","Duplicate") AND (description ~ "elasticity"
OR description ~ "scaling" OR description ~ "adding"
OR description ~ "removing" OR description ~ "node"
OR description ~ "sync" OR description ~ "replic")
```

Table 1 lists the number of bugs selected at each searching step. MongoDB has 25,780 bugs reported on its bug tracking system, where we find 316 in the first step, and 43 in the second step. ZooKeeper has 2677 bugs reported, where we find 188 bugs in the first step, and 9 in the second step.

The selected bugs reveal three main challenges in reproducing elasticity-related bug, which we consider as elasticity tests reproduction challenges: *elasticity control*, *selective elasticity*, and *event scheduling*. These challenges are functional since the tests cannot be reproduced and the bugs corrected without

[3] https://jira.atlassian.com.

Table 1. Selected bugs in the systematic search.

	Total of bugs	Bugs at step 1	Bugs at step 2
MongoDB	25.780	316	43
ZooKeeper	2.677	188	9

solving them. As usual non-functional challenge of the *speediness* is a concern. It is a requirement to be able to run the numerous tests of such systems. Moreover, since cloud computing's billing model is pay-as-you-go, speediness is a cost concern.

- *Elasticity Control* is the ability to reproduce a specific elastic behaviour. All the selected Elasticity-related bugs occur after a specific sequence of resource allocations and deallocations. Therefore, the challenge is to repeat the CBS elastic behaviour by managing sequences of resource allocations and deallocations.
- *Event Scheduling* is the ability to synchronize events to elasticity states. An event is any interaction with or stimulus to CBS, such as forcing a data increment or to simulate infrastructure failures. The challenge is to identify elasticity states at CBS runtime, and to switch among events according to the elasticity state they are associated.
- *Selective Elasticity* is the ability to remove a specific CBS component. The challenge is to identify and to deallocate the resource that hosts the CBS component that must be removed.
- *Speediness* is the ability of reproducing elasticity tests faster than relying on native cloud computing elasticity controllers. The challenge is to repeat the CBS elastic behaviour anticipating the resource changes.

Table 2 shows the quantity of challenges faced by each CBS bug reproduction. As previously mentioned, all the selected bugs face the elasticity control, where 13 MongoDB (30%) and 3 ZooKeeper (33%) do not face the other challenges for their reproductions. Out of MongoDB bugs, 30 bugs (70%) also face challenges rather than elasticity control, within which 6 (14%) bugs face all the challenges. Out of ZooKeeper bugs, 6 bugs (66%) face further challenges, and 5 (55%) of them face all the challenges.

Table 2. Challenges in bug reproduction.

	Bugs considered	Elasticity control	Selective elasticity	Event scheduling	All	Only elasticity control
MongoDB	*43*	*43*	*19*	*17*	*6*	*13*
ZooKeeper	*9*	*9*	*4*	*3*	*5*	*3*

3.2 Architecture Overview

Figure 4 depicts the overall architecture of our approach. The architecture has four main components, which aim at meeting the elasticity test reproduction challenges: *Elasticity Controller Mock* (ECM), *Workload Generator* (WG), *Event Scheduler* (ES), and *Cloud Monitor* (CM).

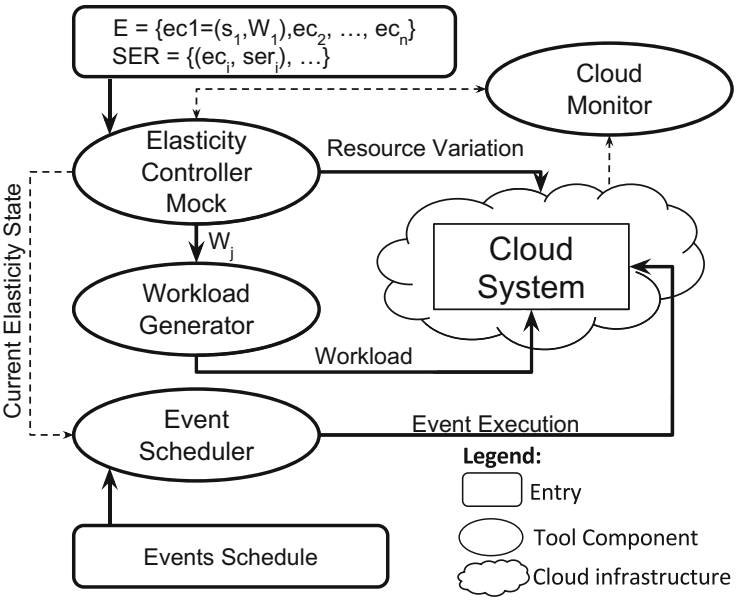


Fig. 4. Overall architecture [6].

The *ECM* simulates the behaviour of the cloud provider elasticity controller, allocating and deallocating determined resources, according to testing needs. It also asks the *WG* to generate the workload accordingly. The role of the *ES* is to schedule and execute a sequence of events in parallel with the other components. Finally, the *CM* monitors the cloud system, gathering information that helps orchestrating the behaviour of the three other components, ensuring the sequence of elasticity states, and their synchronization with the events.

Table 3 summarizes the challenges that each component meets, as we detail in this section.

Elasticity Controller Mock. The ECM is designed to reproduce the elastic behavior. By default, ECM requires as input a sequence of *elasticity changes*, denoted by $E = \{ec_1, ec_2, ..., ec_n\}$, where each ec is a pair that corresponds to an elasticity change. Elasticity change pairs are composed of a required elasticity state (s_i) and a workload (W_i), $ec_i = \langle s_i, W_i \rangle$ where $1 \leq i \leq n$. A workload

Table 3. Challenges met by the architecture components.

Component	Elasticity control	Selective elasticity	Event scheduling	Speediness
ECM	Yes	Yes	Yes	Yes
WG	Yes	No	No	No
ES	No	No	Yes	No
CM	Yes	Yes	No	No

is characterized by an intensity (i.e., amount of operations per second), and a workload type (i.e., set of transactions sent to the cloud system).

ECM reads elasticity change pairs sequentially. For each pair, ECM requests resource changes to meet elasticity state s_i and requests the Workload Generator to apply the workload W_i. Indeed, we have to send the corresponding workload to prevent cloud infrastructure to provoke unexpected resource variations. In particular, it could deallocate a resource that ECM just allocated, because the workload has remained low and under the scale-in threshold.

Rather than waiting for the cloud computing infrastructures for elasticity changes, the ECM directly requests to change the resource allocation (*elasticity control*). Based on both, required elasticity state and workload (elasticity change pair), ECM anticipates the resource changes. To be sure CBS enters the expected elasticity state, ECM queries the CM, which periodically monitors the cloud infrastructure.

The ECM may also lead to a precise resource deallocation (*selective elasticity*). Typically, elasticity changes are transparent to the tester, managed by the cloud provider. To set up the *selective elasticity*, ECM requires a secondary input, i.e., Selective Elasticity Requests (SER). SER is denoted by $SER = \{(ec_1, ser_1), ..., (ec_n, ser_n)\}$, where $ec_i \in E$, and ser_i refers to a *selective elasticity request*. A *selective elasticity request* is a reference to an algorithm (freely written by tester) that gets a resource's ID. When ec_i is performed by ECM, the algorithm referred by ser_i is executed, and the resource with the returned ID is deallocated by ECM.

ECM helps in meeting all of elasticity testing challenges. As earlier explained in this section, it deterministically requests resource variations (*elasticity control* and *selective elasticity*), and helps in ensuring the *event scheduling* providing information of the current elasticity state to the Event Scheduler. As earlier explained in this section, the ECM deterministically requests resource variations (*elasticity control* and *selective elasticity*). In addition, the ECM helps in ensuring the *event scheduling* by providing information of the current elasticity state to the Event Scheduler, and in meeting the speediness by anticipating resource changes.

Workload Generator. The Workload Generator is responsible for generating the workload (W). We base it on our previous work [7], which takes into account a threshold-based elasticity (see Fig. 1), where resource change demand occurs

when a threshold is breached for a while (*reaction time*). Therefore, a workload should result in either threshold breached (for scaling states) or not breached (for ready state), during the necessary time. To ensure this, the Workload Generator keeps the workload constant, either breaching a threshold or not, until a new request arrives.

Considering a scale-out threshold set at 60% of CPU usage, the workload should result in a CPU usage higher than 60% to request a scale-out. In that case, if 1 operation A hypothetically uses 1% of CPU, it would be necessary at least 61 operations A to request the scale-out. On the other hand, less than 61 operations would not breach the scale-out threshold, keeping the resource steady.

The Workload Generator contributes with the Elasticity Controller Mock to meet the *elasticity control* challenge.

Event Scheduler. The ES input is a map associating sets of events to elasticity changes (ec_i), i.e., the set of events that should be sent to the cloud system when a given elasticity change is managed by the ECM. Table 4 abstracts an input where four events are associated to two elasticity changes.

Table 4. Events schedule.

Elasticity change	Event ID	Execution order	Wait time
ec_1	$e1$	1	0 s
	$e2$	2	10 s
	$e3$	2	0 s
ec_2	$e2$	1	0 s
	$e4$	2	0 s

Periodically, the ES polls the ECM for the current elasticity change, executing the events associated to it. For instance, when the ECM manages the elasticity change ec_1, it executes the events e_1, e_2, and e_3. Events have execution orders, which define priorities among events associated to the same state: event e_1 is executed before events e_2 and e_3. Events with the same *execution order* are executed in parallel (e.g., e_2 and e_3). Events are also associated to a *wait time*, used to delay the beginning of an event. In Table 4, event e_2 has a wait time of 10 s (starting 10 s after e_3, but nonetheless executed in parallel). This delay may be useful, for instance, to add a server to the server list a few seconds after the ready state begins, waiting for data synchronization to be finished. The ES meets the *event scheduling* challenge.

Cloud Monitor. The CM helps ECM to ensure *elasticity control* and *selective elasticity*. It periodically requests current elasticity state and stores it in order to respond to the ECM queries, necessary for elasticity control. It also executes

the selective elasticity algorithm of SER, responding to ECM with the ID of the found resources.

3.3 Prototype Implementation

Each component of the testing approach architecture is implemented in Java and communicate with each other through Java RMI. Currently, we only support Amazon EC2 interactions, though one could adapt our prototype to interact with other cloud providers.

Elasticity Controller Mock. The elasticity changes are described in a property file. The entries are set as $\langle key, value \rangle$ pairs, as presented in Listing 2. The key corresponds to the elasticity change name, while the value corresponds to the elasticity change pair. The first part of the value is the elasticity state, and the second part is the workload, divided into intensity and type.

Listing 2. Example of Elasticity Controller Mock Input File (Elasticity Changes).

```
ec1=ready, (1000,write)
ec2=scaling-out, (2000,read/write)
...
ec4=scaling-in, (1500,read)
```

As previously explained, for each entry, the ECM sends the workload parameters to the Workload Generator and deterministically requests the specified resource change. Resource changes are requested through the cloud provider API, which enables resource allocation and deallocation, general infrastructure settings, and monitoring tasks. Before performing an elasticity change, the ECM asks the CM whether the previous elasticity state was reached. The CM uses the Selenium[4] automated browser to gather pertinent information from cloud provider's dashboard Web page.

We use Java annotations to set up selective elasticity requests (SER), as illustrated in Listing 3. A Java method implements the code that identifies a specific resource and returns its identifier as a String type. This method is annotated with metadata that specifies its name and associated elasticity change.

Listing 3. Selective Elasticity Input File.

```
@Selection{name="ser1", elasticity_change="ec4"}
public String select1() {
        ... //code to find a resource ID
        return resourceID;  }
```

Workload Generator. The WG generates the workload according to the parameters received from the ECM (i.e., *workload type* and *intensity*), whereas the workload is cyclically generated until new parameters arrive. It uses existing benchmark tools, setting the workload parameters in the command line.

[4] http://www.seleniumhq.org/.

For instance, YCSB benchmark tool allows three parameters related to the workload: the preset workload profile, the number of operations, and the number of threads. The preset workload profile refers to the workload type, while the multiplication of the two last parameters results in the workload intensity.

Event Scheduler. Event schedules is set in a Java file, where each event is an annotated method, such as the example illustrated in Listing 4. Java methods are annotated with the event identifier, the related elasticity change, the order, and the waiting time. EC periodically polls the ECM to obtain the current elasticity change. Then, it uses Java Reflection to execute the Java methods related to it.

Listing 4. Example of Event Scheduler Input File.

```
@Event{id="e1",elasticity_change="ec1",order="1", wait="0"}
public void event1() { ... }
```

3.4 Prototype Execution

Figure 5 illustrates the prototype execution sequence. This execution starts by the CM component, which interacts with the cloud infrastructure (*Cloud*) to get information that identifies the current elasticity state. Then, the prototype executes the $ec \subset EC$ in parallel to the elasticity states identification. For each $ec \subset EC$, the ECM sends a message to WG, which generates the workload W_i until the ECM sends a message to stop this process. The ECM sends this message when the CM identifies that the current elasticity state has ended. During the workload generation, if es_i is different from *ready*, the ECM changes the resource. Otherwise, it only waits for a given time-frame before moving to the next *ec*. When a new elasticity state begins, the ECM sends a message to the EVs, which leads the execution of all the events related to this state. The prototype repeats this process until the last *ec* ends.

4 Experiments

In this section, we present five experiments. The first two experiments aim at demonstrating the test execution time reduction when using the Elasticity Controller Mock (ECM). The other three experiments aim at controlling the test reproduction of three existing elasticity-related bugs. We conduct all the experiments in the environment described in the next section.

4.1 Experimental Environment

CBS Case Studies. In the experiments, we use two CBS case studies, MongoDB[5] and Apache ZooKeeper[6] (or simply ZooKeeper).

[5] https://www.mongodb.org/.
[6] https://zookeeper.apache.org/.

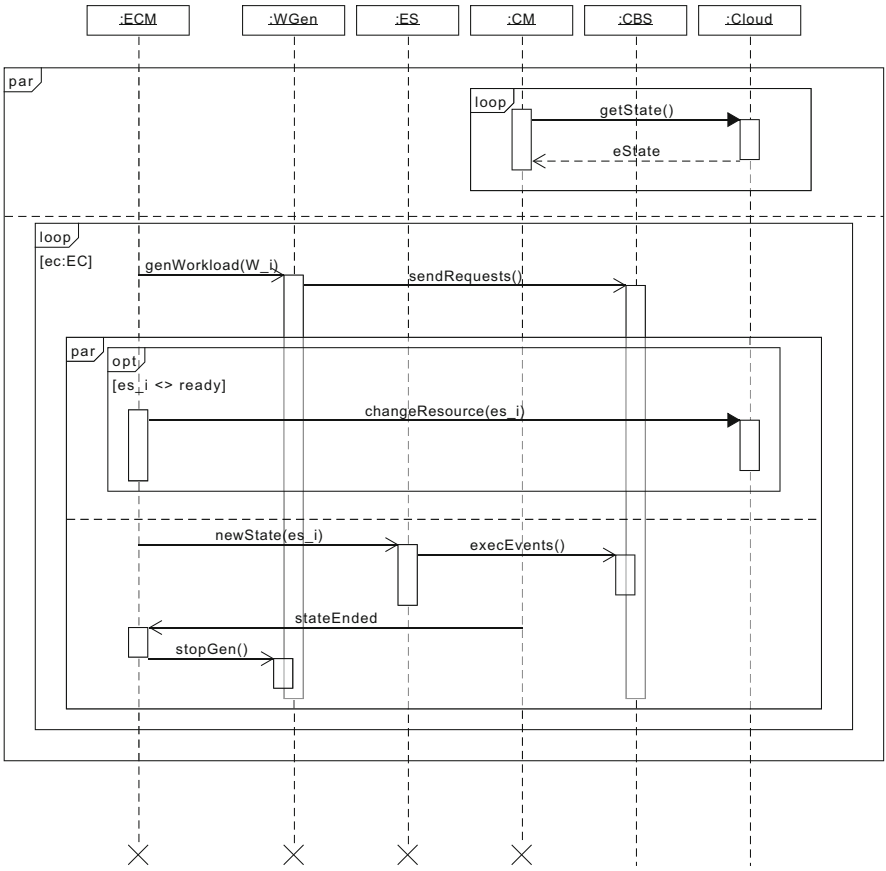


Fig. 5. Prototype execution sequence.

MongoDB is a NoSQL document database. It has three different components: configuration server, MongoS, and MongoD. The configuration server stores meta-data and configuration settings. The MongoS instance is a query router, which ensures load balance, while MongoD instances store and process data.

ZooKeeper is a coordination service for distributed systems. ZooKeeper coordination is intended to be replicated over a set of nodes, called as an *ensemble*. Requests from ZooKeeper clients are forwarded to a single node, the *leader* (which is *elected* using a distributed algorithm). The leader works as a proxy, distributing the request among other nodes called as *followers*. The *followers* keep a local copy of the configuration data to respond to requests.

To generate the workload in the experiments with MongoDB, we use the Yahoo Cloud Serving Benchmark (YCSB) [9], while in the experiments with ZooKeeper, we use an open-source benchmark tool [10].

Cloud Computing Infrastructure. All the experiments are conducted on the commercial cloud provider Amazon Elastic Cloud Compute (EC2), where we set *scale-out* and *scale-in* thresholds as 60% and 30% of CPU usage, respectively. Since the threshold values are not critical for the experiment goals, we set them in an arbitrary manner. We choose a scale-out threshold value as 60% of CPU since it should not result in CBS stress. This threshold value also makes it possible to reduce the execution cost since the workload generation can be executed on a single medium machine (*m3.medium*[7], with a 2.6 GHz vCPU, 3.75 MB of memory, and 4 GB of disk). We set the scale-in threshold value as half of the scale-out threshold value.

In the experiment with MongoDB, the MongoS instance is deployed on a large machine (*m3.large*, with 2 vCPUs of 2.6 GHz, 7.5 MB of memory, and 32 GB of disk), while the other instances are deployed on medium machines. In the experiments with ZooKeeper, every node is deployed on a medium machine.

4.2 Speediness Experiment

In this second set of experiments, we verify whether the Elasticity Controller Mock (ECM) reduces test execution time. In the experiment we lead two CBS case studies, MongoDB and ZooKeeper, through an elasticity states sequence that covers all the possible elasticity state transitions. This is the elasticity states sequence: *ready*, *scaling-out*, *ready*, *scaling-in*, and *ready*. This leading is done in two ways, by using the Elasticity Controller Mock, and by using the Amazon EC2 elasticity controller. The workload pattern used in this experiment is only read operations, which keeps the data size unchanged along the experiment execution.

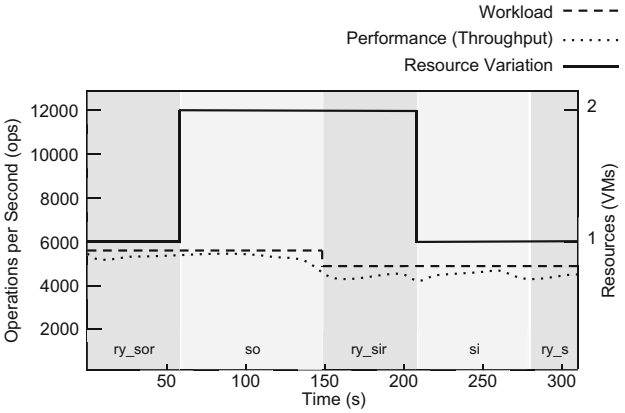
Speediness Considering ZooKeeper. For the ZooKeeper, we consider the following elasticity changes sequence:

$$E = \langle ry\_sor, \langle 5800, r\rangle\rangle, \langle so, \langle 5800, r\rangle\rangle, \langle ry\_sir, \langle 5000, r\rangle\rangle, \\ \langle si, \langle 5000, r\rangle\rangle, \langle ry, \langle 5000, r\rangle\rangle$$

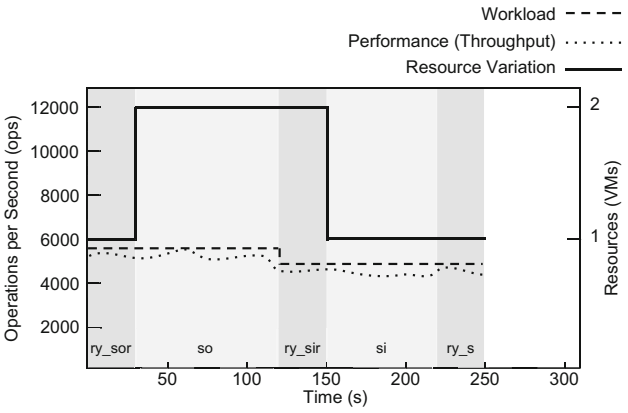
Aiming at accelerating the elasticity changes sequence execution, the first two ready states correspond to *ry_sor* and *ry_sir* sub-states (see Fig. 2), according to the next scaling state in the sequence. Thus, as soon as the CBS enters a ready state, there is a threshold breaching that triggers the next scaling state. The last ready state corresponds to a ry_s sub-state, where none of thresholds is breached. When executing the sequence by using the native Amazon EC2 elasticity controller, we set the scale-out and scale-in reaction times (see Fig. 1) as the minimum allowed, i.e., 60 s. In the ECM, we set this as 30 s, half of the minimum allowed by the Amazon EC2 elasticity controller. In both cases, the last ready state lasts 30 s.

Figures 6(a) and (b) present ZooKeeper performance results when using Amazon EC2 and the ECM.

[7] https://aws.amazon.com/fr/ec2/itype/.



(a) Amazon EC2 Elasticity Controller



(b) Elasticity Controller Mock (ECM)

Fig. 6. ZooKeeper performance.

The execution by using the ECM lasts ≈250 s, while with Amazon EC2 the execution lasts ≈310 s. This difference of (2×30) s is due to our shortest reaction time after the threshold breaching. Both executions show a similar performance variation (doted line), which corresponds to the applied workload (dashed line): it starts by ≈5800 ops and keeps at this level until the end of the scaling-out state, goes down to ≈5000 ops from the second ready state until the end of the execution. When using the ECM, the average performance varies less than 1% compared the execution by using the Amazon EC2 elasticity controller. This value is insignificant, and can be associated to external factors, such as network latencies.

Speediness Considering MongoDB. For the MongoDB, we consider the following elasticity changes sequence:

$$E = \langle ry\_sor, \langle 1500, r\rangle\rangle, \langle so, \langle 1500, r\rangle\rangle, \langle ry\_sir, \langle 1000, r\rangle\rangle, \\ \langle si, \langle 1000, r\rangle\rangle, \langle ry, \langle 1000, r\rangle\rangle$$

The test sequence execution time is ≈330 s when using Amazon EC2 and ≈270 s when using the ECM. The difference corresponds to the (2×30) s shortest reaction time after the threshold breaching. We measured a performance difference of less than 2% comparing the use of our approach with the default Amazon EC2.

4.3 Test Reproduction Experiment

In this section, we describe the use of our approach to reproduce the three bugs, and compare the results to reproduction attempts without our approach. We do not explain in details the setup of reproductions without the proposed approach, though in such executions the *elasticity control* is managed. Indeed, reproducing elasticity is a native feature of cloud computing infrastructures, and we just drive CBS through required elastic behavior using our approach [7].

Selected Bugs. Table 5 summarizes the challenges in the reproduction of the three selected bugs.

Table 5. Challenges in reproducing the three selected bugs.

Bug	Feature		
	Elasticity control	Selective elasticity	Event scheduling
$MongoDB - 7974$	Yes	Yes	Yes
$ZooKeeper - 2164$	Yes	Yes	No
$ZooKeeper - 2172$	Yes	No	Yes

The selected bugs cover all the possible combinations of challenges, constrained by the mandatory presence of *elasticity control*, and the need of at least one of the other challenges. We do not attempt to reproduce any bug that only faces *elasticity control* challenge since one could reproduce the required elastic behavior using our elastic control approach [7].

MongoDB Bug 7974**.** This bug affects the MongoDB versions 2.2.0 and 2.2.2, when a secondary component of a MongoDB replica set[8] is deallocated. Indeed, in a MongoDB replica set, one of the components is elected as primary member, which works as a coordinator, while the others remain as secondary members.

[8] https://docs.mongodb.com/replica-set.

To reproduce this bug, we must follow a specific elastic behavior: initialization of a replica set with three members, deallocation of a secondary member, and allocation of a new secondary member. Therefore, the second step of the elastic behavior requires the deallocation of a precise resource, one of the secondary members. The bug reproduction also requires two events synchronized to elasticity changes. Right after the secondary member deallocation, we must create a unique index, and after the last step of the elastic behavior, we must add a document in the replica set.

In conclusion, the reproduction of this bug faces all the challenges that we consider in this paper: *elasticity control*, *selective elasticity*, and *event scheduling*.

ZooKeeper Bug . This bug is related to ZooKeeper version 3.4.5 and concerns the leader election. According to the bug report[9], in an ensemble with three nodes, when the node running the leader shuts down, a new leader election starts and never ends.

The reproduction of this bug must follow a precise sequence: initialization (allocation of the first node), followed by the allocation of two nodes and the deallocation of the leader node. The main difficulty of reproducing this bug is that when ZooKeeper is deployed on three nodes, the deallocated node is not necessarily the leader. The problem is that during a scale-in, Amazon EC2 removes either the newest or the oldest node and cannot reproduce the bug straightforwardly. In conclusion, the reproduction of this bug faces two challenges: *elasticity control* and *selective elasticity*.

ZooKeeper Bug 2172**.** This bug is related to ZooKeeper version 3.5.0. According to the bug report[10], when a third node is added to a ZooKeeper ensemble, the system enters an unstable state and cannot recover.

After a thorough analysis of the available logs, we understand that the bug occurs when a leader election starts right after the allocation of a third node. More precisely, when a new node joins the ensemble, there is a data synchronization with the leader. Then, if the data is not already synchronized at the moment of the leader election, the bug occurs.

The reproduction of this bug requires a simple elastic behaviour: the allocation of one initial node, and then the allocation of two more nodes. However, this sequence alone does not reproduce the bug: we need to be sure that the leader election starts before the end of the data synchronization process. We can force this by increasing the data amount through an event synchronized with the completion of the third node allocation.

The reproduction of this bug faces two challenges: *elasticity control* and *event scheduling*.

[9] https://issues.apache.org/jira/ZK2164.
[10] https://issues.apache.org/jira/ZK2172.

4.4 Bug Reproductions

MongoDB-7974 Bug Reproduction. This bug reproduction has been already described in our previous paper [6]. To reproduce MongoDB bug 7974 using our approach, we first manually create the MongoDB replica set, composed of three nodes. Then, we set up the following sequence of elasticity changes, which should drive MongoDB through the required elastic behavior:

$$E = \langle ry_1, \langle 4500, r\rangle\rangle, \langle si_1, \langle 1500, r\rangle\rangle, \\ \langle ry_2, \langle 3000, r\rangle\rangle, \langle so_1, \langle 4500, r\rangle\rangle, \langle ry_3, \langle 4500, r\rangle\rangle$$

Since we must deallocate a secondary member of MongoDB replica set at elasticity change ec_2, it is associated to a selective elasticity request (SER). The SER queries MongoDB replica set's members, using MongoDB shell method *db.isMaster*, until finding a member that is secondary.

In parallel to the elasticity changes, we set up two events, $e1$ and $e2$, which respectively create a unique index, and insert a new document in the replica set. The $e1$ is associated to elasticity change ec_3, a ready state that follows the scaling-in state where a secondary member is deallocated. The $e2$ is associated to elasticity change ec_5, the last ready state. Both events are scheduled without waiting time (Table 6).

Table 6. MongoDB-7974 event schedule [6].

Elasticity change	Event ID	Execution sequence	Wait time
ec_3	$e1$	1	0 s
ec_5	$e2$	1	0 s

We repeat the bug reproduction for three times. After each execution, we look for the expression *"duplicate key error index"* in the log files. If the expression is found, we consider the bug is reproduced.

Table 7 shows the result of all the three executions, either using our approach or not. All the attempts using our approach reproduce the bug, while none of the attempts without our approach do it.

Table 7. MongoDB-7974 bug reproduction results [6].

Reproduction	Reproduced	Not Reproduced
With our approach	3	0
Without our approach	0	3

For the executions without our approach, we force MongoDB to elect the intermediate node (in the order of allocation) as primary member[11], what can

[11] https://docs.mongodb.com/force-primary.

occur in a real situation. In this scenario, independent of scale-in settings, cloud computing elasticity controllers always deallocate a secondary member, since Amazon EC2 only allows to deallocated the oldest or newest nodes. This is because we want to see the effect of event synchronization. Therefore, we assure the elastic behaviour is the required to reproduce the bug. Even though we force the reproduction of the required elastic behaviour, this bug still needs the event executions, which must be correctly synchronized. This is the reason the bug is not reproduced without our approach.

ZooKeeper-2164 Bug Reproduction. To reproduce this bug, we translate and complete the scenario (Sect. 4.3) into the following sequence of elasticity changes:

$$E = \langle ry_1, \langle 3000, r\rangle\rangle, \langle so_1, \langle 5000, r\rangle\rangle, \langle ry_2, \langle 5000, r\rangle\rangle, \\ \langle so_2, \langle 10\ 000, r\rangle\rangle, \langle ry_3, \langle 10\ 000, r\rangle\rangle, \langle si_1, \langle 5000, r\rangle\rangle$$

The sequence of elasticity changes first initializes the cloud system with one node, then it requests two scale-out. Once the three nodes are running, the sequence requests a scale-in.

To discover the leader node, we write a SER that is associated to the last elasticity change $e6$ ($\langle si_1, \langle 5000, r\rangle\rangle$). The SER method connects to every Zookeeper node and executes ZooKeeper command named `stat`. This command describes, among other information, the node execution mode: leader or follower.

The sequence of elasticity states, including a selective elasticity, is supposed to reproduce the bug. To verify whether the failure occurs, we write a test oracle, which is implemented in JUnit [11]. It is run after the last elasticity change ($\langle si_1, \langle 5000, r\rangle\rangle$), and repetitively searches for a leader until it is found or the timeout is reached. In the first case, the verdict is *pass*, what means the bug is reproduced and observed. Otherwise, the verdict is *fail*.

As well as in the first experiment, we use two different setups to execute this experiment: with our approach, and without our approach. We repeat the experiment three times for each setup.

Since the selective elasticity is one of the challenges for this bug reproduction, when executing without our approach, we try to reproduce a real scenario, where every node can be elected as a leader. Therefore, we force ZooKeeper to elect a different node as the leader at each execution: the newest, the oldest, then the intermediate node. Then, we use Amazon EC2 to deallocate a node. Its policy is to deallocate either the newest or the oldest node, it is not possible to deallocate the intermediate node. Hence, during two executions we can ask Amazon EC2 to deallocate the leader, but not during the third one.

Table 8 summarizes the results. When using our approach, all the three test executions pass, demonstrating the ability of our testing approach to deterministically reproduce the bug. In contrast, only two executions without our approach pass, the ones where the leader is the newest or the oldest node. Therefore, without our approach the bug was not reproduced deterministically.

Table 8. ZooKeeper-2164 bug reproduction results.

Reproduction	Pass verdicts	Fail verdicts
With our approach	3	0
Without our approach	2	1

ZooKeeper-2172 Bug Reproduction. We create the following sequence of elasticity changes to reproduce this bug (Sect. 4.3):

$$E = \langle ry_1, \langle 3000, r\rangle\rangle, \langle so_1, \langle 5000, r\rangle\rangle, \\ \langle ry_2, \langle 5000, r\rangle\rangle, \langle so_2, \langle 10\ 000, r\rangle\rangle, \langle ry_3, \langle 10\ 000, r\rangle\rangle$$

According to the bug log files, the bug occurs when the leader election starts before the end of the data synchronization between the third node and the previous leader. Thus, the test sequence must ensure that the data synchronization process is longer than the delay needed to start a new election, which is about 10 s according to the log files. Forcing the data synchronization to take long enough, we create an event schedule to associate an event $e1$ to the state so_2, as described in Table 9. The $e1$ requests a data increasing to an amount that should take longer than 10 s to synchronize. Since this experiment uses Amazon $m3.large$ machines, which have a bandwidth of 62.5 MB/s, the data amount must be ≈625 MB of data.

Table 9. ZooKeeper-2172 event schedule.

Elasticity change	Event ID	Execution sequence	Wait time
ec_4	$e1$	1	0 s

We use the test oracle as for the bug 2164, which is associated to the last *ready* elasticity state which is not supposed to be able to elect a leader before the timeout. Table 10 summarizes the experiment execution. In all three executions, the test verdict is *pass*, meaning that the testing approach reproduces the bug successfully. Since Amazon EC2 cannot manage natively the scheduling of events synchronized with elasticity states, it cannot reproduce the bug deterministically.

Table 10. ZooKeeper-2172 bug reproduction results.

Reproduction	Pass verdicts	Fail verdicts
With our approach	3	0
Without our approach	0	3

5 Related Work

Several research efforts are related to our approach in terms of elasticity control, selective elasticity, and events scheduling. The work of Gambi et al. [8,12] addresses elasticity testing. The authors predict elasticity state transition based on workload variations and test whether cloud infrastructures react accordingly. However, they do not focus on controlling elasticity and cannot drive cloud application throughout different elasticity states.

Banzai et al. [13] propose D-Cloud, a virtual machine environment specialized in fault injection. Like our approach, D-Cloud is able to control the test environment and allows testers to specify test scenarios. Test scenarios are specified in terms of fault injection and not on elasticity and events, as in our approach.

Yin et al. [14] propose CTPV, a Cloud Testing Platform Based on Virtualization. The core of CTPV is the private virtualization resource pool. The resource pool mimics cloud infrastructures environments, which in part is similar to our elasticity controller. CTPV differs from our approach in two points: (i) it does not use real cloud infrastructures and (ii) it uses an elasticity controller that does not anticipate resource demand reaction.

Vasar et al. [15] propose a framework to monitor and test cloud computing web applications. Their framework replaces the cloud elasticity controller, predicting the resource demand based on past workload. Contrary to our approach, they do not allow to control a specific sequence of elasticity states or events.

Li et al. [16] propose Reprolite, a tool that reproduces cloud system bugs quickly. Similarly to our approach, Reprolite allows the execution of parallel events on the cloud system and on the environment, but it does not focus on elasticity, one of our main contributions.

6 Conclusion

In this paper, we proposed an approach to reproduce elasticity tests in a deterministic manner. This approach meets four challenges: elasticity control, selective elasticity, event scheduling, and execution time reduction.

We used this approach to reduce the execution time when driving two CBSs, ZooKeeper and MongoDB, throughout an elasticity state sequence that covers all the elasticity state transitions. We compare these executions to the ones without the proposed approach by measuring the CBS performance throughout the executions. The performance in both executions does not present a significant variation, which indicates that the approach reduces the execution time without compromising the CBS behaviour.

We also used the approach to control the reproduction of three bugs of those two CBSs. Indeed, the bugs cannot be deterministically reproduced with state-of-the-art approaches. This also indicates that execution time reduction does not hamper such bug reproductions.

As testing is not only about reproducing existing bugs, but also diagnosing them. An evolution for the proposed approach is to generate different test scenarios combining elasticity state transitions, workload variations, selective elasticity, and event scheduling. Another perspective could be to further investigate the impacts of speediness. In fact, in this paper, we proposed a way to accelerate elasticity test executions, but it lacks a deeper investigation on how fast we can reproduce elasticity tests without compromising them.

References

1. Herbst, N.R., Kounev, S., Reussner, R.: Elasticity in Cloud Computing: what it is, and what it is not. In: ICAC (2013)
2. Bersani, M.M., Bianculli, D., Dustdar, S., Gambi, A., Ghezzi, C., Krstić, S.: Towards the formalization of properties of Cloud-based elastic systems. In: Proceedings of PESOS 2014, New York, NY, USA. ACM (2014)
3. Engstrom, E., Runeson, P., Skoglund, M.: A systematic review on regression test selection techniques. Inf. Softw. Technol. **52**, 14–30 (2010)
4. Mongodb bug 7974: Suppress stack trace on replication errors. (https://jira.mongodb.org/browse/SERVER-7974). Accessed 29 May 2017
5. Zookeeper bug 2164: Fast leader election keeps failing. (https://issues.apache.org/jira/browse/ZOOKEEPER-2164). Accessed 08 Feb 2017
6. Albonico, M., Mottu, J.M., Sunyé, G., Alvares, F.: Making Cloud-based systems elasticity testing reproducible. In: Proceedings of the 7th International Conference on Cloud Computing and Services Science, CLOSER 2017, pp. 495–502, Porto, Portugal, 24–26 April 2017
7. Albonico, M., Mottu, J.M., Sunyé, G.: Controlling the elasticity of web applications on Cloud Computing. In: Proceedings of the 31st SAC. ACM (2016)
8. Gambi, A., Hummer, W., Truong, H.L., Dustdar, S.: Testing elastic computing systems. IEEE Internet Comput. **17**, 76–82 (2013)
9. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking Cloud serving systems with YCSB. In: Proceedings of SoCC 2010, New York, NY, USA. ACM (2010)
10. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: wait-free coordination for internet-scale systems. In: 2010 USENIX, Boston, MA, USA (2010)
11. Gamma, E., Beck, K.: Junit: a cook's tour. Java report (1999)
12. Gambi, A., Hummer, W., Dustdar, S.: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE (2013)
13. Banzai, T., Koizumi, H., Kanbayashi, R., Imada, T., Hanawa, T., Sato, M.: D-Cloud: design of a software testing environment for reliable distributed systems using cloud computing technology. In: Proceedings of CCGRID 2010, Washington, USA (2010)
14. Yin, L., Zeng, J., Liu, F., Li, B.: CTPV: a Cloud testing platform based on virtualization. In: The Proceedings of SOSE 2013 (2013)
15. Vasar, M., Srirama, S.N., Dumas, M.: Framework for monitoring and testing web application scalability on the Cloud. In: Proceedings of WICSA/ECSA Companion, NY, USA (2012)
16. Li, K., Joshi, P., Gupta, A., Ganai, M.K.: ReproLite: a lightweight tool to quickly reproduce hard system bugs. In: Proceedings of SOCC 2014, New York, NY, USA (2014)