

A DSL-based Approach for Elasticity Testing of Cloud Systems

Michel Albonico, Amine Benelallam, Jean-Marie Mottu, Gerson Sunyé

▶ **To cite this version:**

Michel Albonico, Amine Benelallam, Jean-Marie Mottu, Gerson Sunyé. A DSL-based Approach for Elasticity Testing of Cloud Systems. Domain-Specific Model Workshop, Oct 2016, Amsterdam, Netherlands. 10.1145/3023147.3023149 . hal-01437137

HAL Id: hal-01437137
https://hal.science/hal-01437137

Submitted on 19 Jan 2017

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.

A DSL-based Approach for Elasticity Testing of Cloud Systems

Michel Albonico[1,2], Amine Benelallam[1], Jean-Marie Mottu[1], and Gerson Sunyé[1]

[1]AtlanModels team (Inria, Mines Nantes, Lina), France

Email: {amine.benelallam, jean-marie.mottu, gerson.sunye}@inria.fr

[2]Federal Technological University of Paraná (UTFPR), Brazil

Email: michelalbonico@utfpr.edu.br

ABSTRACT

Elastic cloud systems automatically respond to workload changes by (de-)allocating resources according to a configuration specification. Testing such systems requires effort from the tester. In particular, the tester is brought to specify the sequence of resource variations he/she is willing to test the system under, and then, drive the system through this specific sequence of variations while the test is being executed. In this paper, we propose a Domain Specific Language (DSL) aiming at reducing the tester's effort to write and execute elasticity testing. Our DSL abstracts test case specification from the cloud provider's libraries, making our language portable. The DSL compiles into executable code that implicitly orchestrates the execution of elasticity testing. In our preliminary results, we show that our approach reduces the amount of words to specify test cases w.r.t. dedicated libraries. It also shows how much this improvement scales when running a test on multiple cloud providers.

1. INTRODUCTION

One of the core design principles in cloud-based systems (a.k.a. cloud systems) is resource auto-scaling, also known as *elasticity*. It is defined as the ability of a cloud infrastructure to vary its resource configuration according to demand (in particular allocating or deallocating resources) [4, 7, 15, 8]. However, due to dynamic resource allocation, a new class of issues may arise and cause the cloud system to fail [6]. In response to this matter, several work [11, 16, 10, 13, 12, 6] have been proposed to address elasticity testing of cloud systems.

Gambi et al. [13] propose a conceptual framework for testing elastic cloud systems that helps tester to manage four activities: test case generation, test execution, data analysis, and test evolution. In this paper, we focus on the first two activities. While, the first activity generates test cases in order to attend tester's specification, the second activity runs test cases against the cloud system.

These two activities require tester's effort to design and implement test cases, and to deploy and configure cloud system repeatedly to run each test case. In test cases, tester specifies (a) how the system should be set up before running the test, (b) the test scenario (e.g. which request(s) is sent to verify a specific behavior), (c) how the system is supposed to behave correctly or not. That specification requires information dedicated to the elasticity. This includes the elasticity workflow, i. e., through which elasticity configuration the cloud system is led (e. g., sequence of resource allocation and deallocation). To manage the elasticity workflow, workload must be properly generated. The deployment and (re-)configuration of the cloud system also includes load generators setup, setting auto-scaling on cloud provider, (re-)starting cloud system components, etc.

Considering elasticity in test cases is complicated since it requires managing many parameters. Cloud providers, such as Amazon Elastic Compute Cloud (EC2)[1] and Google Cloud Compute Engine (CE)[2], usually provide Command-line Interfaces (CLI). CLIs help the tester to setup elasticity when testing since they abstract cloud system deployment, management, including elasticity's parametrization. However, they admit wide variability of cloud system configuration, more than necessary for elasticity testing. Furthermore, each cloud provider has its own CLI, which preclude portable commands (that execute over any cloud provider).

In this paper, we propose a generic Domain Specific Language (DSL) for elasticity setup when testing cloud systems, helping to generate and execute test cases. Our approach alleviates test case generation by centralizing the elasticity setup in this dedicated language, whereas the other part of a test case can be implemented as usual (e.g. in JUnit) This DSL helps tester to set up cloud system and its dependencies, auto-scaling, desired elasticity workflow, and test execution schedule. Then, it is compiled to a code that automatically executes elasticity testing, without further tester's interaction. In preliminary results, our DSL reduces the number of words to setup elasticity testing, and requires lower tester's effort to adapt a test case to be executed on several cloud providers.

The paper is organized as follows. Section 2 introduces some background. Section 3 describes the DSLs for elasticity testing setup. Section 3 describes how specifications are compiled to executable code. Section 5 discusses about the preliminary results using our approach. Section 6 reports the related work, and Section 7 concludes and lists perspectives.

[1]https://aws.amazon.com/ec2/

[2]https://cloud.google.com/compute/

2. BACKGROUND

In this section, we describe major aspects of cloud computing elasticity, which help understanding our DSL-based approach. We also remain Thiery et al. [19] DSL for setting up deployment of cloud systems.

2.1 Cloud Computing Elasticity

Main cloud infrastructures [1] admit by default threshold-based auto-scaling (i. e., elasticity). Figure 1 depicts a typical threshold-based elasticity. In this figure, we see that *resource demand* (continuous line) varies overtime, such variation essentially follows workload variation. For explanatory reasons, we only consider a resource demand that increases from 0 to 1.5, then decreases to 0.

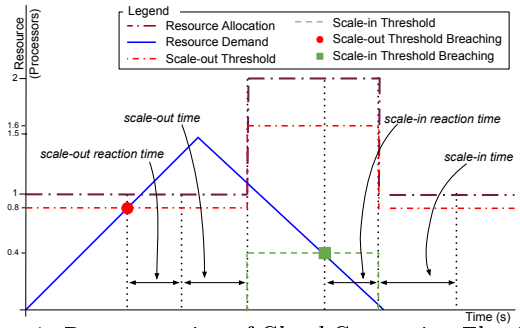


Figure 1: Representation of Cloud Computing Elasticity.

When resource demand exceeds the *scale-out threshold* and remains higher during the *scale-out reaction time*, cloud elasticity controller assigns a new resource. The new resource becomes available after a *scale-out time*, the time cloud infrastructure spends to allocate it. Once the resource is available, the threshold values are updated accordingly. It is similar considering the *scale-in*, respectively. Except that, as soon as the scale-in begins, the threshold values are updated and the resource is no longer available. Nonetheless, the infrastructure needs a *scale-in time* to release the resource.

2.2 Elasticity States

When applications are deployed on a cloud infrastructure, workload fluctuations lead to resource variations (elasticity). These variations drive the application to new, elasticity-related, states. Figure 2 depicts the runtime lifecycle of an application running on a cloud infrastructure.

At the beginning the application is at the *ready* state (*ry*), when the resource configuration is steady (*ry_s* substate). Then, if the application 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 application moves to the *scaling-out* state (*so*) and remains in this state while the resource is added. After a *scaling-out*, the application returns to the *ready* state. It is similar with the *scaling-in* state (*si*), respectively.

2.3 DSL for Cloud System Deployment

When testing cloud systems considering elasticity, testers first need to deploy the system under test. Thiery et al. [19] propose a DSL for setting up deployment of cloud systems.

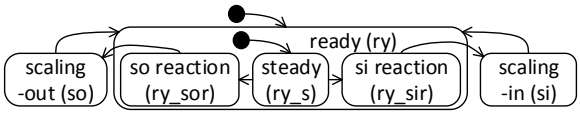


Figure 2: Elasticity states.

Their DSL is divided into two dimensions: deployment bundle, and cloud provider.

2.3.1 *Deployment Bundle (DB)*

It sets up cloud system components and dependencies.

We can set up multiple instances of software bundle. Each software bundle may group either cloud system or testing tool software components. Software component may have dependencies, such as other software component or external files (i. e., configuration files or executable scripts).

Listing 1[3] shows an example of software bundle setup for a Web application. First of all, we describe software artifacts. In the example, *http* is installed by a package installer (e. g., apt-get, yum, etc.), while *phpapp* is an interpreted PHP application, remotely transferred from local (*src*) to remote (*dest*) path. After, we describe external files, also with local and remote paths. Finally, we describes software bundles. In the example, we first bundles Web application artifacts (*app*), then the testing tool ones (*testingTool*).

Listing 1: Example of software bundle setup written in Thiery et al. DSL.

```
software_bundles {
  software httpd : pkg 'apache2' '2';
  software phpapp : src './app/',
    dest '/var/www/app/';
  ... // Other software components.
  source apachecfg : src 'httpd.conf',
    dest '/etc/apache/httpd.conf';
  ... // Other sources.
  bundle wsrv :
    app phpapp, dep (httpd, php, mysq),
    src (apachecfg, createdb, addserver),
    provScript (createdb, addserver);
  bundle appbench :
    testingTool app_bench, dep (java),
    src (bench_conf,bench),
    testScript (bench);
  ... // Other bundles. }
```

2.3.2 *Cloud Provider (CP)*

It sets up cloud providers' resource used for software bundles deployment.

Software bundles are deployed on deployment instances. Each deployment instance starts up an Operational System (OS) on a Virtual Machine (VM) that may have different combinations of computational resource (CPU, memory, etc.). Deployment instances use resource from a cloud provider's geographic zone. Software bundle may require some port configuration, e. g., port 80 for external interactions with Web server.

Listing 2 describes an example of Amazon EC2's resource setup. In this DSL, we start by setting up an VM image (*image*), which refers to existing cloud provider's image. In the example, we set an hypothetical image identifier (*ami-1234*),

[3]Instead of using the syntax of Thiery et al. syntax, we use the same syntax that the one of our proposal to be coherent.

and describe which Operational System (OS)[4] runs in the image. After, we list available cloud provider's geographical zones (*zone*). Finally, we set up deployment instances, referring a software bundle, VM image, cloud provider's machine type, port configuration, and geographic zone.

Listing 2: Example of cloud provider's resource setup written in Thiery et al. DSL.

```
resources EC2 {
  image iU704i386 :
    imageId 'ami-1234',
    os 'Ubuntu' '7.04' 'i386';
  zone EUWest :
'eu-west-1a', 'eu-west-1b';
  instance websrv :
    image iU704i386, machineType m3.large,
    portConfig '80' = '0.0.0.0', zone EUWest,
    bundle wsrv;
... // Other instances. }
```

3. ELASTICITY TESTING DSL

Despite Thiery et al. DSL allows a variety of cloud system deployment, it does not address cloud computing elasticity and elasticity testing. In this paper, we propose a DSL that complements Thiery et al. work, adding support to set up elasticity and elasticity testing. Our DSL is three-dimensional: auto-scaling, elasticity workflow, and test method schedule.

3.1 DSL to Set Up Auto-Scaling (AS)

To enable elasticity, we must set up auto-scaling on cloud provider. Threshold-based auto-scaling is a common strategy among major cloud providers [1]. It basically consists of varying a cloud resource when a threshold is breached for a while (see Section 2.1). Figure 3 illustrates the model that represents threshold-based auto-scaling setup.

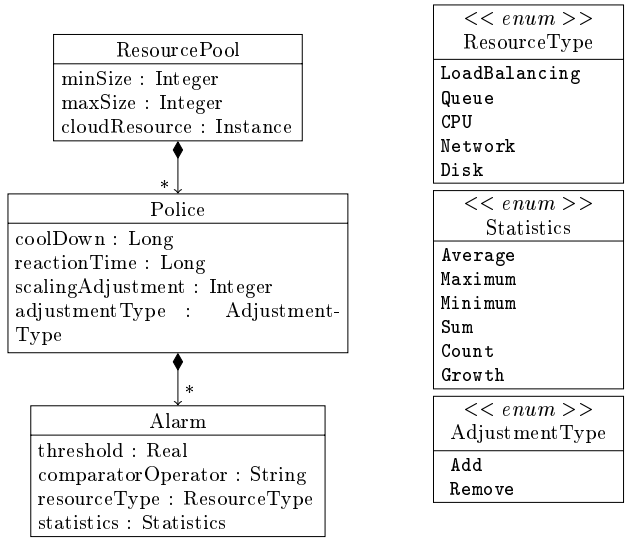


Figure 3: Model of threshold-based auto-scaling setup DSL.

ResourcePool describes the cloud resource (*cloudResource*) that is varied, and restricts its amount (between *minSize* and *maxSize*). Resource variation is regulated according to the *Police* properties. *Police* properties state time constraints (i. e., cool down[5] and reaction time periods), and resource variations (e. g., add one resource) performed when the time constraints are satisfied. *Police* is checked every time an alarm (*Alarm*) is triggered, i. e., threshold is breached. *Alarm* describes a threshold and resource usage that breaches the threshold.

Listing 3 describes an example of threshold-based auto-scaling. The resource pool *wsrv_pool* is associated to the *websrv* deployment instance (see Listing 2), and may have from 1 to 10 instances. As police, we set a cool down period of 60,000 *ms*, and a reaction time with same duration, adding one new resource from *wsrv_pool* resource pool. We set the alarm *highCPU* assuming threshold is breached when the maximum (*statistics=Maximum*) CPU usage (*resourceType=CPU*) is higher than (*comparatorOperator=>*) 60% (*threshold=60*).

Listing 3: Example of threshold-based auto-scaling setup written in our DSL.

```
elasticity {
 pool wsrv_pool : cloudResource websrv,
  minSize 1, maxSize 10;
 ... // Other pools.
 police wsrv_police : resourcePool slaves,
  coolDown 60000, reactionTime 60000,
  scalingAdjustment 1, adjustmentType Add;
 ... // Other polices.
 alarm highCPU : resourceType CPU,
  statistics Maximum, comparison '>',
  threshold 60, police wsrv_police;
 ... // Other alarms. }
```

3.2 DSL to Set Up Elasticity Driving (ED)

For some tests, such as regression testing and bug reproduction, it may be necessary to have a deterministic elasticity, reaching or repeating a strict behavior. In a previous work [5], we address deterministic elasticity generating proper workload that drives cloud system through required elasticity behavior, i. e., sequence of elasticity states. Despite our previous work successfully drives cloud applications through the given sequence of elasticity states, it requires much tester's effort. Tester has to write substantial amount of code to set up elasticity driving.

Another contribution of our DSL is to abstract elasticity driving setup. Figure 4 illustrates the DSL model. In the model, we have a resource pool (*pool*), which refers to cloud resource that is driven. It also admits a workload type (*workType*), and either to generate a sequence of elasticity states (*GeneratedFlow*) or to preset one (*PresetFlow*). To generate a sequence, we have to set the number of scaling-out (*scalingOuts*) and scaling-in (*scalingIns*) a sequence must have. Then, elasticity states are distributed in a way all scaling-in and scaling-out happen, respecting *ResourcePool*'s properties. For a preset sequence, we set the elasticity states in the order we want them to occur.

Listing 4 shows an example of elasticity driving setup. In the example, we set up elasticity driving to drive cloud resource in *wsrv_pool* pool using *Read* workload type. We preset the following sequence of elasticity states: *scaling-out*,

[4]This is because we cannot straightforwardly get OS information from cloud provider.

[5]Period within previous scaling activity takes effect, so new variation is not allowed.

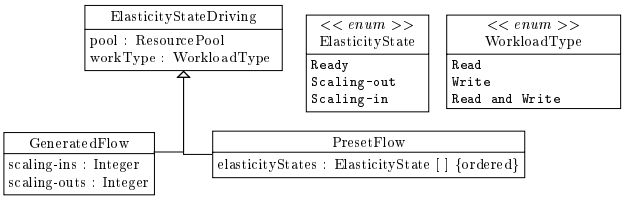


Figure 4: Model of elasticity driving setup DSL.

ready, *scaling-in*, *ready*, *scaling-out*.

Listing 4: Example of elasticity driving setup written in our DSL.

```
driving {
 drive wsrv_drive :
   pool wsrv_pool,  workType Read,
   states set(scaling-out, ready,
              scaling-in, scaling-out, ready);
}
```

3.3 DSL to Set Up Test Methods Schedule (TS)

Here, we follow a principle of our previous paper [6], where some of cloud system's tasks only occur during certain elasticity states. For instance, massive data replication only occurs when new nodes are already added (ready state). Therefore, it is not necessary to test it beforehand, i. e., while resource is being added (scaling-out state).

Our DSL allows tester to set up the execution of test methods during either specific states or all elasticity states. Figure 5 illustrates the model of setup of test method schedule. The model allows to associate *test methods* (*TestMethod*) to test suites (*TestSuite*). Test suites are associated to elasticity states (*ElasticityState*), elasticity states driving, and execution strategy (*Strategy*), i. e., in parallel, or in sequence.

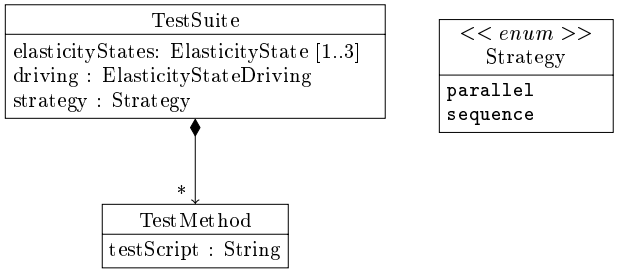


Figure 5: Model of test executions setup DSL.

Test methods have test script (*testScript*) as attribute. A test script can be any executable command, such as a java class (e.g. JUnit) or a shell script. In this way, tester can write generic test methods. Then, he/she associates test methods to elasticity states they must execute along.

Listing 5 shows an example of setup of test method schedule. In the example, we set two test methods (*test.test1* and *test.test2*) from a Java Archive (JAR) file (*test.jar*). Then, they are associated to scaling-out state of elasticity driving *wsrv_drive*, and set to execute in parallel.

Listing 5: Example of setup of test method schedule.

```
tests {
 test t1:
  script 'java -jar ./test.jar test.test1';
  test t2 :
  script 'java -jar ./test.jar test.test2';
 suite s1 : states scaling-out,
  driving wsrv_drive,
  test_method (t1,t2), in parallel;
}
```

4. COMPILING SPECIFICATION INTO EXECUTABLE CODE

In this section, we explain the compilation of elasticity testing setup written in our DSL into executable code. Figure 6 depicts the workflow of this compilation.

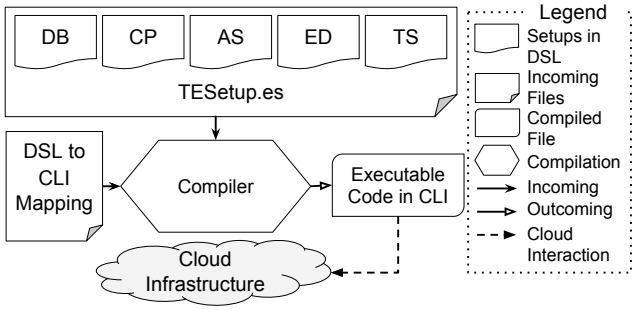


Figure 6: Workflow of compilation of elasticity testing specification into executable code.

First, tester writes test case setup (*TESetup.es*) using our DSL and Thiery et al. one (in the figure, divided by specialization). Then, tester selects a cloud provider's mapping file (*DSL to CLI Mapping*), which maps elements of our DSL to arguments of cloud provider's CLI. Finally, we compile test case setup into an executable code in cloud provider's CLI.

Listing 6 shows a partial Amazon EC2's mapping file, which describes the EC2's command to deploy instances and one of its arguments: image identifier (*–image-id*). This argument value is mapped from *imageId*, set in cloud provider's resource setup (see Listing 2). The compilation of this command would result in a code such as: $aws\ ec2\ run-instances\ --image-id\ iU704i386$ [...].

Listing 6: Example of command file for Amazon EC2.

```
provider AWS {
 instanceCommand 'aws ec2 run-instances' :
  '--image-id' imageId,  ...; }
```

Theoretically, using mapping file makes our approach portable to any cloud provider that allows command line. We provide mapping files for Amazon EC2, Google CE, and Openstack. Furthermore, we could write mapping files to further cloud providers, which is not a tester's task. A specialist, such as a network manager, writes this file once for all, then tester uses it.

5. PRELIMINARY RESULTS

In this section, we measure the impact of using our DSL on writing elasticity testing setup. We consider two case

studies, already used in previous papers, where we test distinct cloud applications through elasticity: 1) a MongoDB replica set [6], and 2) a distributed web application [5].

The main objective of this work is to alleviate tester's effort considering elasticity when generating test cases. Therefore, we so far provide a way to write elasticity part of test cases, then compile it to executable code. The current study do not go further on elasticity testing execution. However, we plan to do it as part of a future work.

5.1 First Case Study (CS1): Testing a MongoDB Replicat Set

The first case study tests MongoDB deployed as a replica set[6]. In this experiment, we drive MongoDB through the following sequence of elasticity states: *ready, scaling-out, ready, scaling-out, ready, scaling-in, ready, scaling-out, ready, scaling-in, ready, scaling-in, ready, scaling-in, ready.* Here, we consider a test method that tests performance of MongoDB through all elasticity states. The current experiment considers only the elasticity setup of a test case, since testers do not use our DSL to write the rest of the test case (i.e. the test methods referred in TS part of our DSL).

5.2 Second Case Study (CS2): Testing a Distributed Web Application

The second case study tests a distributed web application. Its architecture is made by a centralized database server, a load balancer, and n web servers. We drive the web application through 10 scale-out and 10 scale-in in sequence. In our previous paper, we do not test the web application. Here, we consider an hypothetically test case is associated to scaling-out and scaling-in elasticity states. We choose this test method schedule to be different than first case study.

5.3 Results

For each case study, we write the elasticity testing setup in our DSL. These setups are compiled to CLI for three different cloud providers: Amazon EC2, Google CP, and OpenStack. Then, we compare tester's effort on writing elasticity testing in both, our DSL and CLIs. We measure tester's effort in amount of words: *total amount of words*, and *cumulative amount of new words*.

5.3.1 Total Amount of Words

Total amount of words refers to the amount of words in an elasticity testing setup.

Table 1 describes the amount of words of elasticity testing setups for the elasticity testing case studies. Setups written in our DSL contain almost the same amount of words for all cloud providers (only 6 words changed as explained next subsection), while setups written with CLIs differ in amount of words according to cloud provider. Furthermore, setups written in our DSL result in fewer words for all cloud providers and case studies. Considering the amount of words as an effort, our DSL reduces considerably the tester's effort: Amazon EC2 ($CS1 \approx -24\%$, $CS2 \approx -22\%$), Google CP ($CS1 \approx -38\%$, and $CS2 \approx -36\%$), and OpenStack ($CS1 \approx -43\%$, and $CS2 \approx -39\%$).

Figure 7 depicts the effort in amount of words to write setups for the case studies. In the figure, the dashed line connects *CS1* efforts, while the solid line connects *CS2* efforts.

[6]https://docs.mongodb.com/manual/tutorial/deploy-replica-set/

Cloud Provider	CS1	CS2
Our DSL		
All Cloud Providers	246	273
CLIs		
Amazon EC2	326	364
Google CP	392	433
OpenStack	430	476

Table 1: Total amount of words of elasticity testing setups.

Furthermore, we see that such lines never cross each other, and the distance between them is almost homogeneous. This is because from CS1 to CS2, the effort varies proportionally (with an approximation between 1.09% and 1.1%) for every setup. This mean the difference among setups should be constant for other case studies. An encouraging result, which would result in less effort even when writing future elasticity testing setups in our DSL.

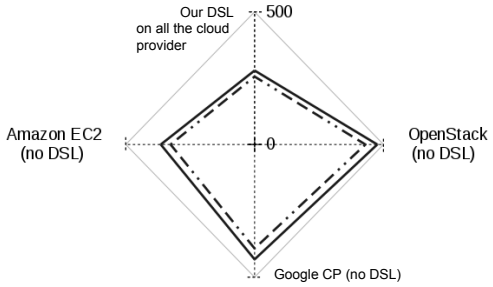


Figure 7: Effort on writing elasticity testing setups.

5.3.2 Cumulative Amount of New Words

Total amount of new words refers to the cumulative amount of words necessary to re-write an existing elasticity testing setup, making it suitable to other cloud provider. For instance, a tester may execute the same elasticity testing over different cloud providers, (re-)writing setups for all of them. We use the following formula to represent: $C_i = C_{i-1} + (S_i \odot S_{i-1})$, where i denotes the sequence the setup is written, and $S_i \odot S_{i-1}$ denotes the amount of new words from previous to current setup (S).

Graph of Figure 8 illustrates the cumulative amount of new words as case studies setup is (re-)written for a given sequence of cloud providers: Amazon EC2, Google CE, and OpenStack. In the figure, continuous lines illustrate the cumulative amount of new words for setups written in our DSL, while dashed lines illustrate the cumulative amount of new words for setups written with cloud providers' CLIs.

In the graph, we cannot see the variation for setups written in our DSL. This is because using our DSL the variation is slight, only 6 words change from one setup to another. These words refer to cloud provider's resource, such as image identifier and zone name, named distinctly among cloud providers. On the other hand, the variation for setups written in cloud providers' CLIs is visible, it more than double from first to last setup ($\approx *2.1$ for both case studies).

6. RELATED WORK

In literature, there are some model/DSL-based approaches

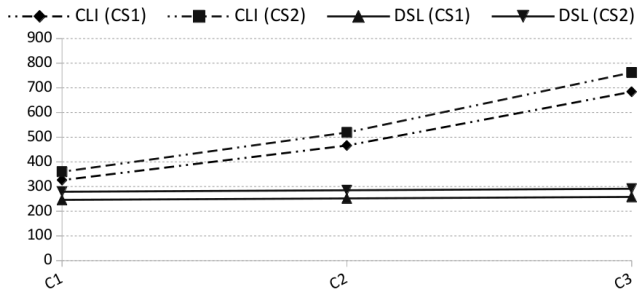


Figure 8: Total amount of new words of elasticity testing setups.

for the deployment and provisioning of cloud systems. However, most of them do not cover elasticity setup. Thiery et al. [19] propose a model-based approach to automate the deployment of cloud systems. Likewise, Kirschnick et al. [17] propose a DSL that is limited to the provisioning and deployment. Other work propose DSLs to deploy Software-as-a-Service (SaaS) [18] and Plataform-as-a-Service (PaaS) [9]. None of these approaches address Infrastructure-as-a-Service (IaaS). Goncalves et al. [14] propose Cloud Modeling Language (CloudML), which models services, resource profiles, and developer's requirements. However, their work requires the cloud provider to describe services and resources in CloudML, which is unusual. Finally, there are commercial DSL-based orchestration tools, such as Chef [2] and Puppet [3]. These tools allow not only the deployment and provisioning of cloud systems, but also the elasticity setup. However, these tools are not suitable for elastic testing as they don't support features such as elasticity states and test method schedule specifications.

7. CONCLUSION

In this paper, we propose a DSL-based approach to set up elasticity testing. Its major contributions are portability and reduction of tester's effort to write elasticity testing specifications. With a few changes in the setup, elasticity testing is executed over multiple cloud providers. With cloud providers' mapping files, we can easily adapt our approach to execute elasticity testing on any cloud provider. Our approach reduces considerable the amount of words on writing elasticity testing specifications. In future work we will focus on automatic resource discovering. For instance, finding the cheapest resource that fits testing requirements. This makes specification in our DSL completely portable: a single specification executed over multiple cloud providers without any change. We also think in new features related to our elasticity testing research that is going on: test case generation, and elasticity controller.

8. REFERENCES

[1] Cloud Computing Trends: 2016 State of the Cloud Survey.

[2] Chef Website, Aug. 2016.

[3] Puppet Website, Aug. 2016.

[4] D. Agrawal, A. El Abbadi, S. Das, and A. J. Elmore. Database scalability, elasticity, and autonomy in the cloud. *DASFAA'11*, pages 2–15, Apr. 2011.

[5] M. Albonico, J.-M. Mottu, and G. Sunyé. Controlling the Elasticity of Web Applications on Cloud Computing. In *Proceedings of the 31st SAC*. ACM, 2016.

[6] M. Albonico, J.-M. Mottu, and G. Sunyé. Monitoring-based testing of elastic cloud computing applications. In *Companion of ACM/SPEC ICPE*, pages 3–6, New York, NY, USA, 2016. ACM.

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

[8] M. M. Bersani, D. Bianculli, S. Dustdar, A. Gambi, C. Ghezzi, and S. Krstić. Towards the Formalization of Properties of Cloud-based Elastic Systems. In *Proceedings of the PESOS 2014*. ACM, 2014.

[9] R. Boujbel, S. Rottenberg, S. Leriche, C. Taconet, J. P. Arcangeli, and C. Lecocq. MuScADeL: A Deployment DSL Based on a Multiscale Characterization Framework. In *COMPSACW, 2014 IEEE 38th International*, pages 708–715, July 2014.

[10] A. Gambi, A. Filieri, and S. Dustdar. Iterative test suites refinement for elastic computing systems. In *Proceedings of the 9th ESEC/FSE 2013*, page 635, New York, New York, USA, Aug. 2013. ACM Press.

[11] A. Gambi, W. Hummer, and S. Dustdar. Automated testing of cloud-based elastic systems with AUToCLES. In *Proceedings of the ASE*, pages 714–717. IEEE, Nov. 2013.

[12] A. Gambi, W. Hummer, and S. Dustdar. Testing elastic systems with surrogate models. In *Proceedings of the 1st CMSBSE*, pages 8–11. IEEE, May 2013.

[13] A. Gambi, W. Hummer, H.-L. Truong, and S. Dustdar. Testing Elastic Computing Systems. *IEEE Internet Computing*, 17(6):76–82, Nov. 2013.

[14] G. Goncalves, P. Endo, M. Santos, D. Sadok, J. Kelner, B. Melander, and J. E. Mangs. CloudML: An Integrated Language for Resource, Service and Request Description for D-Clouds. In *2011 IEEE 3rd CloudCom*, pages 399–406, Nov. 2011.

[15] N. R. Herbst, S. Kounev, and R. Reussner. Elasticity in Cloud Computing: What It Is, and What It Is Not. *Proceedings of the ICAC*, 2013.

[16] S. Islam, K. Lee, A. Fekete, and A. Liu. How a Consumer Can Measure Elasticity for Cloud Platforms. In *3rd ACM/SPEC ICPE*, Proceedings of the ICPE '12, pages 85–96, New York, NY, USA, 2012. ACM.

[17] J. Kirschnick, J. Alcaraz Calero, L. Wilcock, and N. Edwards. Toward an architecture for the automated provisioning of cloud services. *IEEE Communications Magazine*, 48(12):124–131, Dec. 2010.

[18] K. Sledziewski, B. Bordbar, and R. Anane. A DSL-Based Approach to Software Development and Deployment on Cloud. In *Proceedings of the 24th IEEE AINA*, pages 414–421, Apr. 2010.

[19] A. Thiery, T. Cerqueus, C. Thorpe, G. Sunye, and J. Murphy. A DSL for Deployment and Testing in the Cloud. In *Proceedings of the IEEE ICSTW 2014*, pages 376–382, Mar. 2014.