Controlling Cloud-Based Systems for Elasticity Testing

Michel Albonico

UNIVERSITE BRETAGNE LOIRE

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Thèse de Doctorat

ALBONICO MICHEL

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'IMT Atlantique
Label européen
sous le sceau de l'Université Bretagne Loire

École doctorale : Sciences et technologies de l'information, et mathématiques

Discipline : Informatique et applications, section CNU 27
Unité de recherche : Laboratoire des Sciences du Numérique de Nantes (LS2N)

Soutenue le 28 août 2017
Thèse n° : 2017IMTA0027

Controlling Cloud-Based Systems for Elasticity Testing

JURY

Rapporteurs :	**M. CUNHA DE ALMEIDA EDUARDO**, Maître de Conférence, UFPR, Brésil
	Mme DU-BOUSQUET LYDIE, Professeur, Université de Grenoble
Examinateurs :	**M. LAMARRE PHILIPPE**, Professeur, INSA de Lyon
	M. MENAUD JEAN-MARC, Professeur, IMT Atlantique Nantes
Directeur de thèse :	**M. SUNYÉ GERSON**, Maître de Conférence, Université de Nantes
Co-directeur de thèse :	**M. MOTTU JEAN-MARIE**, Maître de Conférence, Université de Nantes

Acknowledgement

I would like to express my special thanks of gratitude to my supervisor Gerson Sunyé and to the IMT Atlantique for giving me the opportunity of doing my PhD in Nantes.

I thank my co-supervisor Jean-Marie Mottu without whom help I would not get the degree.

I thank the AtlanModelers for the moments we had together, and for all the help.

I thank the Brazilian government and the program Sciences Without Borders for the scholarship.

I thank the Federal University of Technology - Paraná for allowing me staying out of my activities for the studies period.

I thank my family and friends for encouraging me to keep myself focused. In special, I would like to thank Paulo Varela, Frederico and Sandra, Amine, Eric, Hamza, and Zied. The moments we had during this period helped me to get there.

In my family, the first two people I want to thank are my wife Daiane and my son Miguel, who were with me during all this period, and never gave up on me. I do not imagine my life without you two.

Finally, I thank to my parents Silvio and Claudete, who worked hard to get me in the University, and supported me always I needed. You did not have the opportunity of having an academic career. However, you were wise enough to see forward, and encourage me whenever I thought studying could not be the right way.

1

Résumé Étendu

L'élasticité est l'une des principales caractéristiques du cloud computing. Il s'agit de la capacité d'un cloud à allouer ou à désallouer automatiquement des ressources informatiques à la demande en fonction de l'évolution de la charge de travail [48] que doit traiter le système déployé sur le cloud (CBS, selon l'acronyme anglais). L'objectif principal de l'élasticité est de maintenir la Qualité de Service du CBS à un niveau adéquat et avec un coût minimum [26]. L'élasticité du cloud doit être assurée au plus juste en ajoutant des nouvelles ressources informatiques lorsque le CBS est surchargé, en désallouant des ressources lorsqu'elles sont sous-utilisées.

1.1 L'Élasticité en Cloud Computing

Le cloud computing a été largement adopté commercialement comme une plate-forme pour l'allocation dynamique de ressources informatiques [78]. Le fournisseur du Cloud loue l'accès à des applications complètes, des environnements de développement et de déploiement, ainsi qu'à l'infrastructure informatique, en fournissant du stockage et des traitements de données [24]. Ainsi, le cloud computing est un modèle d'accès à la demande à des ressources informatiques partagées.

Bersani et al. [26] proposent un modèle de l'élasticité, où des machines virtuelles (Virtual Machines VMs) sont allouées/désallouées à la demande. La Figure 1.1 représente ce modèle de l'élasticité. Les clients envoient leurs requêtes au CBS. La quantité de requêtes caractérise la charge de travail (workload, en opérations/seconde). Les besoins des clients n'étant pas constant, la charge de travail varie avec le temps, ce qui induit des variations dans l'utilisation des ressources informatiques. Pour éviter la surcharge

des ressources ou leur sous-utilisation, un contrôleur gère l'élasticité en faisant varier l'allocation des ressources selon la demande. Il surveille l'infrastructure du cloud et en fonction de seuils prédéfinis fixant l'utilisation optimale des ressources, il décide de réaliser des allocations ou désallocations. Cette stratégie est la norme suivie par les principaux fournisseurs commerciaux de cloud tels que Amazon Elastic Compute Cloud (Amazon EC2) et Google Cloud Platform.

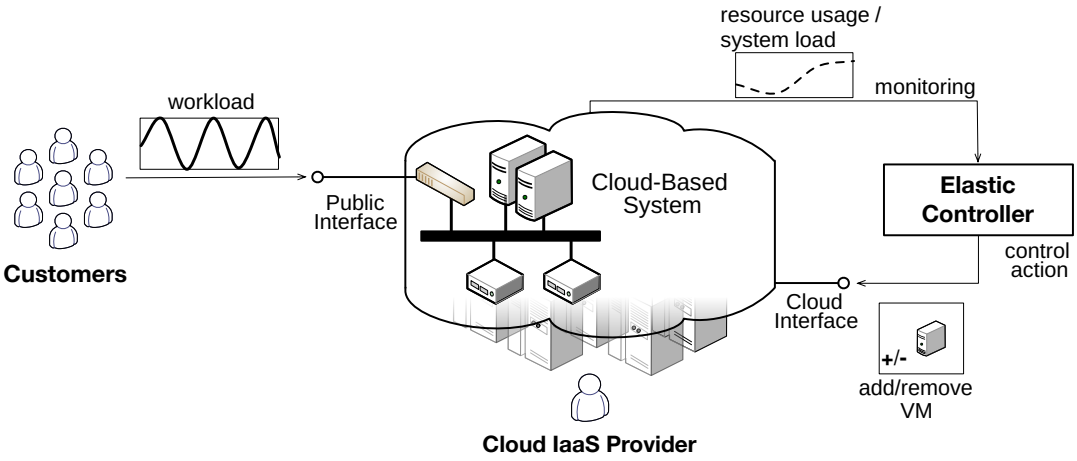


Figure 1.1 – Vue générale de l'élasticité d'un CBS

En mettant en œuvre l'élasticité, un CBS passe par différents états. La Figure 1.2 représente ces états et leurs transitions dans un diagramme de machine à états. Au début, le CBS est lancé et entre dans l'état *ready*, où la quantité de ressource alloué est stable (ry_s substate). Il se place alors dans le sous-état stable (steady). Ensuite, si le CBS est exposé pendant un certain temps à un nombre de requête au delà du seuil nécessitant une allocation *scale-out threshold* (sous-état ry_sor), le contrôleur d'élasticité commence à allouer une nouvelle ressource. À ce stade, le CBS se place dans l'état *scaling-out* et reste dans cet état pendant que la ressource est allouée. Après un *scaling-out*, l'application revient à l'état *ready*. De la même manière, lorsque le seuil de désallocation est franchi pendant un certain temps (sous-état ry_sir), le contrôleur d'élasticité commence à supprimer une ressource et les CBS se déplacent vers l'état *scaling-in*.

1.2 Motivation

L'élasticité dans le cloud computing n'est pas trivial et peut affecter la Quality of Service (QoS) du CBS ou causer des défaillances. Selon Bersani et al. [26]:

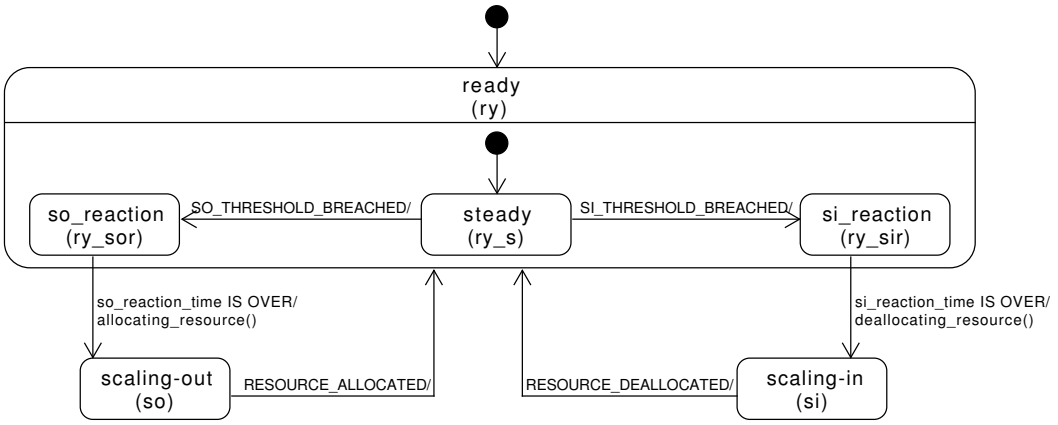


Figure 1.2 – Les états vis-à-vis de l'élasticité pris par un CBS

> L'allocation et désallocation de ressources peut entraîner des opérations non triviales à l'intérieur du système. La synchronisation et l'enregistrement de composants, la réplication et la migration des données sont les exemples les plus connus, qui peuvent également dégrader la QoS du système.

Par conséquent, pour garantir la qualité d'un CBS, il doit être soigneusement testé en considérant élasticité, ce qui nécessite des adaptations de ressources.

Gambi et al. [39] proposent un *framework* conceptuel pour les tests pendant l'élasticité qui couvre quatre activités de test : la génération de cas de test, l'exécution des tests, l'analyse des données et l'évolution des tests. La génération de cas de test et l'exécution des tests sont primordiaux dans les tests logiciels, tandis que l'analyse des données et l'évolution des tests sont des activités de test post-exécution. Dans cette thèse, nous nous concentrons sur les deux premières activités, requises pour l'exécution de tout test pendant l'élasticité. Compte tenu de la complexité du déploiement et de la gestion des CBSs élastiques, ces activités font face aux problème suivants.

La génération de cas de test peut se faire manuellement ou automatiquement. Elle est guidée pour atteindre des objectifs de test spécifiques, ce qui produit des cas de test regroupés en suites de tests [39]. Les cas de test spécifient les caractéristiques et les configurations du CBS et du générateur de charge utilisé pour générer la charge de travail en entrée du CBS, la configuration des propriétés d'élasticité sur le Cloud, ainsi que la configuration des outils de test. Chaque configuration nécessite une certaine expertise, elle a tendance à être complexe et laborieuse en raison de la quantité de paramètres impliqués. Ainsi, il est nécessaire d'abstraire ces configurations, et de rendre les langages de configuration de l'élasticité plus spécifiques à cette tâche. Gambi et al. [39] suggère également que les testeurs utilisent des outils génériques pour gérer automatiquement les

exécutions de test sur différentes plates-formes.

La génération ou l'écriture de cas de test manuellement peut entraîner l'oubli de configurations critiques, qui en manquant des bugs risque d'entraîner la défaillance du CBS. En effet, le test pendant l'élasticité admet de nombreux paramètres et une sélection aléatoire peut entraîner des cas de test qui ne combinent que des paramètres qui ne perturbent pas l'exécution du CBS. D'autre part, compte tenu du nombre de paramètres, le test de tous les cas de test possibles peut être long et coûteux. Par conséquent, nous avons besoin d'une approche qui réduit le nombre de cas de test, tout en maintenant la capacité à trouver des problèmes d'élasticité.

L'objectif principal en testant pendant l'élasticité le CBSs est de vérifier la manière dont il fonctionne pendant ces états spécifiques à l'élasticité. Plus précisément, les testeurs peuvent vouloir tester le CBS tout au long d'un comportement élastique prédéfini. Par exemple, il peut être nécessaire de tester le comportement du CBS tout au long d'une séquence d'allocation de ressources. Dans un autre cas, il peut être nécessaire de stresser un CBS en le menant à travers une alternance d'allocations et désallocations, ce qui devrait nécessiter les différentes adaptations CBS à plusieurs reprises. L'élasticité réagit principalement aux variations de la charge, nous devons donc exprimer les cas de test en termes de charge de travail pour forcer un état d'élasticité particulier ou déclencher des adaptations élastiques spécifiques [39]. Cependant, ce contrôle fait face à un défi dans l'estimation des variations de charge de travail nécessaires pour mener un CBS à travers un comportement élastique prédéfini. En effet, l'impact d'une charge de travail peut être différent en fonction des CBSs et de la ressource sur laquelle il est déployé. Par conséquent, nous avons besoin d'une approche qui, compte tenu d'une charge de travail et d'un CBS, estime avec précision les variations de charge de travail qui mènent le CBS tout au long d'un comportement élastique requis.

Si un cas de test est écrit afin de tester une adaptation spécifique du CBS, il ne devrait être exécuté que si une telle adaptation est effectuée. Par exemple, considérons que le testeur veut tester la *synchronisation d'un composant* (e.g., des nodes dans notre cas d'étude MongoDB), i. e., lorsqu'un nouveau composant est ajouté au système en cours d'exécution. La synchronisation des composants n'est effectuée qu'après l'allocation d'une nouvelle ressource. Par conséquent, le test de cette adaptation nécessite d'être exécuté à ce moment précis. C'est une tâche difficile, où les testeurs doivent identifier différents états du CBS en continu, et en parallèle, exécutez les tests appropriés.

Les test doivent être déterministes quand ils sont répétés [36, 15]. De cette manière, quand un test détecte un bug, il peut être relancer pour le diagnostiquer et corriger. De la même manière, il est possible d'effectuer des tests de non-régression. Chaque exécution doit reproduire le même comportement, ce qui exige que la conception des tests soit déterministe. Dans le cas d'un test pendant l'élasticité, la première exigence consiste à reproduire le comportement élastique. En outre, la seule reproduction du comportement élastique peut ne pas suffire à la reproduction de certains tests liés à l'élasticité, et

prévenir la détection d'un bug. La reproduction de certains tests peut nécessiter une combinaison d'adaptation avec d'autres conditions. C'est le cas du bug 7974 de la base de données MongoDB NoSQL, dont le diagnostic nécessite que les tests reproduisent des interactions avec le système pendant des états spécifiques : la création d'un index unique avant que l'un de ses nœuds de cluster ne soit supprimé et télécharge un document après qu'un nouveau nœud est été ajouté au cluster. D'autre part, le bug 2164 d'Apache ZooKeeper ne se produit que lorsque le composant leader quitte un système avec trois nœuds, ce qui nécessite qu'une ressource spécifique (i. e., celle qui héberge le leader) soit supprimée. Par conséquent, nous avons besoin d'une approche qui reproduise le comportement élastique CBS, et en parallèle, qu'elle satisfasse d'autres conditions pour la reproduction des bugs liés à l'élasticité.

1.3 Contributions

Nous proposons cinq contributions pour traiter ces problèmes. Chaque contribution est brièvement décrite ici.

1.3.1 Langage Spécifique au Domaine (DSL) pour le Test d'Élasticité

Nous proposons un langage spécifique au domaine (DSL) pour la configuration des tests pendant l'élasticité, qui réduit la complexité de cette configuration en utilisant une syntaxe adaptée. En outre, notre DSL réduit le nombre de mots par rapport à d'autres langages. Nous proposons également un moyen de compiler les spécifications de test d'élasticité dans notre DSL d'une manière indépendante du fournisseur cloud. Cela permet d'exécuter le même test pour différents fournisseurs de cloud sans modification des spécifications.

Ce DSL est divisé en trois parties indépendantes du fournisseur de cloud : déploiement des composants CBS (*Deployment*), élasticité (*Elasticity*), et configuration des tests pendant l'élasticité (*ElasticityTesting*). Nous proposons également une quatrième partie (*Provider*), qui dépend du fournisseur du cloud, utilisée pour répertorier les ressources disponibles sur un fournisseur de cloud.

La Figure 1.3 représente un aperçu de notre méthodologie pour compiler la configuration de test d'élasticité écrite dans notre DSL vers le code exécutable. Dans la figure, nous voyons que les configurations dépendantes du fournisseur de cloud (*Provider Dependent*) et les configurations indépendantes du fournisseur de cloud computing (*Provider Independent*) sont décrites dans des fichiers séparés. Ensuite, si le même test doit être exécuté sur différents fournisseurs de cloud, le seul fichier qui doit être modifié est *Provider Dependent*. En outre, ce fichier ne doit être écrit qu'une seule fois, et un fichier *Provider Dependent* peut être réutilisé à partir d'exécutions précédentes des tests.

Notre méthodologie de compilation est divisée en deux étapes : *resource matching* et *génération de scripts*.

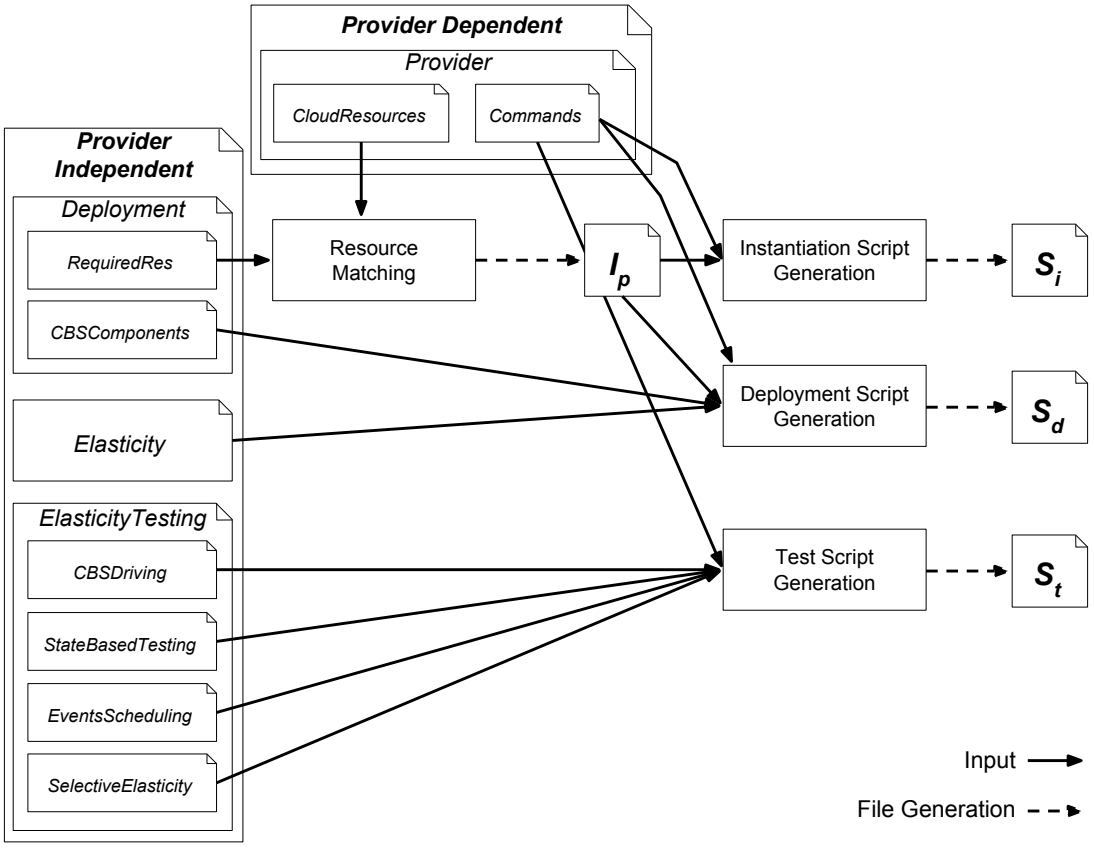


Figure 1.3 – Compilation des configurations de test pendant l'élasticité en code exécutable

La phase de correspondance des ressources consiste à associer les spécifications dans *RequiredRes* et *CloudResources* d'un fournisseur de cloud p, en générant un fichier I_p. Ce fichier contient les descriptions de ressources dans *RequiredRes* mises à jour avec des valeurs de ressources associées en *CloudResources*.

La deuxième étape de notre méthodologie de compilation consiste à générer tous les scripts nécessaires à l'exécution des tests pendant l'élasticité : instanciation de ressources (S_i), CBS (S_d) et tests eux mêmes (S_t). Ces scripts contiennent des commandes Command-line Interface (CLI), utilisées pour interagir avec le fournisseur du cloud.

Cette approche réduit considérablement la quantité de mots pour l'écriture des spécifications de test pendant l'élasticité, et ces spécifications sont portables entre les fournisseurs de cloud.

1.3.2 Génération de Séquences de Test pour le Test d'Élasticité

Les tests d'élasticité ont un très grand espace de configuration, où l'exécution de tous les tests possibles est prohibitive. Par conséquent, nous devons générer un nombre réduit de configurations tout en veillant à ce qu'elles soient pertinentes. Étant donné que la technique Combinatorial Interaction Testing (CIT) (Combinatorial Interaction Testing) ont été utilisées dans la littérature, en présentant des résultats convaincants [63], nous basons notre méthodologie sur celle-ci. La CIT est une stratégie qui consiste à tester toutes les combinaisons de paramètres T-wise d'un système. Cela signifie que, en considérant les valeurs des paramètres n, les tests combinatoires T-wise n'introduisent que $2^T \cdot \binom{n}{T}$ configurations Par rapport à $n!$ Requis pour des tests exhaustifs.

La Figure 1.4 représente notre flux de travail méthodologique, qui se divise en trois étapes: 1) Tout d'abord, nous modélisons les aspects du comportement élastique (*paramètres d'élasticité*), tels que les états d'élasticité, les seuils et la charge de travail dans un Classification Tree Model (CTM). 2) Ensuite, nous générons un ensemble de configurations de test couvrant toutes les interactions T-wise valides entre les paramètres de l'élasticité. 3) Enfin, nous générons des séquences de test couvrant toutes les transitions possibles entre les configurations de test. Finalement, les séquences de test peuvent être exécutées sur un CBS.

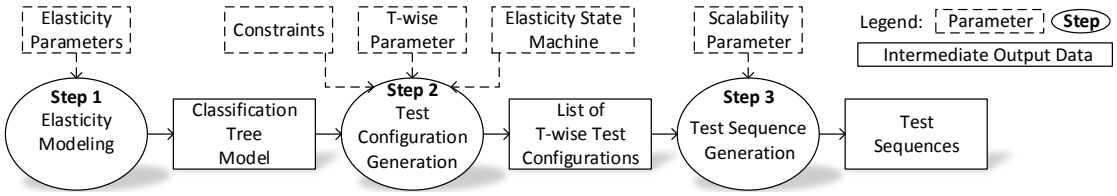


Figure 1.4 – Les trois étapes de notre méthodologie

Les résultats préliminaires encouragent d'autres recherches, alors que les cas de test générés par la couverture par paires révèlent déjà plusieurs problèmes liés à l'élasticité dans une étude de cas.

1.3.3 Pilotage du CBS à Travers de l'Élasticité

Nous affirmons que les testeurs devraient pouvoir conduire le CBS de manière déterministe, en contrôlant l'élasticité en fonction de la propriété qu'ils souhaitent tester. En plus d'être plus précis dans les tests, cela peut également réduire le temps d'exécution des tests en optimisant l'élasticité. Cela induit également une réduction des coûts puisque la plupart des fournisseurs de cloud utilisent la politique de *pay-as-you-go*, où les consommateurs paient pour le temps qu'ils utilisent les ressources.

La Figure 1.5 représente le processus proposé, qui est divisé en trois phases d'exécution: *workload profiling*, *calcul de la charge de travail* et *application leading*. Le CBS peut réagir de manière distincte lorsqu'il est exposé à la même charge de travail [26]. Par conséquent, avant de calculer la variation de la charge de travail, nous devons faire un profil de chaque combinaison de CBS et de la charge de travail. Gambi et al. [42] considèrent qu'une charge de travail d'entrée a trois caractéristiques: *la charge de travail (dont son type)*, *le mélange de demandes (pattern)*, et *l'intensité des demandes*. Basé sur cela, nous calculons *l'intensité des demandes* qui conduisent le CBS dans des états d'élasticité requis (RES). Dans la phase *application leading*, nous menons les CBSs en utilisant les intensités de charge de travail calculées (WI). Nous exposons le CBS à chaque intensité de charge de travail jusqu'à ce que l'état d'élasticité demandé se termine. Nous surveillons périodiquement l'infrastructure de cloud computing pour identifier si les états d'élasticité sont atteints.

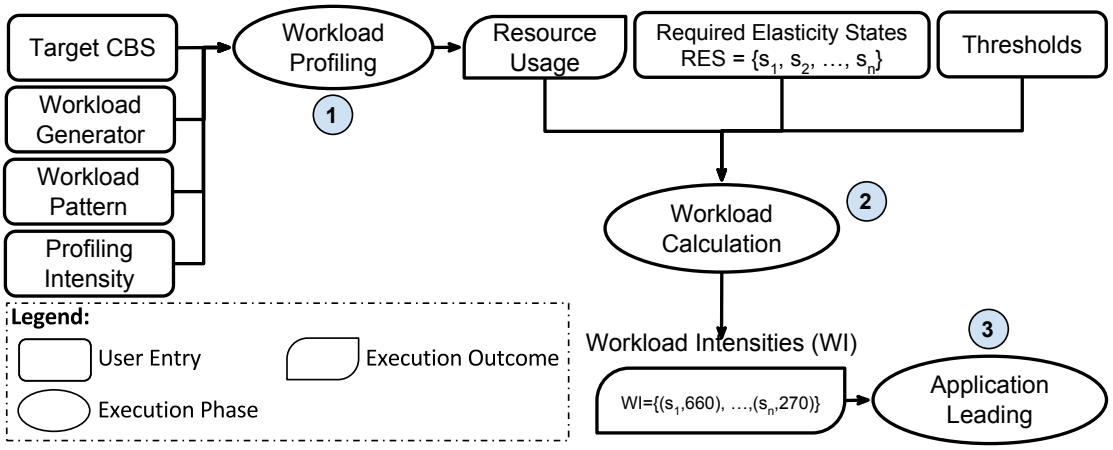


Figure 1.5 – Processus pour piloter le CBS

Les résultats montrent que l'approche est capable de conduire CBSs à travers des comportements élastiques prédéfinis en peu de temps, et sans stresser les CBS.

1.3.4 Test Basé sur les États d'Élasticité

Nous proposons une méthodologie pour tester des CBSs qui est basé sur les états d'élasticité. Nous nous concentrons sur l'association des tests avec des états d'élasticité spécifiques et la synchronisation des exécutions de test en conséquence.

Pour associer un test à un état d'élasticité, les testeurs doivent configurer certains paramètres prédéfinis: la priorité (*priority*), le retard (*delay*) et la répétition *repeat*. *Priority* se réfère à la séquence dans laquelle le test est exécuté, dans le cas de plusieurs méthodes de test pour le même état. Les tests avec la même priorité sont exécutées en

parallèle. *Delay* est le temps par lequel l'exécution du test est retardée, permettant des adaptations du CBS spécifiques. *Repeat* est un paramètre booléen qui définit si le test doit ou non être ré-exécuté pendant l'état d'élasticité. S'il est à vrai (*true*), l'exécution du test est répétée jusqu'à ce que CBS soit dans l'état d'élasticité voulu.

Grâce à cette méthodologie, nous avons découvert les causes de la dégradation de la performance d'une étude de cas. Pour une telle étude de cas, la méthodologie est utilisée pour identifier l'état d'élasticité où se produisent des dégradations de performance, puis tester le étude de cas pendant cet état.

1.3.5 Reproduction des Tests pendant l'Élasticité

Nous proposons un prototype qui aborde les spécificités de la reproduction des tests d'élasticité : le contrôle de l'élasticité, l'élasticité sélective, et la planification de événements. Nous proposons également un moyen d'accélérer l'élasticité lors de la reproduction des tests. En effet, la conduite de CBSs prend beaucoup de temps puisque les contrôleurs de l'élasticité prennent un certain temps pour réagir à une demande de ressources. Étant donné qu'un test peut être ré-exécuté plusieurs fois, cela implique un coût élevé.

La Figure 1.6 représente l'architecture globale du prototype. Cette architecture est composée de quatre composants: Elasticity Controller Mock (ECM), Workload Generator (WGen), *Event Scheduler* (ES) et *Cloud Monitor* (CM).

Le rôle du composant ECM est de fournir le *contrôle de l'élasticité* et l' *élasticité sélective*. Il simule le comportement du contrôleur d'élasticité du fournisseur de cloud, ce que imite l'addition et la suppression des ressources de façon déterministe. Le rôle du Cloud Monitor (CM) est d'*identifier les états d'élasticité*, en maintenant cette information à jour sur l'ECM, ce qui aide à assurer le *contrôle de l'élasticité*. Le CM implémente le composant de surveillance du pilote d'élasticité. Le WGen génère la charge de travail qui devrait conduire les CBS à travers des états d'élasticité demandés. Le WGen est également basé sur le prototype du pilote d'élasticité. De manière analogue à son nom en anglais, l'Event Scheduler (EvS) assure la *planification des événements* en synchronisant les événements avec des changements élastiques. L'EvS reçoit comme entrée un ensemble S, contenant des paires d'événements (e) et un changement d'élasticité (ec). Lorsqu'un nouveau changement élastique commence, l'EvS reçoit un message de l'ECM. Ensuite, tous les événements connexes sont exécutés en fonction de leurs paramètres.

Ce prototype accélère également la reproduction des tests. Nous utilisons ce prototype pour reproduire 3 bugs représentatifs, où le prototype a réussi à les reproduire tous.

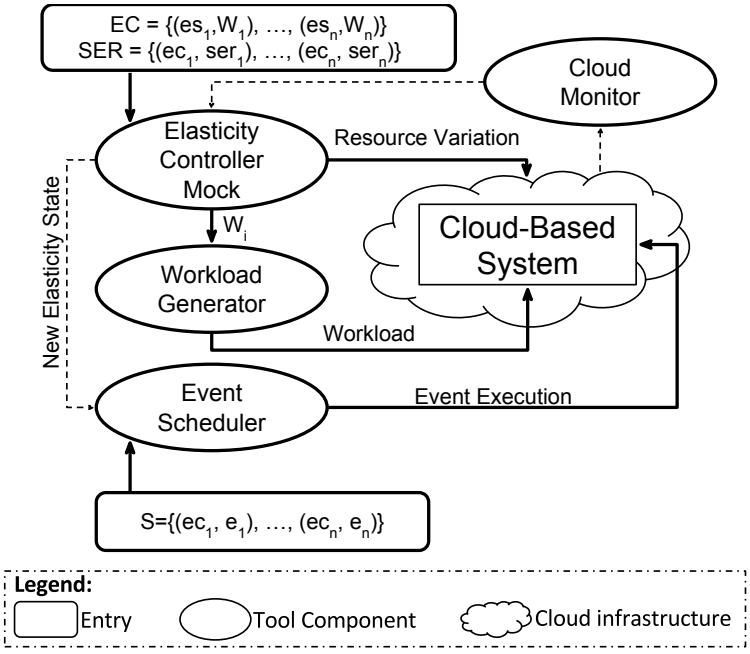


Figure 1.6 – Architecture Génerale

1.4 Travaux Futurs

Nous avons l'intention de mettre toutes ces approches ensemble dans un *framework* de test et de le mettre à la disposition du milieu de la recherche. Cependant, avant cette étape, nous visons à améliorer chaque approche séparément.

La phase de compilation de l'approche DSL est jusqu'à présent manuelle, et nous souhaitons la mettre en œuvre dans le cadre d'un travail futur. Une autre perspective consiste à étendre le DSL pour que les testeurs l'utilisent pour écrire leurs tests plutôt que d'autres langages (e.g., JUnit). Enfin, nous pourrions modifier la syntaxe du DSL pour la rapprocher d'un langage de programmation plutôt que d'un fichier de configuration.

Concernant la génération de cas de test, nous prévoyons mener une expérimentation complète qui identifie quelle est la couverture minimale des paramètres de test d'élasticité pour révéler la plupart ou tous les problèmes liés à l'élasticité. Cela nécessite également une évaluation plus approfondie des paramètres de test d'élasticité, telles que des configurations de charge de travail supplémentaires. Une autre perspective est de prendre en compte une plus grande suite de tests, qui couvre d'autres aspects en plus de la performance du CBS et d'autres études de cas de CBS.

Le pilote d'élasticité considère que le CBS a une évolutivité linéaire, ce qui peut ne pas être vrai en fonction des CBSs ou de la quantité de ressources. Par conséquent, nous prévoyons proposer une stratégie qui considère l'évolutivité non linéaire. Une

autre perspective est d'adapter notre approche pour considérer des politiques d'élasticité prédictive, car jusqu'à présent, cela ne fonctionne qu'avec des réactives.

2

Introduction

In cloud computing, the *cloud provider*, i. e., the company that administrates cloud computing infrastructures, rents access to a pool of computing resources. Elasticity is one of the main characteristics that arose with cloud computing, which is the ability of a system to allocate or deallocate computing resources on demand to meet the workload changes [48].

Bersani et al. [26] present a general model of cloud computing elasticity. In their model, customers send their requests (workload) to the system deployed on a cloud computing infrastructure (the Cloud), i. e., Cloud-Based System (CBS). Since customers are seasonal, the workload varies over time, which likewise varies the demand of computing resources. To avoid resource overuse or underuse, the cloud computing elasticity controller (or simply *elasticity controller*) monitors the resource usage, and decides when to scale-out (allocate) or scale-in (deallocate) a resource based on preset resource usage thresholds.

2.1 Motivation

Since the elastic controller scales the resources at CBS run-time, the CBS must deal with intermittent computing resources by adapting themselves constantly. This adaptations are not trivial and may affect the CBS execution. According to Bersani et al. [26]:

> "Scaling up or down 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 Quality of Service (QoS)."

Therefore, to guarantee the CBS quality, we must test it in the elasticity presence, i. e., *elasticity testing*. For the remainder of this thesis, we must be aware of the difference between scalability testing and elasticity testing, which is motive of confusion [48]. While the scalability testing consists in testing the CBS after a resource scaled out or in, the elasticity testing includes the CBS testing during resource scaling.

All over this thesis, we only consider elasticity testing, which have some requirements. Section 2.2 describes elasticity testing requirements. Section 2.3 describes the contributions of this thesis to meet such requirements.

2.2 Problem Statement

This section presents and discusses five elasticity testing requirements and the activities necessary to meet them.

2.2.1 CBS Driving Throughout Elasticity

The main goal of testing elastic CBS is to verify how the system works in the presence of elasticity. More specifically, testers may need to test CBS throughout a preset elastic behavior. For instance, one could need to test a CBS by interleaving resource scale out and in, which should reproduce most of CBS adaptations. Since elasticity controllers react to variations in the workload [39], to drive the CBS, tests must lead workload variations that result in the preset elastic behavior. However, this faces a challenge in estimating the workload variations. Indeed, the effect of a workload depends on the CBS and the resource that hosts it. Therefore, the CBS driving depends on profiling the effects of the workload on the CBS, and then, calculating and generating the workload variations.

2.2.2 Elasticity Test Synchronization with CBS States

If testers aim at testing a specific CBS adaptation, they should only execute the test when such adaptation occur. For instance, let us consider the tester needs to test a CBS component registration, i. e., when a new component registers to the running CBS. Such registration is only performed when the new component is running, i. e., after a resource scale-out. Thus, tests that verify the registration of CBS components should only be executed after a resource scale-out. Another case is to test the CBS during different resource configurations, e. g., during a resource allocation, and when it is already allocated. This reduces each test extent, and makes the analysis of test results

more specific. Both test synchronizations require to model the CBS states, to identify these states at CBS run-time, and to synchronize test executions according to CBS states.

2.2.3 Elasticity Test Reproduction

During software development, testers have to execute regression tests regularly [36], which requires the design of deterministic tests. Then, tests expose the tested systems to the same conditions as the previous execution, which should reproduce the same behavior. Testers must design elasticity tests to repeat the elastic behavior and possible further conditions, which we call *time-based events*, such as user interactions with the CBS. This is the case to test the MongoDB NoSQL database bug 7974 [8]. The first requisite to reproduce this bug, is the following elastic behavior: create a MongoDB replica set [11] with three nodes (three resource scale-out), remove a MongoDB node (one resource scale-in), and add a new MongoDB node (one resource scale-out). The bug reproduction also requires two time-based events: 1) to create a unique index before one of the MongoDB nodes is removed (the resource scale-in), and 2) upload a document after a new node is added (before the last resource scale-out). To repeat an elastic behavior testers must design the elasticity test to repeat the workload generation. To repeat time-based events testers must design the elasticity test to synchronize their executions by identifying when they occur.

Reproducing elasticity tests sometimes also requires to repeat a strict variation of CBS components. Indeed, CBS components can have different status, such as master and slaves, assigned dynamically, such as by an election algorithm. Since elasticity controllers work at resource level, they do not consider the CBS component status to make decision of which resource to variate. Thus, during the test execution CBS components vary in a non-deterministic manner, where different executions may result in distinct component variations. However, to reproduce the same CBS behavior, elasticity tests must manage the CBS component variation in a deterministic manner. For instance, the Apache ZooKeeper bug 2164 [13] only occurs when the leader component leaves a ZooKeeper cluster. When testing such bug, the test must manage to repeat the deallocation of the leader component.

2.2.4 Elasticity Test Implementation

Implementing elasticity tests is complex, laborious, and requires the tester to master cloud computing and its particularities. The tester must interact with the cloud provider multiple times for: deploying and configuring the CBS and testing tools, configuring the elasticity controller, and executing the tests. Since each cloud provider has its own interfaces (e. g., Web dashboard, Command-line Interface (CLI), and Application Programming Interface (API)), when testers need to execute tests over different cloud computing infrastructures, they must repeat all the process. This requires a method that

abstracts the complexity of elasticity testing implementations, reduces the tester interactions with the cloud provider, and is portable among cloud computing infrastructures.

2.2.5 Elasticity Test Generation

Testers can create elasticity tests manually, or use test case generators to create them automatically [39]. Creating elasticity tests manually or randomly may miss critical combinations of elasticity testing parameters, which reveal CBS issues. Indeed, elasticity tests admit many parameters for each interaction with the cloud provider. For instance, the CBS deployment allows to configure different resources where to deploy it, while the elasticity controller accepts a range of values to configure the resource usage thresholds. Furthermore, different workloads can drive the CBS throughout elasticity. Ideally, a set of elasticity tests should cover all the possible combinations of elasticity testing parameters. However, given the great number of parameters, this set is too large, which impedes its execution within a reasonable time and an affordable cost. Therefore, the set of elasticity tests should cover critical combinations of elasticity testing parameters, and be small enough to executed in a feasible time.

2.2.6 Summary of Elasticity Testing Requirements

Table 2.1 summarizes the elasticity testing requirements and the activities necessary to meet them.

2.3 Contributions

This thesis proposes approaches that meet the elasticity testing requirements. This section briefly describes each approach.

To drive the CBS, we propose an approach divided into three phases: *workload profiling*, *workload calculation*, and *application leading*. The *workload profiling* phase consists in profiling the resource usage for combinations of CBSs and workloads. Then, in the *workload calculation* phase, the approach uses the resource usage profiles to calculate the workload variations that drive the CBS throughout a preset elastic behavior. Finally, in the *application leading* phase, the approach generates the calculated workload variations. This approach allows to drive different combinations of CBSs and workloads in a deterministic manner. In a second approach, we propose a way to synchronize elasticity tests by CBS state. We model the CBS states as resource configurations, which we call *elasticity states*: the *scaling-out* state is the resource allocation period, the *ready* state is the period while the resource is steady, i. e., it is not being varied, and the *scaling-in* state is the resource deallocation period. We also propose a way to associate the tests

Elasticity Testing Requirement	Activities to Meet the Requirement
CBS Driving Throughout Elasticity	*-Profile the effect of the workload on the CBS.* *-Calculate the necessary workload variations.* *-Lead the workload generation.*
Elasticity Test Synchronization with CBS States	*-Model the CBS states.* *-Identify the states at CBS run-time.* *-Coordinate the synchronization of tests with the CBS states.*
Elasticity Test Reproduction	*-Repeat the CBS elastic behavior.* *-Repeat the time-based events.* *-Repeat the CBS component variation.*
Elasticity Test Implementation	*-Deploy and configure the CBS.* *-Configure the elasticity parameters of the elasticity controller.* *-Deploy and configure the elasticity testing tools.* *-Execute the elasticity test executions.*
Elasticity Test Generation	*-Model the elasticity testing parameters.* *-Generate a small set of elasticity tests that cover critical combinations of the modeled elasticity testing parameters.*

Table 2.1 – Elasticity Testing Requirements

to elasticity states. Then, the approach monitors the CBS and the Cloud to identify the elasticity states, and coordinates the tests execution accordingly.

The third approach ensures the elasticity test reproduction by repeating the elastic behavior, the time-based events, and the CBS component variation in a deterministic manner. To repeat the elastic behavior, the approach only repeats the workload generation, configure such as in the first approach. To repeat the time-based events, the approach consists in associating them to elasticity states, such as in the second approach. Then, the approach identifies the elasticity states at CBS run-time, and synchronizes the time-based events accordingly. To repeat the CBS component variation, the approach has a component that mimics the elasticity controller, and bases the resource deallocations on the CBS component status.

The fourth approach introduces a Domain Specific Language (DSL) to configure elasticity tests. Testers use this DSL to configure the deployment and the configuration of the CBS and the testing tools and the elasticity controller parameters, and to plan the test execution. Then, this approach uses a cloud provider-independent compilation strategy to execute elasticity test configurations over different cloud computing infrastructures.

Finally, the fifth approach proposes a way to generate a small set of elasticity tests that cover critical combinations of elasticity testing parameters. In this approach, testers model the elasticity testing parameters into a Classification Tree Model (CTM) by mapping them as dependent variables into a tree structure. In this CTM, the parameter

values are tree leaves. Then, the approach traverses the tree, and generates a set of elasticity tests that covers all T-wise combinations of parameters values [63]. This means that, when considering n parameter values, T-wise combinatorial testing investigates only $2^T \cdot \binom{n}{T}$ combinations compared to the $n!$ required for exhaustive testing. Finally, the approach arranges the generated elasticity tests in test sequences that mimic CBS reconfigurations, and therefore, can be executed in a single run. In the experiments, this approach reveals several elasticity-related issues of a case study by reducing the number of elasticity tests by $\approx 92\%$.

2.4 Outline of the Thesis

The remainder of this thesis is organized as follows. Chapter 3 introduces cloud computing and software testing to contextualize the work in this thesis, and discusses the major work in CBS elasticity testing. Then, Chapter 4 to Chapter 7 present the approaches proposed in this thesis. Chapter 4 presents the approach for CBS driving. Chapter 5 presents the approach for elasticity state-based tests. Chapter 6 presents the approach for elasticity test reproduction. Chapter 7 presents the DSL-based approach for elasticity test implementation. Chapter 8 presents the approach for elasticity test generation. Chapter 9 concludes this thesis by recapitulating all the contributions, and gives directions of future work.

3

State of the Art

This chapter presents the literature related to the topics addressed in this thesis. Section 3.1 starts by contextualizing cloud computing, its service models and the elasticity feature. Then, Section 3.2 contextualizes software testing. Section 3.4 presents research efforts that focus on automating CBS deployment on the Cloud. Section 3.5 discusses about the state of the art of CBS elasticity testing.

3.1 Cloud Computing

Cloud computing infrastructures have been adopted commercially as a platform for dynamic resource allocation [78]. The cloud provider rents access to featured application, development and deployment environments, and computing infrastructure, such as data storage and processing [24]. Indeed, cloud computing is a model for on demand access to shared computing resource pool.

3.1.1 Cloud Computing Service Models

the Cloud provides computing resources as three *service models* [24, 73, 65]: Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), and Infrastructure as a Service (IaaS). Figure 3.1 depicts the roles of customers and cloud providers for each cloud computing service model.

Definition 1 (SaaS) *It is the service model where the cloud provider manages the underlying cloud infrastructure completely, while the only consumers role is to use the*

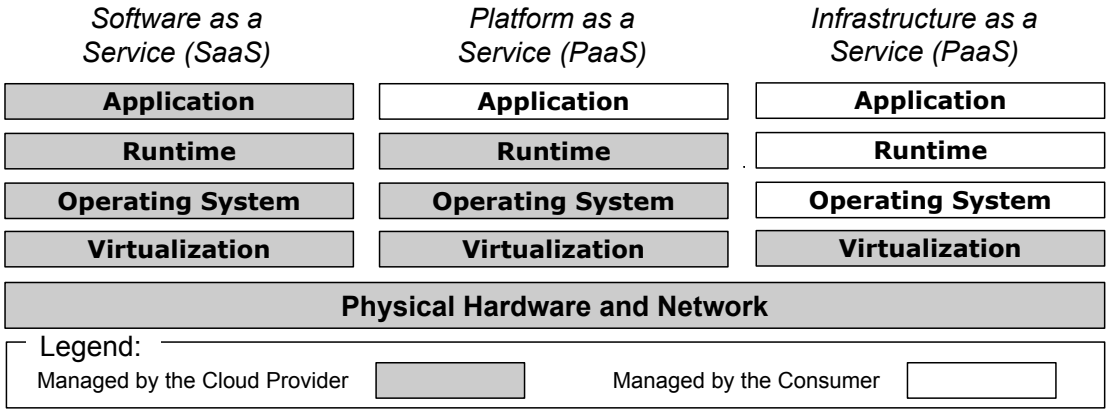


Figure 3.1 – Cloud Computing Service Model (Adapted from Schouten [35])

provided application.

Definition 2 (PaaS) *It is the service model where the cloud provider still manages the underlying cloud infrastructure, while consumers can run and deploy application on the Cloud in a restricted way.*

Definition 3 (IaaS) *It is the service model where the cloud provider only manages the layer of resource virtualization, while consumers can deploy arbitrary software.*

In the SaaS model, consumers cannot deploy and manage software. In the PaaS model, consumers have to the languages and tools supported by the cloud provider. In the IaaS model, consumers manage the operating system, virtual resources, such as network interfaces, and can deploy software using the language they choose. It is the model that most resembles a system deployment on physical machines.

3.1.2 Cloud Computing Elasticity

On the cloud, computing resources are allocated on demand [24]. This feature, called by several authors [19, 24, 48, 26] as *elasticity*, is the ability of a cloud infrastructure to vary computing resource as soon as possible, according to application demand.

Consumers can configure the *cloud computing elasticity controller*, i. e., mechanism that provides elastic resource allocation, in diverse manners. Galante and Bona classify elasticity controllers into four main categories [38]: *scope*, *policy*, *purpose*, and *method*. Figure 3.2 shows the classification proposed by them, each one with its sub-classifications.

The *scope* is where the elasticity controller controls the elasticity: *infrastructure* (IaaS), *platform* (PaaS) or *application* (SaaS). The *policy* is the type of elasticity man-

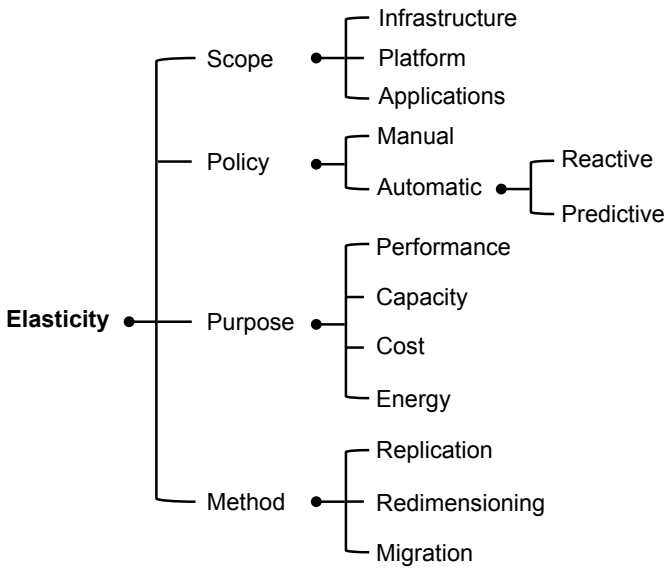


Figure 3.2 – Classifications of Elastic Mechanisms [38]

agement: *manual* or *automatic*. The *automatic* policy has two subdivisions: *reactive*, which reacts to a monitored factor, such as Central Processing Unit (CPU) usage, and *predictive*, which uses heuristic methods to predict the elasticity. The *purpose* of elasticity can be diverse, including *performance*, *capacity*, *cost*, and *energy* saving. Finally, the *method* is the manner the elasticity controller scales the computing resources: *replication*, *redimensioning*, or *migration*. The *replication* method consists in adding or removing instances on a virtual environment. The *redimensioning* method consists in resizing computing resources, such as CPU and memory, from a running virtual machine, i. e., vertical scaling. The *migration* method consists in transferring a Virtual Machine (VM) that is executing on a physical server to another one.

Elasticity Model

Bersani et al. [26] present a model for elasticity, where an elasticity controller allocates/deallocates VMs (by *replication*), in a *reactive* manner. The authors do not consider any specific purpose for the elasticity. Figure 3.3 depicts a high level view of the elasticity model they propose. Customers send their requests (i. e., workload) to the system deployed on the Cloud, i. e., CBS. Since customers are seasonal, the workload varies over time, which likewise varies the demand of computing resources. Avoiding resource overuse or underuse, the elasticity controller varies the amount of resource on demand. It monitors the cloud infrastructure, and decides whether scale-out (allocate) or scale-in (deallocate) a resource, based on preset resource usage thresholds. This strategy

is also a standard among commercial cloud computing infrastructures, such as Amazon Elastic Compute Cloud (Amazon EC2) and Google Cloud Platform.

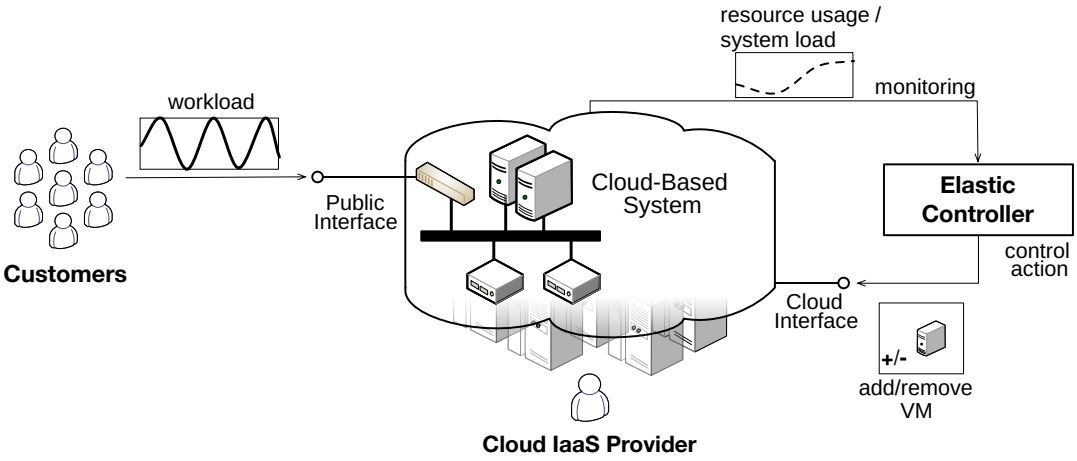


Figure 3.3 – High Level View of IaaS Elasticity (Adapted from Bersani et al. [26])

Figure 3.4 depicts a hypothetical workload that is used to detail Bersani et al. [26] model. This workload first has an intensity increase from 0 to ≈ 150 Operations per Second (OPS), and then it decreases to 0.

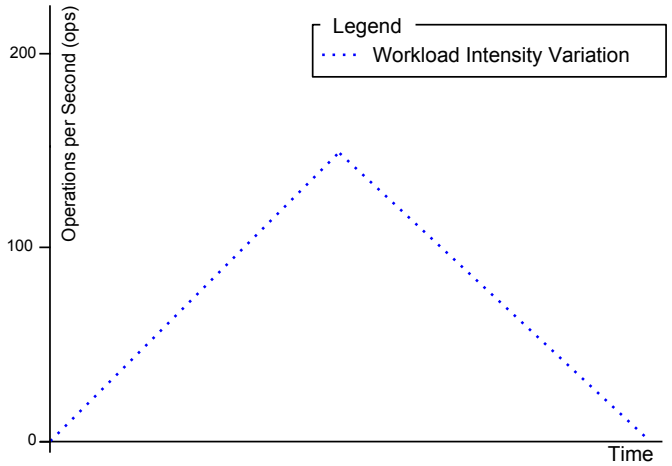


Figure 3.4 – Hypothetical Workload Intensity Variation

Figure 3.5 illustrates the resource demand variation for the hypothetical workload of the previous figure. In the figure, the allocated resource is 1 mono-processed VM. The workload below the *Allocated Resource* line means resource over-provision, while

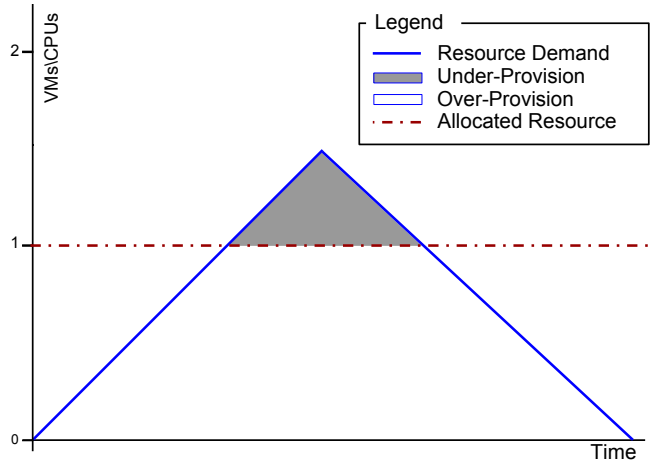


Figure 3.5 – Hypothetical Resource Demand Variation

the gray area above it corresponds to resource under-provision. The resource under-provision can be harsh for the consumer. Therefore, one could deploy the same CBS on 2 VMs. Considering the same workload, there would not have resource under-provision. However, most of the time the resource would be over-provisioned, what means wasting of resource. In this case, an elastic infrastructure (i. e., infrastructure that provides elasticity) could ameliorate the resource usage.

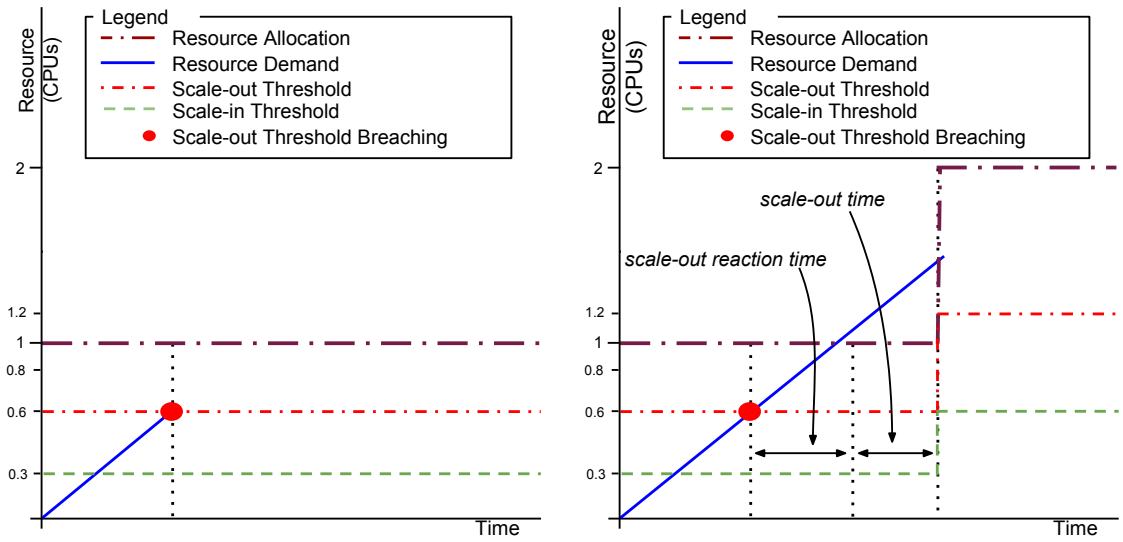
Figure 3.6 illustrates how an elasticity controller reacts to the hypothetical workload growth. The resource demand increases until breaching the *scale-out threshold*. If it breaches the scale-out threshold only temporary, the elasticity controller does not react to it. Otherwise, if it breaches the scale-out threshold for a preconfigured time, i. e., *scale-out reaction time*, the elasticity controller instantiates a new resource (see Figure 3.6b). The elasticity controller takes a while to instantiate a new resource and make it available, i. e., *scale-out time*. Once the resource is available, the elasticity controller considers new scale-in and scale-out threshold values, which refer to the new amount of resource.

Figure 3.7 illustrates the elasticity controller reaction to the hypothetical workload decreasing. In the figure, the resource demand becomes lower than the *scale-in threshold* (Figure 3.7a) for a while, i. e., *scale-in reaction time*, the elasticity controller releases a resource (Figure 3.7b). We call the period to release a resource as *scale-in time*. When a resource release begins, the resource is no longer available, and the elasticity controller considers new threshold values immediately.

Figure 3.8 depicts the complete scenario: a resource scale-out, and then a resource scale-in. Compared to Figure 3.5, the under-provision area is smaller. However, this is only an illustrative example that do not focus on an optimal resource allocation.

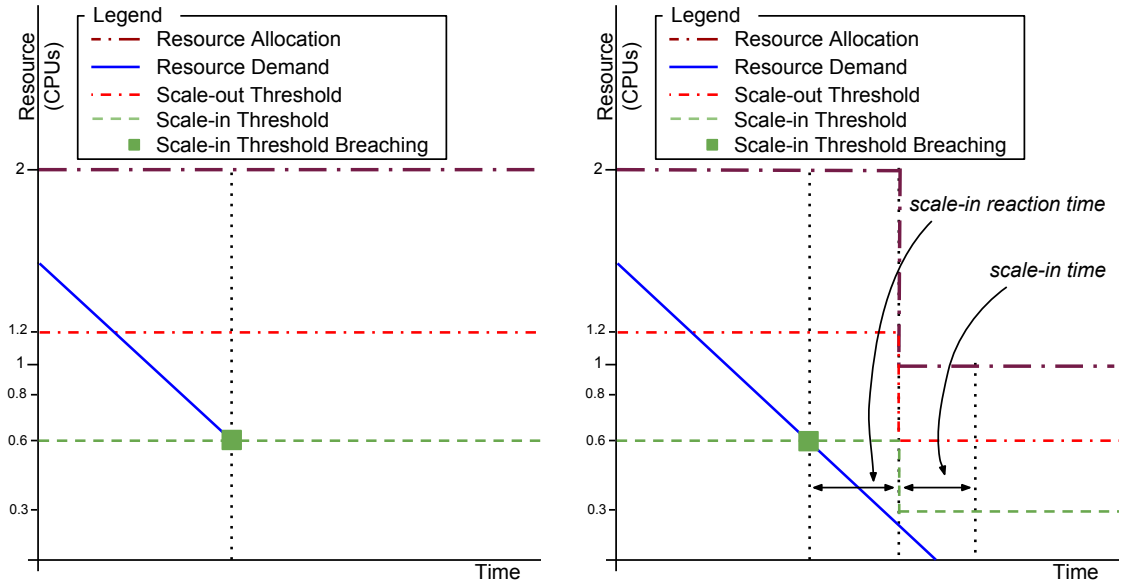
We propose the following, which we use all over this thesis:

— **Scale-out Threshold**: is the resource usage boundary used as a parameter to



(a) Scale-out Threshold Breaching **(b)** Scale-out Period

Figure 3.6 – Resource Scale-out



(a) Scale-in Threshold Breaching **(b)** Scale-in Period

Figure 3.7 – Resource Scale-in

trigger a resource scale-out.

— **Scale-out Reaction Time**: is the time during which a resource demand must breach the scale-out threshold to trigger a resource scale-out.

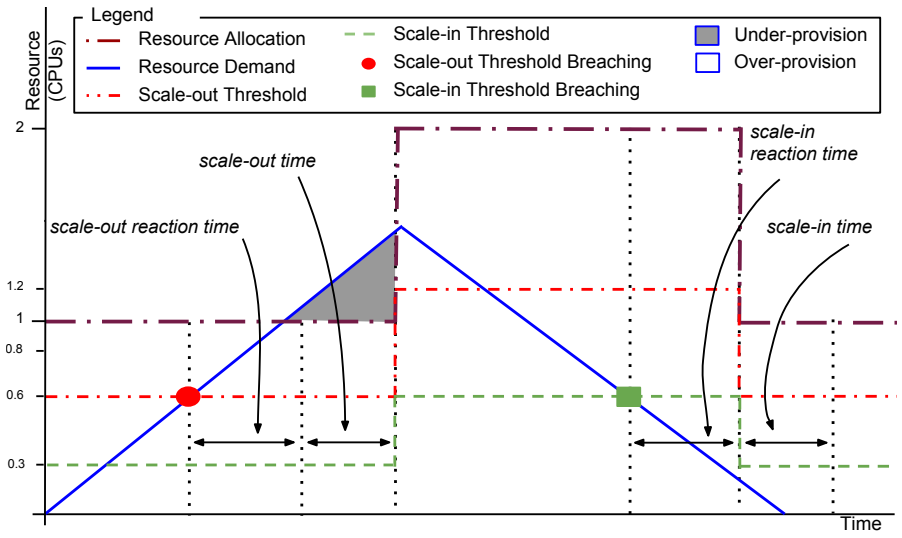


Figure 3.8 – Scale-out and Scale-in Periods

— **Scale-out Time**: is the time the elasticity controller takes to scale-out a resource.

— **Scale-in Threshold**: is the resource usage boundary used as a parameter to trigger a resource scale-in.

— **Scale-in Reaction Time**: is the time during which a resource demand must breach the scale-in threshold to trigger a resource scale-in.

— **Scale-in Time**: is the time the elasticity controller takes to scale-in a resource.

3.2 Software Testing

Many factors can cause a software to fail, such as a wrong design, hardware abnormalities, or unexpected input. Software testing raises software quality and reliability by identifying its fails [61]. There are two main processes used for software testing: static and dynamic testing. *Static testing* is a process that reviews the software design, architecture, or code for errors without executing it, while *dynamic testing* is a process that needs the software execution [6].

Figure 3.9 illustrates the dynamic testing process by dividing it into four main steps [5]. As a *first step*, testers design and implement tests, which consists in configuring the *test specifications* and the *test environment requirements*. Then, the *second step* consists in setting up testing environment according to requirements. Once the environment is ready, testers can execute the test specifications (*third step*). After test execution, the analysis of results may notice issues. In that case, tests report the issue as a *fourth step*.

Almeida [34] represents these processes in a more specific manner. Figure 3.10

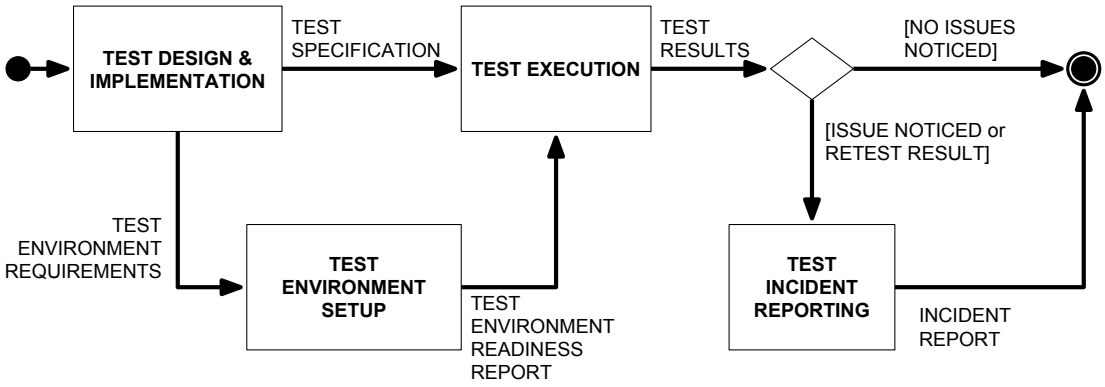


Figure 3.9 – Dynamic Test Processes [5]

illustrates this process. Initially, the tester writes a *test case*, which describes the *test specifications*. Then, the tester executes this test case on the system that is being tested, i. e., System Under Test (SUT), which is associated to the *test environment setup*. The test execution generates *test results* (*actual results*) that a *test oracle* compares to the *expected results*. Such comparison results in a *test verdict*. If the *actual* and the *expected* results match, the test passes, i. e., no issues noticed. Otherwise, the test fails.

Test cases are design to test functional or non-functional aspects of the SUT, i. e., functional or non-functional testing. *Functional* testing "verifies a program by checking it against SUT design document or specification" [53]. For instance, it sends an input to the SUT and checks whether the output is the expected. Non-functional testing is the testing of non-functional requirements of the SUT. For instance, the way a system operates, such as its performance, rather than specific behaviours of that system.

3.3 Test Case Generation

"The most important consideration in software testing is the design and creation of effective test cases" [61].According to Gambi et al., testers can write test cases manually or automatically by using a test case generation tool [39]. This section presents automatic strategies for test cases generation.

The number of test cases can be high depending on the SUT configuration domain, where each test case covers a SUT configuration. This makes the testing execution expensive [29]. Therefore, the number of test cases must be little, a problem which several work [29, 46, 63, 64, 62, 66, 27] solve by using Combinatorial Interaction Testing (CIT). In CIT, testers model the SUT as $M_{SUT} = \{P, V, C\}$, where P is the set of parameters $P = \{p_1, p_2, \ldots, p_n\}$, V is the set of parameter values $V = \{V_1, V_2, \ldots, V_n\}$ where V_i is a set of p_i values, and C is the set of constraints among parameters. Then, testers leverage a CIT algorithm that generates a set of test cases that covers all T

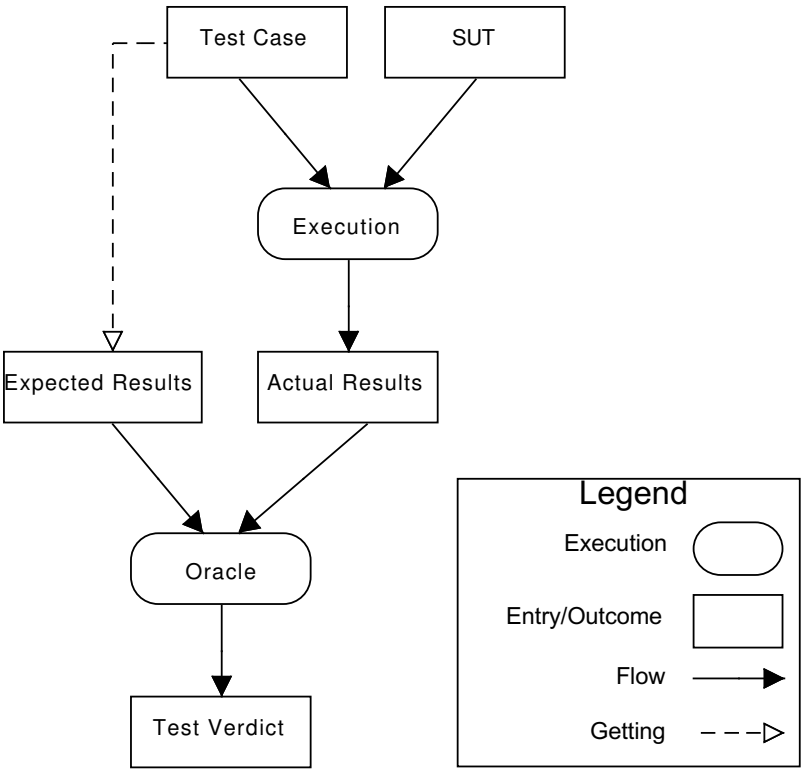


Figure 3.10 – Dynamic Software Testing Execution

combinations of parameter values and satisfies the constraints.

The survey proposed by Grindal et al. [46] ranks the main coverage degrees as: *N-wise*, *t-wise*, *pairwise* (or 2-*wise*), and *each-used* (or 1-*wise*). *N-wise* coverage, also known as *all-wise*, generates test cases that cover all possible combination of values of the *N* parameters. *t-wise* coverage requires the inclusion of every possible combination of *t* parameter values in a test cases. *Pairwise* (2-*wise*) coverage requires the inclusion of every possible combination of pair of parameter values in a test case. *Each-used* coverage is the simplest coverage criteria, which requires the inclusion of every parameter value in at least one test case in the test suite.

Figure 3.11 illustrates the test suite sizes by coverage degree, where test suites generated with the highest degree include all the test cases in test suites generated with the lowest levels.

Table 3.1 shows test suite sizes (in number of test cases) of coverage degrees for two examples.

1. 8 parameters ($N = 8$) with 4 possible values each ($V = 4$), and

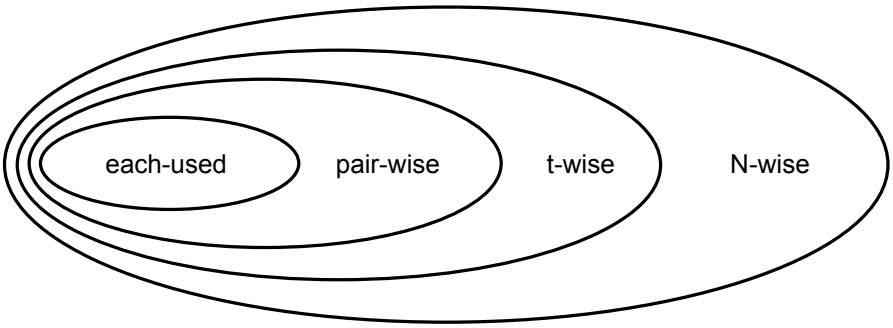


Figure 3.11 – Test Suite Size (Space) by Coverage Degree

2. 4 parameters ($N = 4$) with 8 possible values each ($V = 8$).

For illustrative reasons, in the examples, all the parameters have the same quantity of possible values. The test suite size for *N-wise* coverage is the product of all possible values $\prod_{i=1}^{N} |V_i|$. For *t-wise* coverage, the test suite size derives from the number of possible values for each one of the *t* parameters with the most choices. In the table, *3-wise* illustrates *t-wise*, where the test suite size calculation is $O(nml)$, where *n*, *m* and *l* are the number of possible values for each one of the three parameters with the most choices. For *pairwise* coverage, the test suite size calculation is $O(nm)$, where *n* and *m* are the number of possible values for each one of the two parameters with the most choices. Finally, the test suite size for *each-used* coverage is the maximum number of possible values $Max_{i=1}^{N} V_i$, where v_i denotes the number of possible values for the ith parameter.

Coverage Degree	**Test Suite Size Calculation**	**Example** 1 N=8, V=4	**Example** 2 N=4, V=8
N-wise	$\prod_{i=1}^{N} V_i$	65 536	4096
3-wise (t-Wise)	$O(nml)$	64	512
pairwise	$O(nm)$	16	64
each-used	$Max_{i=1}^{N} V_i$	4	8

Table 3.1 – Test Suite Sizes by Coverage Degree

In Table 3.1, test suite size reduces exponentiallyas coverage degree decreases. In both examples, given the number of test cases, executing test suites generated with the highest coverage degree (*N-wise*) is exhaustive. Nevertheless, test suites generated with the lowest coverage degree (*each-used*) may not be representative, which generates only 4 test cases for the first example.

Despite *pairwise* coverage only generates few test cases, it is a standard among software testing practitioners, and its adoption continues to increase [56]. One of the reasons is that even with few test cases, which represents less effort in executing them, empirical experiences demonstrate that it still reveals 50-75% of software faults [71, 33]. In the literature, pairwise coverage is used in distinct domains: Software Product Line (SPL) [64], software with high variability [62], self-adaptive systems [66], and scenario-based testing [27].

Perrouin et al. [64] use pairwise coverage to generate test cases for SPL products. The authors use Feature Model (FM) [54] to represent product features and dependencies, formalizing them by using the constraint language *Alloy* [59]. Then, they use a constraint solver to generate valid test cases from the FM with pairwise coverage.

Nguyen et al. [62] propose an approach that combines model-based and combinatorial testing. The authors convert system Final State Machine (FSM) paths into a CTM [28]. CTM are also known as predictive models, where data observations are modeled as dependent variables. They are used to structure system features into a hierarchical tree, where the root element is the system, and the branches (*compositions*) represent its components. Non-decomposable components are defined as *classifications*, structured in *classes* that implement them. In their approach, Nguyen et al. generate test cases by covering pairwise combinations of classes of the CTM.

Sen et al. [66] present an approach for self-adaptive systems. Their approach generates a sequence of test configurations (or test cases) to evaluate the extent to which reconfigurations affect system QoS. The authors also model system parameters as a FM, using pairwise combinations to generate test cases. Then, their approach orders the test cases into a test sequence that mimics system reconfigurations.

Finally, Bousquet et al. [27] propose the use of pairwise coverage to generate test suites for scenario-based testing. Scenario-based testing requires a sequence of actions to test an application. The authors consider these actions as ordered method call instances, denoted factors. Therefore, a test case is composed of a sequence of factors. For test case generation, they extend classical pairwise coverage into method call instantiation coverage, where they cover pairwise combinations among possible factor values.

3.4 Cloud Automation

Cloud computing adds an extra layer of complexity to the process of software development [70], and consequently, to CBS testing. It requires specific interactions with the cloud provider: the tester must request the needed computing resources, deploy the CBS, and configure the CBS and the elasticity parameters. Testers can write deployment scripts that use cloud provider APIs for such interactions. However, cloud provider APIs are heterogeneous, which requires the tester to re-write the deployment scripts when testing is executed on different cloud providers.

Several research efforts from both, research [70, 72, 40] and industry [17, 18] communities, aim at softening the deployment complexity by proposing frameworks or/and languages for the deployment and management of CBS on the Cloud.

Sledziewskiz et al. [70] presents a DSL for software deployment on the Cloud. In their approach, the user describes his software in a DSL, whose editor is available as a SaaS. Then, software description is compiled to code that is used by their own deployment tool to deploy software onto the Cloud.

Thiery et al. [72] propose a DSL to set up the deployment of CBS in a provider-independent manner. Their DSL is divided into three main parts: the first part describes available cloud provider resources, the second part describes software components that must be deployed, and the third part describes the resource requirements for each software component. Programs written in this DSL are compiled into executable code, which can be used to interact with the cloud provider. When deploying CBS on other cloud providers, the only part that must be changed is the one that describes the cloud provider resources .

Gambi et al. [40] present the AUToCLES [40], a Test-As-A-Service (TaaS) in the form of black-box system testing. AUToCLES implements all the functionalities required to manage the lifecycle of test executions on the Cloud. Indeed, testers use a Web interface to provide test specifications and elastic system configuration, and AUToCLES automates all the process. AUToCLES requires that applications are self-managing, that is, they have all the logic to automatically scale up or down.

Chef [17] is an open source framework for infrastructure automation, which abstracts the management of infrastructure resources using configuration units called *cookbooks*. Chef is composed of three main components: Chef development kit (DK), Chef server, and Chef client. The Chef DK is a set of tools used to code infrastructure automation, and to interact with Chef server and the Chef clients. The Chef server is a central repository, where cookbooks are stored, and where they are read from by the Chef clients. Chef client is an agent that runs locally on every node that is being managed by Chef. It is also the component that receives user interactions, and where the cookbooks are ran.

Puppet is a commercial framework that works in the same manner as Chef, where the user writes code that automates infrastructure management and software deployment. In Puppet, this code is called *manifest*. In their Website [18] they state that it "provides a standard way of delivering and operating software, no matter where it runs." Using a specific language, the user sets up the software components she wants to execute, component configurations, and how the infrastructure looks like. Puppet runs in an agent/master architecture, where servers run Puppet master, and managed nodes run Puppet agents. Puppet master stores the manifests, while Puppet agents, which run on managed nodes, poll the master on a given schedule, checking for differences in the manifests, and applying them. The user can also interact with Puppet agents and directly request them to execute a manifest.

Table 3.2 summarizes the cloud automation work presented in this section. Two of these work are purely DSL [70, 72], and Chef and Puppet are frameworks for infrastructure automation that also propose DSLs. AUToCLES is a TaaS, it uses a Web interface for elastic configurations rather than a DSL.

One of the goals of this thesis is to abstract the implementation of elasticity testing. Out of the approaches for cloud automation, all of them support infrastructure automation and software deployment. However, they do not support the elasticity configuration, only the AUToCLES does it, but at application level.

	Approach	Infrastructure Automation	CBS Deployment	Elasticity Configuration
Sledziewskiz et al. [70]	*DSL*	*YES*	*YES*	*NO*
Thiery et al. [72]	*DSL*	*YES*	*YES*	*NO*
AUToCLES [40]	*TaaS*	*YES*	*YES*	*at Application Level*
Chef [17]	*Infrastructure Automation*	*YES*	*YES*	*NO*
Puppet [18]	*Infrastructure Automation*	*YES*	*YES*	*NO*

Table 3.2 – Summary of Cloud Automation Approaches

3.5 Testing Elastic Cloud-Based Systems

This section presents different research efforts on elasticity testing. Section 3.5.1 starts by discussing elasticity metrics, which positions the existing concerns on CBS elasticity. Then, Section 3.5.2 discusses the research efforts on elasticity testing.

3.5.1 Elasticity Metrics

Weinman [75] propose two elasticity metrics: over-provisioning and under-provisioning. Figure 3.12 illustrates these metrics. In the figure, a hypothetical resource demand curve ($D(t)$) indicates the needed resource. The dotted line represents the resource allocation at each instant ($R(t)$), which grows linearly. For Weinman, an idealistic elasticity would be $R(t) = D(t)$. Therefore, he identifies situations where resource exceeds the demand (over-provisioning, where $R(t) > D(t)$) or the resource is less than the demand (under-provisioning, where $R(t) < D(t)$). *Over-provisioning* refers to how much money the consumer wastes by paying for resource that is not being used by the current workload. In contrast, *under-provisioning* refers to how much the resource shortage affects the consumer.

Islam et al. [51] base their metrics on Weinman's ones. In addition to Weinman's metrics, Islam et al. propose financial penalties for over-provisioning and under-provisioning periods. The basis of their over-provisioning penalty model is the difference between chargeable (i. e., resource that the user pays for) and demanded resource. In their under-provisioning penalty model, customer sets financial penalties for performance

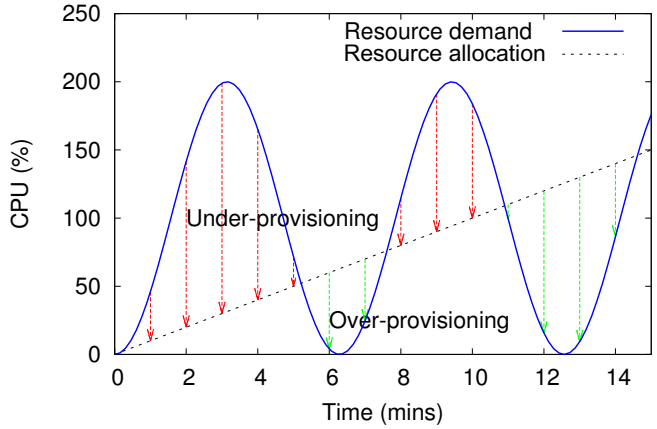


Figure 3.12 – Resource Under and Over-Provisioning [75]

degradations, e. g., a dollar value for each percent of rejected requests. Then, they calculate a total penalty rate per execution, which sums up all the over-provisioning and under-provisioning penalty scores.

Shawky and Ali [68] base their elasticity metrics on material's elasticity ones. The authors define physics elasticity as "the physical property of a material that returns to its original shape after the stress". Its notation is

$$E = \frac{Stress}{Strain} \tag{3.1}$$

where *Stress* is the measure of how strong a material is. That is, how much pressure the material can stand without physical changes. *Strain* is the measure an object stretching.

For cloud computing elasticity, Shawky and Ali model *stress of cloud* as the ratio between new demands of computing capacity and the allocated computing capacity. To measure the *strain of cloud*, the authors use the change in the average network bandwidth (in and out) and the time to allocate/deallocate a resource. Both measurements comparison before and after a resource scaling indicates how elastic a cloud infrastructure is.

Coutinho et al. [31] propose metrics for performance analysis of CBS through elasticity. Their metrics consider the *response time* of elasticity operations and *used resources*. Basically, their metrics measure the time and the used resources through four different states: *under-provisioning*, *over-provisioning*, *stable*, and *transitory*. *Under-provisioning* is the period while a resource is being added. *Over-provisioning* is the period while a resource is being removed. *Stable* is the period while a resource is stable (neither addition nor removal). *Transitory* is the period the effects of addition or removal of resources are still in progress.

Summary of Elasticity Metrics

Table 3.3 summarizes the metrics presented in this section. This thesis aims at testing CBS throughout elasticity. However, all the existing metrics described in this section only propose how to measure cloud computing infrastructure elasticity. Most of the metrics compare the demand and the allocated resources. The only metric related to the CBS is the response time, used by Coutinho et al., who also introduce the elasticity measurement during different CBS states.

	Elasticity Purpose	**Metrics**	**Measures**
Weinman [75]	*resource allocation*	*over-provisioned, and under-provisioned resources*	*demanded and allocated resource*
Islam et al. [51]	*cost*	*over-provisioning penalty, under-provisioning penalty, and total penalty rate*	*chargeable and demanded resource, and performance degradation*
Shawky and Ali [68]	*resource allocation*	*stress* and *stain of cloud*	*demanded and allocated resources, network bandwidth, and time for resource changes*
Coutinho et al. [31]	*performance, and resource allocation*	*response time, and used resource through different states*	*response time, and allocated resource*

Table 3.3 – Summary of Elasticity Metrics

3.5.2 Test of Elastic Cloud-Based Systems

Out of testing approaches for cloud computing, several work execute tests on the Cloud [25, 49], while others test the Cloud [30, 32, 47, 69, 77, 76, 37, 45, 44]. The first group of work use cloud infrastructure as resource pool to execute tests that require huge amounts of computing resource, which is out of this thesis scope. Part of the work for CBS testing focus on benchmarking CBSs [30, 32, 47, 69, 77] rather than effectively testing them, while other part focuses on testing the cloud infrastructure reaction to resource demand [76, 37, 45, 44]. This section only presents the work that effectively test CBSs throughout elasticity, which are not numerous[52, 74, 42, 41].

Islam [52] investigate the effect of fine-scaled workload burstinesson the elasticity of cloud based Web applications. They propose a manner to model fluctuations into the arrival of existing workloads, which makes such workloads highly variable at fine timescale. They experiment a Web application (TPC-W [12]), evaluating how fine-scaled burstiness affects resource allocation and the Web application performance. In their experiments, fine-scale burstiness affects the case study in a condisderable manner.

While the main focus of Vasar et al. [74] work is not to test CBS they propose a strategy to test the scalability of Web applications throughout elasticity. The main contribution of their work is a predictive algorithm that auto-scales Web servers based

on weighted average of requests arrival rate. They generate the workload based on trace logs [2], comparing the scalability of Web application when using standard reactive elasticity and their algorithm. In their experiments, the authors consider the reaction time and maximum throughput.

Gambi et al. introduce novel ideas on CBS testing, identifying resource challenges, and given future direction on this field [42]. They use elastic physics materials metaphor, where the SUT is the material, the workload is the stress factor, and the changes in the resource is the material deformation, mapping mechanical testing metaphor to analogies in elastic CBS. The authors also propose a conceptual framework for CBS testing. Their framework has four main components: test case generation, test execution, data analysis, and test evolution. Test case generation receives testing goals as input and produces a set of test specifications as output. Then, these specifications lead the CBS testing, and the testing result analysis sometimes requires refining test specifications. This whole process is cyclically (re-)executed.

In a second work, Gambi et al. [41] distinguish two types of system strain: *fair operations* that are compatible to the system resources, and *excessive operations* that overload the system. They consider that it is easy to break a system by stressing it with excessive load, while a system can break even when it receives fair operations. However, finding a right sequence of fair operations to break a system is a complex task. Therefore, they aim at generating test cases that expose systems to fair operations, causing multiple resource changes, to identify faults related to elasticity. They combine surrogate models, that abstract system description, and search-based methods to find specific test cases that break the system.

Summary of CBS Testing Approaches

Table 3.4 summarizes the work discussed in this section. Gambi et al. [42] work is a conceptual testing framework. This work gives directions for part of the approaches that we propose in this thesis, such as the approaches for CBS driving throughout elasticity and elasticity tests generation. The other work are all related either to scalability or performance testing, and their elasticity purpose is a miscellaneous. Islam et al. and Vasar et al. work generate the workload based on log traces, while Gambi et al. [41] generates the workload by using genetic algorithms. Therefore, no work proposes a way to drive the CBSs in a deterministic way, which is one of the problematics this thesis resolves. Finally, none of them covers the other problematics.

3.6 Conclusion

Cloud computing elasticity is a new trend that introduces many challenges in software testing, such as the ones discussed in the thesis introduction. Only few work in the

	Type of Testing	Elasticity Purpose	Workload Generation
Islam et al. [52]	*performance testing*	*cost* and *QoS*	*based on log traces*
Vasar et al. [74]	*scalability testing*	*response time* and *throughput*	*based on log traces*
Gambi et al. [42]	*conceptual testing*	–	–
Gambi et al. [41]	*scalability testing*	*resource efficiency*	*genetic algorithms*

Table 3.4 – Summary of Elasticity Testing Approaches

literature focus on elasticity testing, and they do not cover the problematics tackled in this thesis. The work for cloud automation lack of procedures for elasticity setup. Among the work that propose elasticity metrics, Coutinho et al. one is the only work that considers CBS testing. The authors also propose a manner to test CBS throughout different states. However, they identify such states by a post-execution strategy. The approaches for elasticity testing are diverse, where none of them propose a manner to lead CBS throughout elasticity in a deterministic way.

4

Driving Cloud-Based Systems (CBS) Throughout Elasticity

Elastic cloud infrastructures vary resources (allocate, or deallocate) at CBS runtime. To deal with these variations, CBS must behave in an elastic manner, i. e., adapt itself for the new resource configuration. If the CBS is not designed in such way, it may fail or its performance may not be adequate.

Testers should be able to drive CBS in a deterministic way, controlling its elastic behavior. Thus, they can be more specific and model situations they judge as critical. Furthermore, this can also reduce testing execution time since the elasticity behavior 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.

To manage resource changes, the tester can interact with cloud provider directly. However, in a real-world scenario elasticity derives from workload variations, which changes the resource demand. Then, the elasticity controller reacts to new resource demands, and allocates or deallocates resources.

This chapter proposes an approach to control CBSs driving throughout elasticity states by varying the workload accordingly. Section 4.1 defines elasticity states after the typical CBS elasticity. Section 4.2 describes an approach for CBS driving. Section 4.3 describes a prototype that implements the approach for CBS driving. Section 4.4 describes the results of experiments conducted by using the prototype. Finally, Section 4.5 concludes, and gives directions of future work.

4.1 Elasticity States

We identify three *elasticity states* in the elastic behavior presented in Figure 3.8 : *scaling-in* (si), *scaling-out* (so), and *ready* (ry), which contains the substates *stable* (ry_s), *scale-out threshold breaching* (ry_sor), and *scale-in threshold breaching* (ry_sir).

Figure 4.1 depicts these states and their transitions in a Unified Modeling Language (UML) State Machine diagram. At the beginning, the CBS enters into the *ready* state (ry), when the resource configuration is steady (ry_s substate). Then, if the CBS is exposed to a workload that breaches the *scale-out threshold* (sub-state ry_sor) during the *scale-out reaction time*, the elastic controller starts adding a new resource. At this point, the CBS enters the *scaling-out* state (so) and remains in this state while the resource until the elasticity controller finishes adding the new resource. After a *scaling-out*, the application returns to the *ready* state. Then, when the workload leads the resource demand to a level that breaches the *scale-in threshold* during the *scale-out reaction time* (ry_sir substate), the elastic controller starts removing a resource and the CBS enters the *scaling-in* state (si).

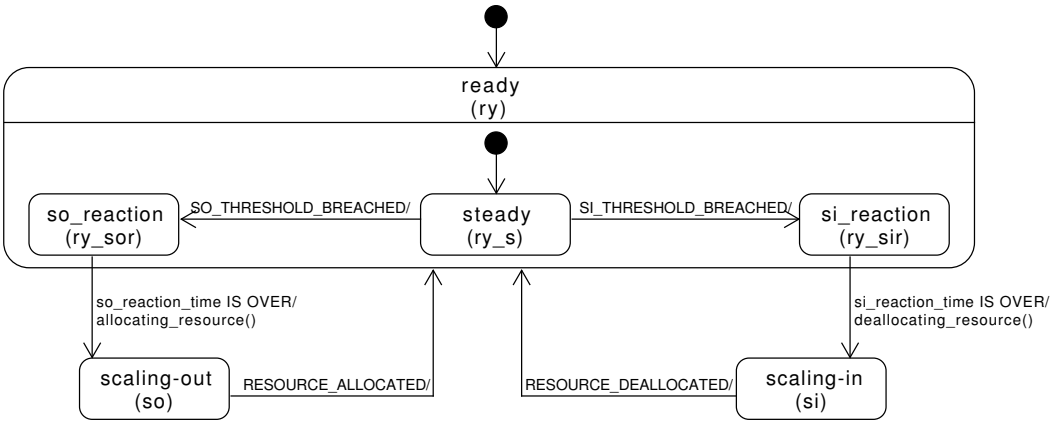


Figure 4.1 – Elasticity States

4.2 CBS Driving Approach

This section describes the approach for CBSs driving throughout a preset list of elasticity states. Elasticity state transitions occur due to workload variations that eventually breach the thresholds. Therefore, this approach proposes a way to generate the workload variations in a deterministic manner.

Gambi et al. consider that an input workload has three characteristics [42]: *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 of time. Given a workload type, this approach calculates the *requests intensity variation* that should drive the CBS throughout a preset list of elasticity states.

Figure 4.2 depicts the approach workflow, which has three execution phases: *workload profiling* (Section 4.2.1), *workload calculation* (Section subsec:workloadcalc), and *application leading* (Section 4.2.3).

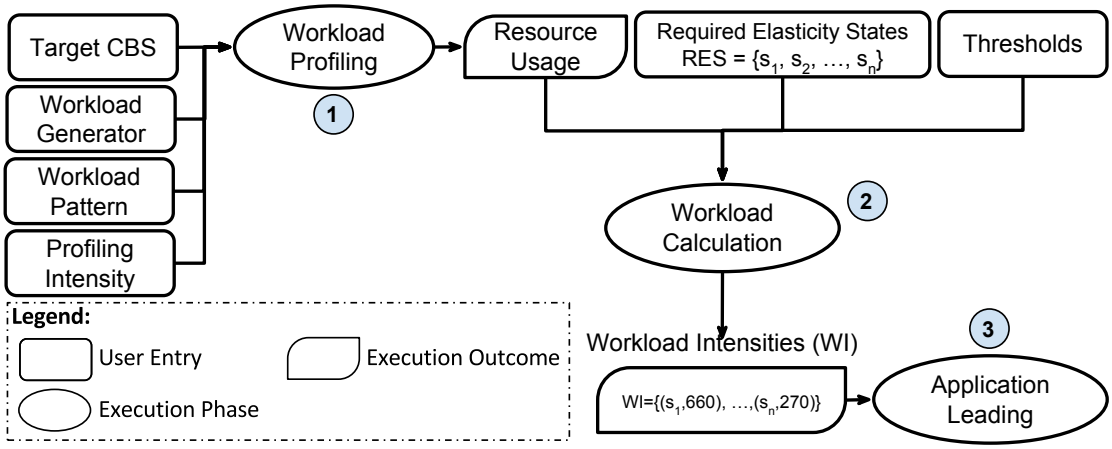


Figure 4.2 – CBS Driving Procedure Workflow

4.2.1 Workload Profiling

CBSs may react in a distinct way when exposed to the same workload [26]. Therefore, before calculating the workload variation, we must profile each combination of CBS and workload.

The *workload profiling* phase has four parameters: the *target CBS*, the *workload generator*, the *workload type*, and *Profiling Intensity (PI)*. The *target CBS* is any application that runs on the Cloud and supports 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, 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.

4.2.2 Calculation of Workload Variations

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

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 ($\top$). The scale-in multiplier, denoted by M_{si}, is the workload intensity that breaches the scale-in threshold ($\perp$).

The following equation calculates the scale-out multiplier:

$$M_{so} = \frac{PI \cdot \top}{ARU} \tag{4.1}$$

The following equation calculates the scale-in multiplier:

$$M_{si} = \frac{PI \cdot \perp}{ARU} \tag{4.2}$$

Table 4.1 shows an example of multipliers. For their calculations, we hypothetically consider that the PI value is 100, the ARU value is $10\,\%$ of CPU usage (i. e., *0.1*), the scale-out threshold ($\top$) corresponds to $60\,\%$ of CPU usage (i. e., *0.6*), and the scale-in threshold ($\perp$) corresponds to $30\,\%$ of CPU usage (i. e., $0{,}3$). Thus, M_{so} value is equal to 600, while M_{si} is equal to 300.

Threshold	Threshold Value	Multiplier
$scale\text{-}out$	$60\%\ of\ CPU\ usage$ (0,6)	$M_{so} = 600$
$scale\text{-}in$	$30\%\ of\ CPU\ usage$ (0,3)	$M_{si} = 300$

Table 4.1 – Example of Multipliers

Workload Intensities

Before calculating the workload intensities, we must choose how to variate the workload, where its variation may follow one of the following strategies: (i) gradual ramp up or down, (ii) random, (iii) based on log traces, and (iv) in stages. Since a gradual ramp up and down variation (i) either stresses the CBS or lasts too long, we discard it. Since the CBS driving methodology aims at controlling the elasticity according to a preset elastic behavior, we also discard the random (ii) and base on log traces (iii) strategies. Therefore, we choose to variate the workload in stages (iv), which allows to control of the CBS elastic behavior.

Figure 4.3 depicts the possible downsides of using ramp up workload variation. The figure depicts two situations: *hurried rate* (Figure 4.3(a)), and *lazy rate* (Figure 4.3(b)). In a *hurried rate*, a scale-out threshold breaching occurs right after the previous resource scale-out completes, which requests resource changes as soon as possible. For that, the workload increases in a rate determined by the gradient formula as follows:

$$HR = \frac{M_{so}}{x} \tag{4.3}$$

where x represents the total time for scale-out a resource (i. e., $x = \Delta T_r^{so} + \Delta T^{so}$). During the *hurried rate* the resource is often under-provisioned, what potentially stresses the CBS. To avoid the resource under-provision, we should increase the workload using the *lazy rate*, calculated as follows,

$$LR = \frac{M_{so}}{2 \cdot x} \tag{4.4}$$

which delays the threshold breaching $(2 \cdot x)$, and never stresses the CBS. However, depending on the RES, CBS driving could take too long, what makes it time–and perhaps–cost prohibitive, since in the Cloud long periods can result in high cost.

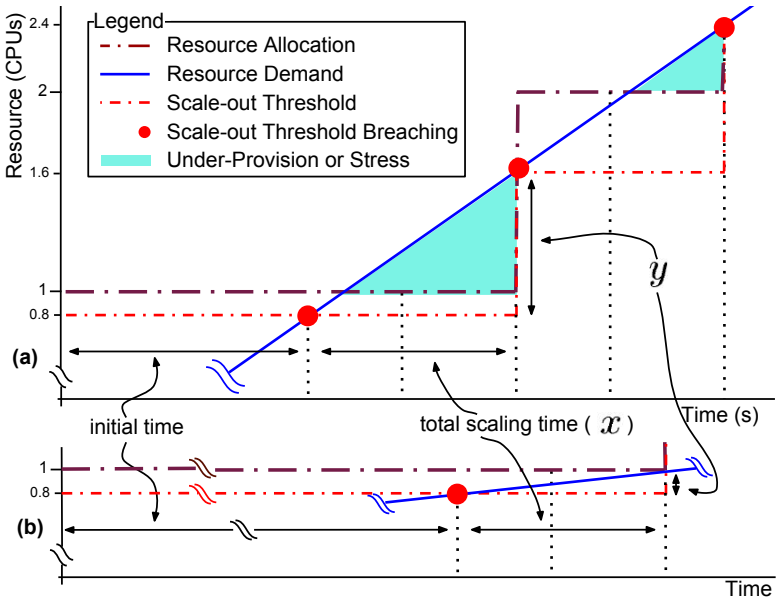


Figure 4.3 – Hurried Rate (a) and Lazy Rate (b)

Varying the workload *in stages* consists in generating a fixed workload intensity per period of time, which corresponds to the elasticity state in RES. We calculate a workload intensity for each elasticity state in RES, either breaching or not the thresholds. Indeed,

the intensity should be M_{so} for a scaling-out state, M_{si} for a scaling-in state, breaching the respective thresholds, and between M_{so} and M_{si} for a *ready* state, when we should not breach any threshold. However, since *scaling* states (*scaling-out* and *scaling-in*) change the amount of resources over time, the amount of Allocated Resource (AR) is a key parameter. In this case, considering a linear scalability, which is the case of distributed Web applications [58], to breach the thresholds one only need to multiply M_{so} and M_{si} by AR. We call the product of this multiplication as *current multiplier* (CM), where $CM_{so} = M_{so} \cdot AR$ and $CM_{si} = M_{si} \cdot AR$. However, external factors that affect resource consumption, such as network latency and concurrent processes, may disturb the CBS driving. Therefore, we also consider a percentage of resource variation, which we call as *predictive variation index*, denoted by ι. The approach calculates the workload intensities as following:

$$WI^{ry\_s} = CM_{so} \cdot (1 - \iota \cdot 2) \tag{4.5a}$$

$$WI^{ry\_sor} = CM_{so} \cdot (1 + \iota) \tag{4.5b}$$

$$WI^{so} = CM_{so} \cdot (1 + \iota) \tag{4.5c}$$

$$WI^{ry\_sir} = CM_{si} \cdot (1 - \iota) \tag{4.5d}$$

$$WI^{si} = CM_{si} \cdot (1 - \iota) \tag{4.5e}$$

In the *ready* sub-state ry_s, the thresholds are not breached and the resource remains unchanged. Therefore, the workload intensity for this state (WI^{ry_s}) must lead the resource usage to a level $\iota \cdot 2$ lower than CM_{so} (Equation 4.5a). Thus, the workload intensity leads the resource demand close to the scale-out threshold, a significant amount of work, but without breaching any threshold. However, one could use any intensity that keeps resource demand between scale-out and scale-in thresholds. The goal here is this intensity to result in a high amount of work, forcing the CBS to process high amounts of data.

The *ready* sub-state ry_sor is when the workload breaches the scale-out threshold. To ensure that the workload intensity breaches such threshold, it is ι percent higher than CM_{so} (Equation 4.5b). The workload intensity for the *scaling-out* state (Equation 4.5c) must keep the scale-out threshold breached while a new resource is allocated.

The *ready* sub-state ry_sir is when the workload breaches the scale-in threshold. We calculate the workload intensity for this state as ι percent lower than CM_{si} (Equation 4.5d). This keeps the resource demand just below the scale-in threshold. We use the same calculation for *scaling-in* state (Equation 4.5e).

To exemplify these calculations, we use the following RES:

$$RES = \{s_1^{ry\_sor}, s_2^{so}, s_3^{ry\_sor}, s_4^{so}, s_5^{ry\_sir}, s_6^{si}, s_7^{ry\_sir}, s_8^{si}, s_9^{ry\_s}\} \tag{4.6}$$

Table 4.2 shows the sequence of states for the example RES, the amount of resource during each state, and the workload intensity we arbitrarily use $10\,\%$ (i. e., $0{,}1$) as ι value. In this example, we consider that the initial and minimum resource amount is 1, and this amount varies according to the elasticity state. For instance, the RES state s_2 is a *scaling-out* state, when the resource amount increases. Then, at the following RES state (s_3), the resource amount is 2.

RES State	Elasticity State/Substate	Resource Amount	Workload Intensity
s_1	ry_sor	1	660
s_2	so	1	660
s_3	ry_sor	2	1320
s_4	so	2	1320
s_5	ry_sir	3	810
s_6	si	2	540
s_7	ry_sir	2	540
s_8	si	1	270
s_9	ry_s	1	270

Table 4.2 – Example of Workload Intensities Values

The output of the *workload calculation* is a set of pairs $(elasticity\ state, intensity)$, as the following set (extracted from Table 4.2).

$$\begin{aligned} WI = \{&(ry\_sor, 660), (so, 660), (ry\_sor, 1320), (so, 1320), \\ &(ry\_sir, 810), (si, 540), (ry\_sir, 540), (si, 270), (ry\_s, 270)\} \end{aligned} \tag{4.7}$$

4.2.3 Application Leading

In the *application leading* phase, we lead the CBS using the calculated workload intensities (WI). Algorithm 1 illustrates this approach phase. We expose the CBS to each workload intensity until the elasticity state ends. We monitor cloud infrastructure periodically to identify the elasticity states.

Figure 4.4 depicts the relation between workload and resource variation. The figure is only illustrative, where we hypothetically consider all the elasticity states last for the same time, i. e., 60 seconds. In the figure, we consider the workload intensities listed in Table 4.2, where each time slot between vertical lines is an elasticity state in RES. The solid line represents both the workload variation (left axis y) and the resource demand variation (right axis $y2$). The dashed line represents the variation in the amount of AR (right axis). We see that AR amount increases from 1 to 3, then decreases to 1.

Algorithm 1: Application Leading

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

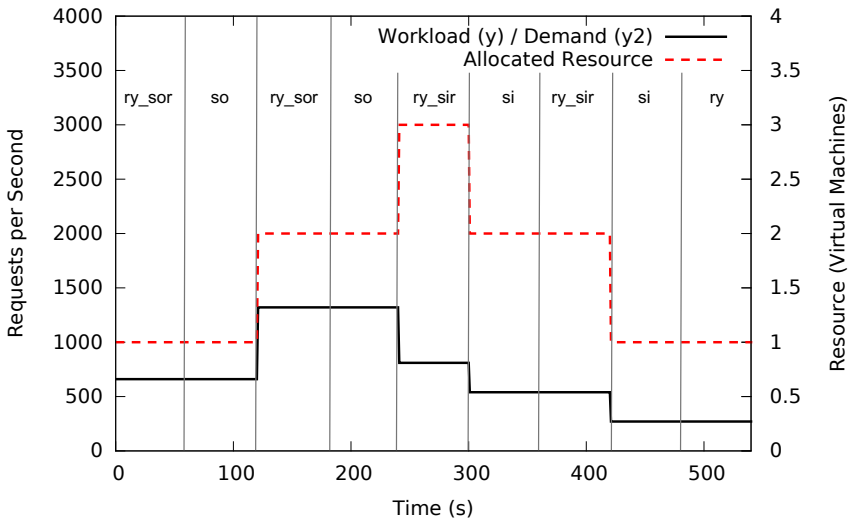


Figure 4.4 – Example of Workload and Resource Variation During Application Leading

4.3 Elasticity Driver Prototype

This section presents a prototype that implements the CBS driving approach, which has two components: a *coordinator*, and several *generators*. These two components communicate by using remote calls (e. g., Remote Method Invocation (RMI)), which allows to deploy them on distinct VMs.

The *coordinator* has several roles: front-end, monitoring, calculation of workload intensities and synchronization of load tasks. Each *generator* runs and controls an instance of a *load generation application*, such as a benchmark tool, and leads it to execute the load tasks received from the coordinator, i. e., workload type and intensity. Using multiple generators we reproduce a more realistic workload, which also allows us to generate high workload intensities.

Figure 4.5 shows an UML sequence diagram that represents the application leading phase. The *coordinator* starts by monitoring the resource status on the *cloud frontend*,

identifying the elasticity states. In the sequence, the *coordinator* balances the workload generation among the multiple *generators*. Then, the *generators* keep generating the same workload until receiving another instruction from the coordinator, which occurs every time the *coordinator* identifies a new elasticity state. This process is repeated for all pairs in WI.

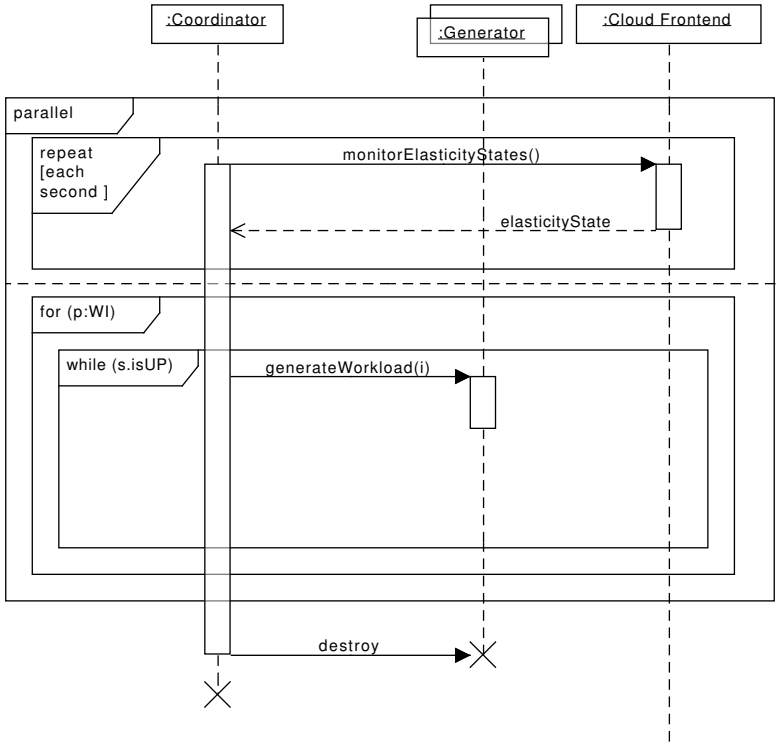


Figure 4.5 – Sequence Diagram of Application Leading Phase

In the current version, we implement the prototype in Java, where all the parameters are configured in a Java property file. The generators can run any load generation application that has an API, or supports command line execution. We identify elasticity states and monitor resource usage by using cloud provider's API, which allows us to get instances' information, such as resource usage statistics and initialization status.

4.4 Experiments

In this section, we present and discuss the three experiments we conduct. The first experiment (Section 4.4.4) aims at confirming the assumption that driving CBS

throughout RES by using a gradual ramp up workload variation in a *hurried rate* stresses the CBS. The second experiment (Section 4.4.5) aims at confirming the assumption that driving CBS throughout RES by using a gradual ramp up workload variation in a *low rate* takes too long. Finally, the last experiment (Section 4.4.6) aims at controlling the feasability the CBS driving approach.

4.4.1 Cloud Infrastructure Setup

We conduct the experiments on Amazon EC2 cloud infrastructure, with the *scale-out threshold* as 60% of CPU usage, and the *scale-in threshold* as 30% of CPU usage. We configure both *reaction times* as 60 seconds.

Table 4.3 describes the configuration of each VM type used in the experiments.

Machine Type	CPU	Memory	Disk
$m3,medium$	1 vCPU (2,4 GHz)	3,7 GB	10 GB
$m3,large$	2 vCPU (2,4 GHz)	7,5 GB	10 GB

Table 4.3 – Configuration of Amazon EC2 Virtual Machines

4.4.2 Case Study

As a CBS case study, we use the PHP version of the *RUBBoS*, a realistic web application modeled after an online news forum like Slashdot [10]. Figure 4.6 depicts the *n*-Tier deployment of RUBBoS in the experiments. In a *m3.large* VM, we deploy

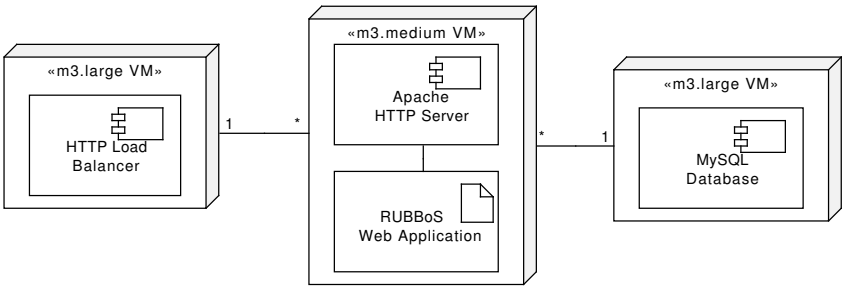


Figure 4.6 – RUBBoS n-Tier Architecture

the HAProxy [1] Hypertext Transfer Protocol (HTTP) load balancer, which distributes the requests among Web servers. The Web servers host RUBBoS, and run on *m3.medium*

1. http://www.haproxy.org/

VMs. It uses a centralized MySQL database server, deployed on a *m3.large* VM. The elasticity controller scales Web server VMs during the experiments, which initial amount of this resource is 1.

4.4.3 Workload Generation

As *load generation application*, we use the RUBBoS benchmark tool. We choose the workload type *browse*, and vary the workload intensity by number of clients, whereas for the workload profiling phase, we use 100 clients ($PI = 100$).

Table 4.4 describes the threshold and multipliers for the experiments. In the workload profiling phase, the ARU values is $40\,\%$ of CPU usage ($ARU = 0{,}4$). Therefore, the workload *multipliers* are $M_{so} = 150$, and $M_{si} = 75$.

Threshold	Threshold Value	Multiplier
$scale\text{-}out$	$60\%\ of\ CPU\ usage$ (0,6)	$M_{so} = 150$
$scale\text{-}in$	$30\%\ of\ CPU\ usage$ (0,3)	$M_{si} = 75$

Table 4.4 – Multipliers for the Experiments

4.4.4 Gradual Workload Variation in a Hurried Rate

In this experiment, we drive the CBS case study by using a workload that varies in a *hurried rate*.

For this experiment and the next one, we must calculate the gradual variation, which requires the total times of scale-out and scale-in (see Section 4.2.2). We discover which are the scaling-out and scaling-in times when varying resource changes manually, and calculating their times at the end. A *scaling-out* takes ≈360 seconds, while a *scaling-in* takes ≈120 seconds. Considering reaction and scaling time values, the *total scale-out time* is 420 seconds, while the *total scale-in time* is 180 seconds.

Using the HR formulas from Section 4.2.2, the increasing hurried rate is $\approx 0{,}36$ $(150/420)$ OPS, and a decreasing hurried rate is $\approx 0{,}42$ $(75/180)$ OPS. In this experiment, we drive the CBS through 5 scale-out, then 5 scale-in, considering such rates.

For the *scale-out*, we must increase the workload for 2520 seconds, i. e., *(scale-outs* $+1$*)* $\cdot$ *total scale-out time*, where the $+1$ is the time to breach the first scale-out threshold. This results in a workload that goes from 0 to 907 OPS. For the *scale-in*, we must decrease the workload for 1080 seconds. We decrease the workload from 450 to 0 since 450 is the WI^{ry_sir} that breaches the scale-in threshold when AR is 6, i. e., amount of resource after 5 scale-out. Scale-out and scale-in sum up together ≈1 hour.

Figure 4.7 illustrates the behavior of the CBS case study and the cloud infrastructure during this experiment.

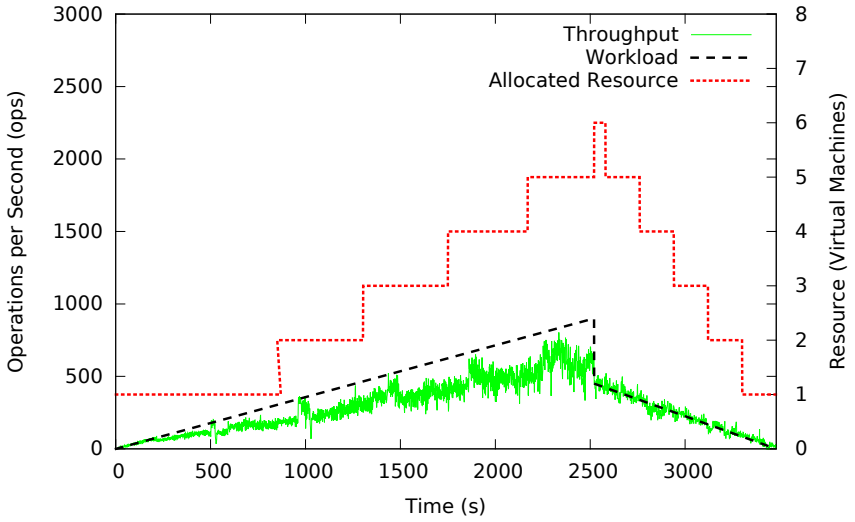


Figure 4.7 – Gradual Workload Variation in a *Hurried Rate*

In the figure, we see that most of the time the workload (dashed line) is higher than the throughput (solid line). This represents the CBS stress, which occurs due to resource under-provision, as we described in Section 4.2.2, confirming the assumption that requesting elasticity states by varying the workload gradually in a hurried rate stresses the CBS.

4.4.5 Gradual Workload Variation in a Lazy Rate

In this experiment, we drive the CBS case study by using a workload that varies gradually over time in a *lazy rate*. Using the LR formulas, the increasing lazy rate is $\approx 0{,}18$ $(150/2 \cdot 420)$ OPS, and a decreasing hurried rate is $\approx 0{,}21$ $(75/2 \cdot 180)$ OPS. We also drive the CBS through 5 scale-out, then 5 scale-in. For the *scale-out*, we must increase the workload for $5{,}040$ seconds, and decrease it for $2{,}160$ seconds, summing up 2 hours.

Figure 4.8 illustrates the CBS behavior in this experiment execution. In this figure, the workload and the throughput lines are close each other. Despite the throughput is not as steady as the workload, this is not a problem. Indeed, this reflects RUBBoS benchmark *thinking time*, which delays some requests to mimic real users behavior. We see that the execution time of this experiment takes the double of time compared to the hurried rate experiment.

This experiment results confirm the second assumption about gradual workload variation: requesting elasticity states by varying the workload gradually in a lazy rate takes the double of time than in a hurried rate. However, it does not stress the CBS.

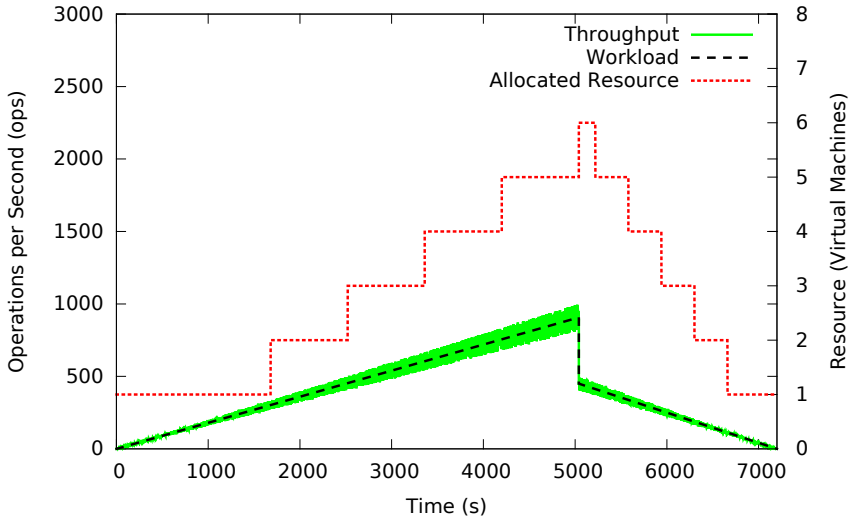


Figure 4.8 – Gradual Workload Variation in a *Lazy Rate*

4.4.6 Experiment with the Elasticity Driver

In this experiment, we use the elasticity driver to drive the CBS case study throughout the 5 scale-out and 5 scale-in. This is the sequence of elasticity states:

$$RES = \{s_1^{ry\_sor}, s_2^{so}, s_3^{ry\_sor}, s_4^{so}, s_5^{ry\_sor}, s_6^{so}, s_7^{ry\_sor}, s_8^{so}, s_9^{ry\_sor}, s_{10}^{so}, s_{11}^{ry\_sir}, s_{12}^{si}, s_{13}^{ry\_sir}, s_{14}^{si}, s_{15}^{ry\_sir}, s_{16}^{si}, s_{17}^{ry\_sir}, s_{18}^{si}, s_{19}^{ry\_sir}, s_{20}^{si}, s_{21}^{ry\_s}\} \quad (4.8)$$

Table 4.5 lists the elasticity states and the calculated workload intensities, where we consider ι value as $10\,\%$ ($\iota = 0{,}1$).

Figure 4.9 illustrates these experiment results. The elasticity driver prototype successfully drives the CBS case study throughout the RES. Workload and throughput lines are close each other all along the experiment, which shows that the prototype does not stress the CBS at any time. This experiment executes within the time of the first experiment (*hurried rate*). Therefore, the elasticity driver drives the CBS in a minimal time, without stress.

4.5 Conclusion

The results of the experiments confirm that a gradual variation of workload either stresses the CBS or prolongs the CBS driving. Indeed, varying the workload in a *hurried rate* stresses the CBS, while varying the workload in a *low rate* prolongs the time the CBS remains in each elasticity state. The elasticity driver drives the CBS throughout

RES State	Elasticity State/Substate	Resource Amount	Workload Intensity
s_1	ry_sor	1	165
s_2	so	1	165
s_3	ry_sor	2	330
s_4	so	2	330
s_5	ry_sor	3	495
s_6	so	3	495
s_7	ry_sor	4	660
s_8	so	4	660
s_9	ry_sor	5	825
s_{10}	so	5	825
s_{11}	ry_sir	6	450
s_{12}	si	5	375
s_{13}	ry_sir	5	375
s_{14}	si	4	300
s_{15}	ry_sir	4	300
s_{16}	si	3	225
s_{17}	ry_sir	3	225
s_{18}	si	2	150
s_{19}	ry_sir	2	150
s_{20}	si	1	75
s_{21}	ry_s	1	75

Table 4.5 – Workload Intensities Values for the RES

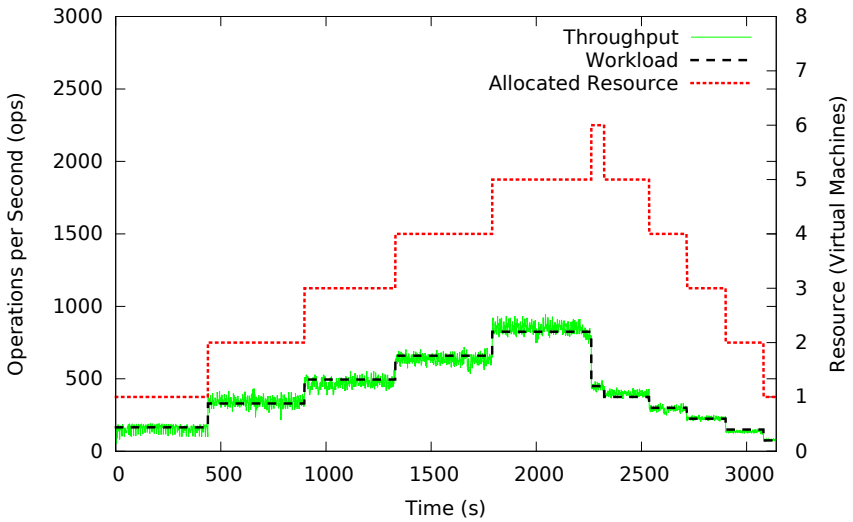


Figure 4.9 – Driving Using the Elasticity Driver

elasticity within the same time of using a hurried and gradually increased workload, but without stressing the CBS. This part of the work is published at [22].

The elasticity driver is the first step on driving CBS throughout elasticity. So far, it considers the CBS scalability as linear, which may not be true depending on the CBS or the amount of allocated resources. Therefore, as a first future work we plan to propose a strategy that considers non-linear scalability. A second improvement is to profile more generic workloads, such as workloads based on log traces. Finally, we plan to adapt the elasticity driver to address predictive elasticity policies since so far it only works with reactive ones.

5

Elasticity State-Based Testing

Driving the CBS through different elasticity states triggers elasticity state-related adaptations, which only occur during specific states. For instance, a component registration only occurs during the scaling-out state, before CBS stabilizing itself. In the literature, most of the testing approaches only test CBS during the *ready* state, when CBS is already stable. However, even if CBS recovers when it is back to the ready state, we must test it through the scaling periods must they can still affect customers experience. Therefore, CBS testing should consider all the elasticity states.

Coutinho et al. [31] proposes an approach that considers multiple CBS states. However, in their approach, testers must run the same test throughout CBS execution, and identify the CBS states after the execution. In such scenario, the test of a component registration, for instance, would execute through all the states unnecessarily.

This chapter proposes an approach to test CBS that switches among test cases based on their association to elasticity states. This approach identifies elasticity state occurrences at real-time, and switches among tests accordingly.

This chapter has three sections: Section 5.1 presents the elasticity state-based testing approach, Section 5.2 describes a prototype that implements the approach,and Section 5.3 describes some experiments and discusses their results. Finally, Section 5.4 concludes, and gives directions of future work.

5.1 Methodology for Elasticity State-Based Testing

This section presents the elasticity state-based testing approach. This approach focus on associating tests with specific elasticity states (Section 5.1.1) and synchronizing tests

execution accordingly (Section 5.1.2).

5.1.1 Test Methods Setup

To associate a test with an elasticity state, testers must set up some predefined parameters: *priority*, *delay*, and *repeat*. *Priority* is the sequence in which the test executes, in case of multiple test methods for the same state. Test methods with the same priority execute in parallel. *Delay* is the period of time before the test execution, which allows specific adaptations. *Repeat* is a boolean parameter, which configures whether or not the test must execute repeatedly during the elasticity state. If it is *true*, test execution repeats as long as CBS is in the elasticity state.

Table 5.1 shows three test examples: *t1*, *t2*, and *t3*. Test *t1* executes during all the elasticity states ($*$), *t2* during *scaling-out* and *scaling-in*, and *t3* during *scaling-out*. Since both, *t2* and *t3*, execute during the *scaling-out* state and have the same *priority*, they execute in parallel, whereas *t3* has a *delay* of 10 seconds and does not repeat.

Test	Elasticity State	Priority	Delay	Repeat
t1	*	1	0	*true*
t2	*scaling-out*, *scaling-in*	2	0	*true*
t3	*scaling-out*	2	10	*false*

Table 5.1 – Examples of Test Parameters

5.1.2 Test Synchronization

Algorithm 2 describes the test synchronization. This algorithm receives as an input a set of *tests* ($Tests$) ordered by the priority parameter, and the RES. Testers can execute the tests in two manners: using the elasticity driver (Section 4.3), or in parallel to alternative approaches. For instance, testers could expose a CBS to a random workload rather than driving it with the elasticity driver. If this algorithm executes apart from the elasticity driver, RES value is null, and the algorithm starts an elasticity state monitor. Otherwise, the algorithm uses the elasticity driver elasticity state monitor.

The monitor updates the variable $eState$ periodically , which keeps information of the current elasticity state. Then, the test synchronization executes while CBS is running (mon,$cbsIsRunning$) and RES is incomplete or unset ($i < sts \mid\mid sts == null$). At each iteration, the algorithm executes all the tests ($test \subset Tests$) related to the current elasticity state (cs_i), where each test execution respects the test parameters. Since the approach only focus on synchronizing test executions, the algorithm requires the tester

to write the logic necessary to assign a verdict to the test. Indeed, each test must return a boolean assertion as test verdict, which expresses whether the test execution passed or failed. Then, the algorithm stores each test verdict into a matrix V.

Algorithm 2: Test Synchronization

```
   Data: Tests Tests, Required Elasticity States RES
 1 i ← 0;
 2 if RES == null then
 3 |   mon ← monitor();
 4 else
 5 |   sts ← RES,size();
 6 end
 7 cs_i ← null;
 8 while mon,cbsIsRunning && (i < sts || sts == null) do
 9 |   if cs_i <> mon,eState then
10 |   |   killAllTests();
11 |   |   i++;
12 |   |   cs_i ← mon,eState;
13 |   |   foreach test ⊂ Tests do
14 |   |   |   if cs_i ⊂ test,eStates then
15 |   |   |   |   if test,delay > 0 then
16 |   |   |   |   |   wait(test,delay);
17 |   |   |   |   else
18 |   |   |   |   |   if test,repeat then
19 |   |   |   |   |   |   V_{cs_i}^{test} ← repeat(test);
20 |   |   |   |   |   else
21 |   |   |   |   |   |   V_{cs_i}^{test} ← test,run();
22 |   |   |   |   |   end
23 |   |   |   |   end
24 |   |   |   end
25 |   |   end
26 |   end
27 end
```

Tests configure to repeat have multiple verdicts, one for each execution, which requires an overall verdict calculation for each of them. Algorithm 3 calculates the overall verdict for such tests. It receives as input all the test verdicts ($v \subset V^{test}$), and an accuracy index (ai). Then, it verifies whether the percentage of *pass* verdicts is higher than or equal to ai, when the overall verdict is *pass*. Otherwise, it is *fail*. Indeed, among test verdicts there can be false positives. For instance, some conditions, such as

concurrent software executions or bandwidth issues, may disturb the CBS performance. Therefore, the algorithm uses an ai parameter, which expresses a tolerated margin of fail verdicts.

Algorithm 3: Test Oracle

Data: Test Verdicts V, and Accuracy Index ai

if $(|\{v \subset V^{test} : v = pass\}|/|V^{test}| * 100) \geq ai$ **then**
  return $pass$;
else
  return $fail$;
end

5.2 Prototype for Elasticity State-Based Testing

This section describes a prototype that implements the approach for elasticity state-based testing. Section 5.2.1 describes a way to write the tests, while Section 5.2.2 describes a way to read and execute the tests.

5.2.1 Writing Tests

We propose to write tests in Java programming language, which is popular, cross-platform, and has a rich standard library. A Java class, which we call *test file*, contains all the tests, where annotated Java methods ($@StateBasedTest$) implement the tests. This annotation supports the parameters described in Table 5.1: *elasticity state*, *priority*, *delay* and *repeated*.

Listing 1 presents a simple example of test file for the three tests illustrated in Table 5.1. In this example, tests execute naive math, where *t1* sums 2 and 2, *t2* multiplies 2 by 2, and *t2* divides 2 by 2. Each tests, as an assertion, verifies whether the calculated value is equal to the expected one, and returns the boolean value of such assertions.

Listing 1 – Example of Test File

```
public class Tests {
@StateBasedTest(state="*", priority = 1, delay = 0, repeat = true)
public boolean t1() {
 Integer sum = 2 + 2;
 return sum.equals(4);
}
@StateBasedTest(state = "scaling-out,scaling-in", priority = 2, delay
    ↪ = 0, repeat = true)
 public boolean t2() {
 Integer mult = 2 * 2;
 return mult.equals(4);
```

```
}

@StateBasedTest(state = "scaling-out", priority = 2, delay = 10,
    ↪ repeat = false)
public boolean t3() {
  Integer div = 2 / 2;
  return div.equals(1);
}
}
```

5.2.2 Executing Tests

We write the test synchronization algorithm in Java, where the the elasticity driver monitor identifies the elasticity states. This algorithm uses Java Reflection to read the *test file* and the test methods. It is a tester responsibility to write the code that connects to CBS and test it.

5.3 Experiments

This section discusses the two experiments where the elasticity state-based testing approach assists us in identifying performance degradations and their causes. Section 5.3.1 presents the web application case study, and the motivation for the two experiments. Section 5.3.2 describes the first experiments, which aims at testing performance degradations per elasticity state. Section 5.3.3 describes the second experiment, which aims at testing Web servers availability after a resource scale-out. Finally, Section **??** discusses the experiments results.

5.3.1 Case Study and Motivation

The CBS case study are native Apache HTTP servers deployed in a distributed manner on two different cloud providers: Amazon EC2 and Google CP. Each server is deployed on a small VM, *t1.small* on Amazon EC2, and *g1.small* on Google CP. Cloud provider elasticity controllers (de)allocate HTTP servers on-demand, and the cloud provider native load balancer balances the incoming requests among the servers. The elasticity driver drives the CBS throughout the following RES, which corresponds to two resource scale-out:

$$RES = \{s_1^{ry\_sor}, s_2^{so}, s_3^{ry\_sor}, s_4^{so}, s_5^{ry}\} \tag{5.1}$$

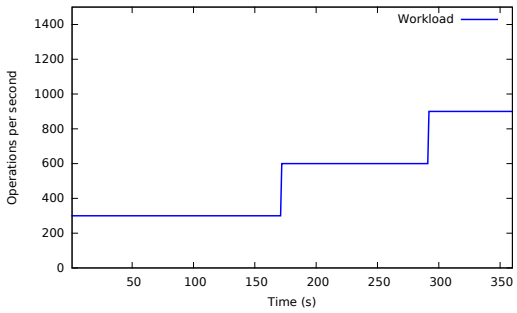
As workload generator, we use the *httperf* [60].

Elasticity State/Substate	Resource Amount	Workload Intensity (Amazon EC2)	Workload Intensity (Google CP)
ry_sor	1	320	300
so	1	320	300
ry_sor	2	640	600
so	2	640	600
ry	3	960	900

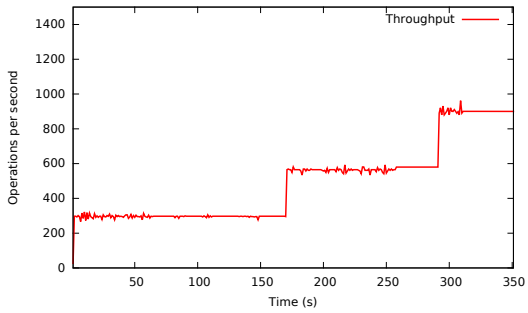
Table 5.2 – Workload Intensities

Table 5.2 shows the workload intensities calculated by the elasticity driver for both cloud providers.

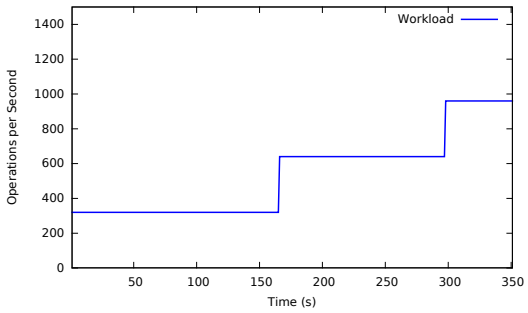
Figures 5.1a and 5.1c depict the workloads generated by the elasticity driver to drive CBS on Amazon EC2 and Google CP. Figure 5.1b depicts the throughput, on Amazon EC2, while Figure 5.1d depicts the throughput on Google CP. On Amazon EC2, the workload and throughput variations are close to each other, which indicates that the performance is as expected. However, on Google CP, the CBS has several throughput drops (throughput lower than the workload), which indicate performance degradations.



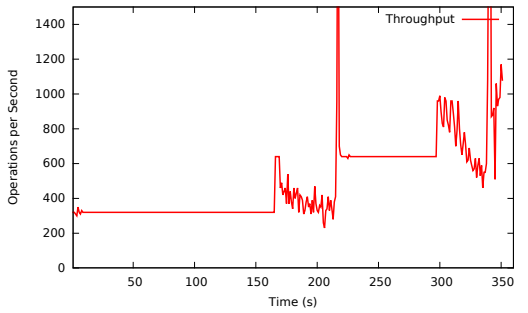
(a) Workload for Amazon EC2



(b) Throughput on Amazon EC2



(c) Workload for Amazon EC2



(d) Throughput on Google CP

Figure 5.1 – Workload and Throughput During the Case Study Driving on Google CP

The experiment described in Section 5.3.2 aims at discovering when the degradations occur. A further experiment (Section 5.3.3) checks the cause of these degradations.

We execute both experiments on Google CP, the cloud provider where CBS faces performance degradations.

5.3.2 First Experiments: Performance Testing per Elasticity State

This experiment aims at identifying which are the elasticity states where the performance degradations occur. This experiment uses the CBS driving described in the previous section, and in parallel, the approach for elasticity state-based testing syhchronizes a performance test (*testPerformance*). Table 5.3 shows the setup of this test, which must execute repeatedly in every elasticity states.

Test	Elasticity State	Priority	Delay	Repeat
testPerformance	*	1	0	*true*

Table 5.3 – Test Parameterization for Experiment 1

Listing 2 shows the test file for this experiment, which contains the test *testPerformance*. This test algorithm calls the external procedure *getPerformance* each 10 seconds, which reads the workload and throughput from workload generation logs. Then, the algorithm asserts the performance by comparing both values. For such comparison, the algorithm uses a relax index, which configures the limit of divergence between the workload and the throughput (*performanceLimit*), i. e., performance degradation. In the algorithm, this index is 20 % (0,2). Then, if throughput is higher than or equal to *performanceLimit*, there is no performance degradation (assertion returns true, which means the verdict is pass). On the contrary, there is performance degradation (assertion returns false, which means the verdict is fail).

Listing 2 – Test File for Experiment 1

```
public class Tests {
@StateBasedTest(state="*", priority = 1, delay = 0, repeat = true)
public boolean testPerformance() throws InterruptedException {
 // Read the Current Performance
 Map<Integer,Integer> performance = getPerformance();
 Set<Integer> keys = performance.keySet();
 Integer workload = keys.iterator().next();
 Integer throughput = performance.get(workload).intValue();
 // Relax Index
 Float relaxIndex = (float) 0.2;
 Float performanceLimit = (float) (workload * (1-relaxIndex));
 // Sleep for 10 seconds
 Thread.sleep(10000);
 // Assertion
 if (throughput >= performanceLimit) {
```

```
    return true;
  } else {
    return false;
  }
}
}
```

Figure 5.2 depicts the total of pass and fail verdicts for the two states through which the CBS is driven: *ready* and *scaling-out*. The 8 fail verdicts are all in the ready state, which means that it is the state where the performance degradations occur. Looking back to the RES used to drive the CBS, we find three ready states: one at beginning of the sequence, and two that follow scaling-out elasticity states. In Figure 5.1d, there are two periods of performance degradations, which coincide with the ready states that follow scaling-out states. Therefore, a plausible cause of performance degradations is the Google CP load balancer, which may be sending requests to the new Web server (added during the scaling-out state) before it is ready, i. e., HTTP service is up. This is what the next experiment investigates.

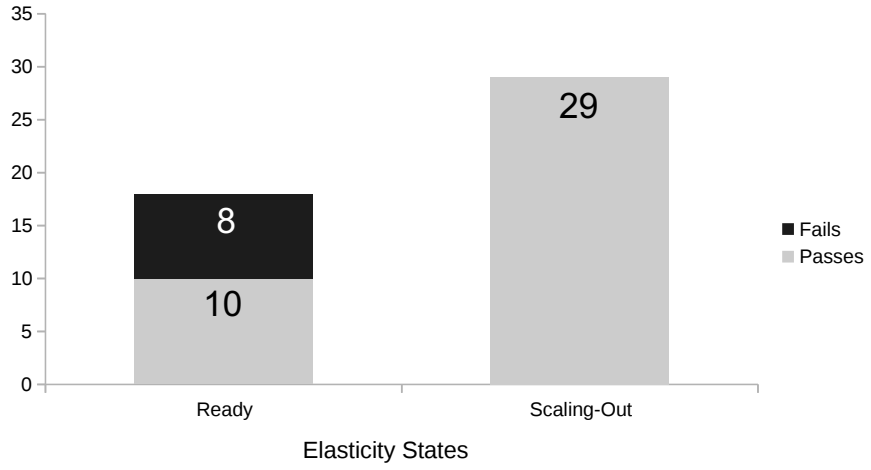


Figure 5.2 – Test Verdicts for the First Experiment

5.3.3 Second Experiment: Testing Web Servers Availability

This experiment aims at checking the assumption that new Web server receives requests before being ready. This experiment also repeats the case study driving, and execute a test (*testServers*) that tests whether the last allocated Web server is ready.

Table 5.4 shows the parameterization for this test. Since the first experiment reveals that the performance degradations are only in the ready state, we configure this test to

execute in this state. It is also configured with priority 1 since it is the only test, not to be delayed or repeated.

Test	Elasticity State	Priority	Delay	Repeat
testServers	*ready*	1	0	*true*

Table 5.4 – Test Parameterization for Experiment 2

Listing 3 shows the test file for this experiment, which contains the test *testServers*. This test algorithm calls an external procedure that at each 10 seconds (same interval as in the first experiment) gets the host address of the last allocated Web server. Then, it tries to connect to this server, and if the server is unavailable, the assertion returns fail. Otherwise, the assertion is pass.

Listing 3 – Test File for Experiment 2

```
public class Tests {
@StateBasedTest(state="ready", priority = 1, delay = 0, repeat = true
    ↪ )
public boolean testServers() {
 boolean allListenning = true;
 Integer port = 80;
 ArrayList<String> servers = getServers();
 for (String host : servers) {
 Socket s = null;
  try {
   s = new Socket(host, port);
  } catch (Exception e) {
   allListenning = false;
  }
 }
 Thread.sleep(10000);
 return allListenning;
}
}
```

Figure 5.3 illustrates the verdicts of the test execution, where there are 10 pass verdicts, and 8 fail verdicts. The verdicts match to the ones assigned in the first experiment during the ready state. This is an evidence that performance drops occur due to a load balancer issue, where the new Web server added during the scaling-out state receives requests before its HTTP service is running.

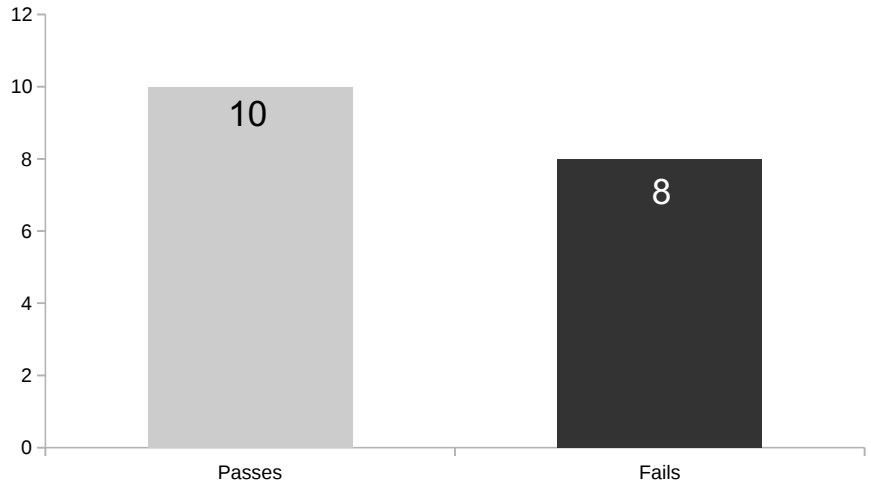


Figure 5.3 – Test Verdicts for the First Experiment

5.4 Conclusion

Thank to the approach for elasticity state-based testing the experiments reveal the causes of performance degradations on Google CP. In the first experiment, the proposed approach assists the elasticity states identification where performance degradations occur. In the second experiment, it synchronizes the Web server test to execute during the ready state, which reduces test execution efforts.

The elasticity state-based testing approach reduces the efforts in executing elasticity tests. However, we can still reduce this effort by improving the monitoring process to consider more specific CBS states. For instance, this approach idea is to test CBS adaptations. Thus, if we improve the approach to identify CBS adaptations rather than the current elasticity states, the test would be more specific, which would reduce the test execution efforts.

6

Elasticity Test Reproduction

During software development, regression tests should be executed regularly [36] to detect, diagnose, and correct bugs. Each execution must reproduce the same behavior, which requires test design and execution to be deterministic. For CBS elasticity testing, Chapter 4 already discusses the CBS driving importance. However, for some tests this may not be enough, since they may require to combine elasticity states along with further conditions.

By analysing the bug tracking of two popular CBSs, i. e., MongoDB and ZooKeeper, we discover some requirements for reproducing elasticity tests. One case is for the MongoDB NoSQL database bug 7974 [8]. For testing this bug, the test must manage to repeat the following elastic behavior: create a MongoDB replica set [11] with three nodes (three resource scale-out), remove one of the MongoDB node (one resource scale-in), and add a new MongoDB node (one resource scale-out). The test must also manage to repeat two time-based events: 1) to create a unique index before one of the MongoDB nodes is removed (the resource scale-in), and 2) upload a document after a new node is added (before the last resource scale-out). Another case is for the Apache ZooKeeper bug 2164 [13], which only occurs when the leader component leaves a ZooKeeper cluster. For testing this bug, the test must manage to repeat the CBS components variation in a deterministic manner, which consists in managing to repeat the deallocation of the leader component.

This chapter presents a prototype that tackles the discovered requirements for elasticity test reproduction: *the repetition of an elastic behavior*, *the repetition of time-based events*, and *the repetition of CBS components variation*. The approach also accelerates the test reproduction by anticipating the reaction to resource demand. Indeed, driving

CBS is time consuming since elastic controllers take a while (at least 60 s) to react to a resource demand. Since a test may be re-executed for multiple times, this implies in high cost.

This chapter is organized as follows: Section 6.1 describes the requirements for elasticity test reproduction. Section 6.2 presents a prototype that aims at tackling such requirements. Finally, Section 6.3 presents the experiments that validate the prototype.

6.1 Requirements for Elasticity Test Reproduction

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

To discover which are the conditions that CBSs face, we analyse elasticity-related bugs reported in the bug tracking of two popular CBSs: ZooKeeper, and MongoDB. Bug reports have rich information since developers use it to reproduce bugs. Therefore, this information reveals 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 bugs 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 behavior, such as the ones that restart a VM rather than remove or add one.

The two CBSs projects use *JIRA* [1] issue tracker to report their bugs. Therefore, for the Step 1, we use the query in Listing 4 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 that 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 4 – 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")
```

1. https://jira.atlassian.com

Table 6.1 lists the number of bugs selected at each search step. MongoDB has 25,780 bugs reported on its bug tracker system, where we find 316 in the first step, and 43 in the second step. ZooKeeper has 2,677 bugs reported, where we find 188 bugs in the first step, and 9 in the second step.

	Total of Bugs	**Bugs in Step** 1	**Bugs in Step** 2
MongoDB	25,780	316	43
ZooKeeper	2,677	188	9

Table 6.1 – Selected Bugs in the Systematic Search

The selected bugs reveal three main requirements for reproducing elasticity-related bug, which we also consider for elasticity tests reproduction: *elasticity control*, *selective elasticity*, and *event scheduling*. Despite *speediness* is not a requirement, it is an exigency to reduce the elasticity test reproduction cost.

— *Elasticity Control* is the ability to reproduce a specific elastic behavior. All the selected Elasticity-related bugs occur after a specific sequence of resource allocations and deallocations.

— *Selective Elasticity* is the need to manage a specific resource deallocation. For instance, deallocating a resource associated to the master component of a cloud-based system.

— *Event Scheduling* is the necessity to synchronize further events that may happen in parallel to elasticity changes. An event is any interaction with or stimulus to CBS, such as forcing a data increment or to simulate infrastructure failures.

— *Speediness* is the ability of reproducing a CBS test as fast as possible. Repeating an elastic behavior consists in repeating the workload generation, where we already know which are the workload and the predicted resource usage. Thus, to make the elastic behavior repetition faster, the approach anticipates the resource changes.

Table 6.2 shows the quantity of requirements for each CBS bug reproduction. As said previously, all the selected bugs require elasticity control, where 13 MongoDB (30 %) and 3 ZooKeeper (33 %) do not require the other requirements. Out of MongoDB bugs, 30 bugs (70 %) also need further requirements, within which 6 (14 %) bugs need all the requirements. Out of ZooKeeper bugs, 6 bugs (66 %) need further requirements, and 5 (55 %) of them need all the requirements.

6.2 Prototype for Elasticity Test Reproduction

This section describes the prototype for elasticity test reproduction, which meets all the requirements from previous section. The prototype only focus on repeating

	Elasticity Control	Selective Elasticity	Event Scheduling	All	Only Elasticity Control
MongoDB	*43*	*19*	*17*	*6*	*13*
ZooKeeper	*9*	*4*	*3*	*5*	*3*

Table 6.2 – Requirements for Bug Reproduction

test executions, whereas for testing purposes one could use the approach presented in Section 5.

6.2.1 Overall Architecture

Figure 6.1 depicts the overall architecture of the prototype. This architecture has four components: Elasticity Controller Mock (ECM), Workload Generator (WGen), *Event Scheduler* (ES), and *Cloud Monitor* (CM).

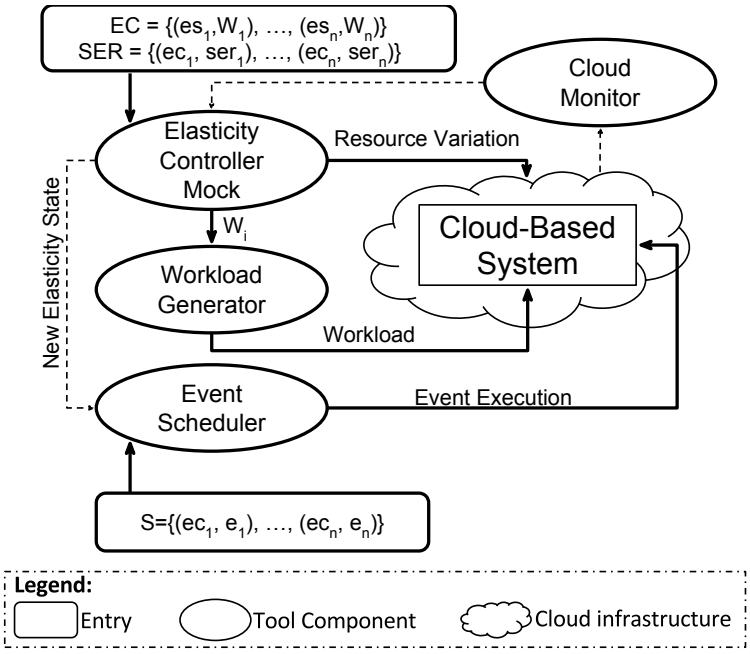


Figure 6.1 – Overall Architecture

Table 6.3 summarizes the requirements ensured by each component, which we further explain in the sequence.

Component	Elasticity Control	Selective Elasticity	Event Scheduling	Speediness
ECM	*Yes*	*Yes*	*No*	*Yes*
WG	*Yes*	*No*	*No*	*No*
ES	*No*	*No*	*Yes*	*No*
CM	*Yes*	*No*	*No*	*No*

Table 6.3 – Requirements Ensured by the Components

Elasticity Controller Mock (ECM)

The role of the ECM component is to provide *elasticity control* and *selective elasticity*. It simulates the behavior of the cloud provider elasticity controller by allocating and deallocating given resources.

The ECM receives two inputs, a set of Elasticity Changes (EC) and a set of Selective Elasticity Requests (SER). Each $ec \subset EC$ is a pair composed of an elasticity state (es) and a workload (W), where the workload consists of types of work and intensities (see Section 4.2.2). Each $ser \subset SER$ is a pair composed of an elasticity change (ec) and an *elasticity selection* (ese).

Listing 5 presents the EC settings in a Java property file.

Listing 5 – Example of Elasticity Changes Input

```
ec1=ready, (660,read)
ec2=scaling-out, (660,read)
ec3=ready, (540,read)
ec4=scaling-in, (270,read)
```

Listing 6 presents the SER settings in an annotated Java method.

Listing 6 – Example of Selective Elasticity Input File

```
@Selection{name="ese1", elasticity_change="ec4"}
public String select1() {
        ... //code to find a resource ID
        return resourceID;
}
```

For each $ec \subset EC$, before requesting the elasticity change, the ECM sends the workload parameters to the WGen component, and notifies the Event Scheduler (EvS) that a new elasticity state has began. Then, it starts the specified resource change by using cloud provider API to interact with cloud infrastructure directly. Transitions between ec are determined by the end of the elasticity state set in ec, which is informed by the Cloud Monitor (CM).

When a pair $ser \subset SER$ contains the current ec, the ECM executes the ese. The ese refers to an executable code that returns a resource identifier. For instance, this code could query the CBS to discover the identifier of the resource that hosts the CBS master component. Then, ECM deallocates that specific resource.

The ECM anticipates the elasticity changes without waiting for the elasticity controller to react to a resource demand. Indeed, elastic controllers, such as the ones from Amazon EC2 and Google CP, have a minimum reaction time. For instance, Amazon EC2 takes a minimum of $60\,\mathrm{s}$ to react to a resource demand. Since the approach generates the workload in a deterministic manner, we already know which resource should change, and anticipate this change.

Cloud Monitor (CM)

The only role of the CM is to *identify elasticity states*, keeping this information up to date on the ECM, helping in ensuring the *elasticity control*. The CM implements the monitoring component of the prototype introduced in Section 4.3, with a new functionality: sending an alert to the ECM when the current elasticity state ends.

Workload Generator (WGen)

The WGen is to generate the workload that drives the CBS throughout the RES, which also ensures the *elasticity control*. The WGen is based on the elasticity driver prototype introduced in Section 4.3. However, rather than receiving a sequence of workload intensities, it receives workload pairs $\langle type, intensity \rangle$ at a time. Then, the workload generator keeps generating the workload until a new pair arrives.

Event Scheduler (EvS)

Analogously to its name, the EvS ensures the *event scheduling* by synchronizing events with elasticity changes. When a new elasticity change begins, EvS receives an input S from the ECM, which contains pairs of an event (e) and an elasticity change (ec). Then, the event scheduler executes all the related events according to their parameters. The EvS implements the Algorithm 2 introduced in Section 5.1.2, from which we only suppress the repetition. This is because events have punctual interactions with the CBS, which occur once and at a specific time. Listing 7 illustrates the settings by using an annotated method in Java, where each event parameter is an element of the $@Event$ annotation.

Listing 7 – Example of Event Scheduler Input File

```
@Event{elasticity_change="ec1", priority="1", delay="0"}
public void event1() {
        ... // Any code that interacts to the CBS
}
```

6.2.2 Architecture Workflow

Figure 6.2 illustrates the prototype execution sequence. This execution starts by the CM component, which interacts with the Cloud ($Cloud$) 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 WGen, 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 timeframe before moving to the next ec. When a new elasticity state begins, the ECM sends a message to the EvS, which executes all the events related to this state. The prototype repeats this process until the last ec ends.

6.3 Experiments

This section presents three experiments that aim at validating the elasticity test reproduction approach. The experiments reproduce three representative elasticity-related bugs from two different CBS: MongoDB and ZooKeeper. The first bug is the *MongoDB-7974* from the MongoDB elasticity-related bugs described in Section 6.1. The other two bugs, *ZooKeeper-2164* and *ZooKeeper-2172* from the official bug tracking of ZooKeeper, another popular CBS.

We attempt to reproduce all the bugs in two ways: using the elasticity test reproduction approach, and manually (relying on the cloud computing infrastructure). Then, we compare both approaches results to verify whether they meet the requirements described in this chapter.

6.3.1 Experimental Environment

CBS Case Studies

MongoDB is a NoSQL document database. MongoDB has three different components: the configuration server, MongoS and MongoD. The configuration server stores metadata and configuration settings. While MongoS instances are query routers, which ensure load balance, MongoD instances store and process data.

Apache ZooKeeper [1] is a service for maintaining configuration information. A cluster of ZooKeeper nodes makes an *ensemble*, composed of a *leader* and *follower* nodes, where a distributed algorithm elects the leader. The leader works as a proxy, distributing the request among the *followers*. The *followers* keep a local copy of the configuration data to respond to requests.

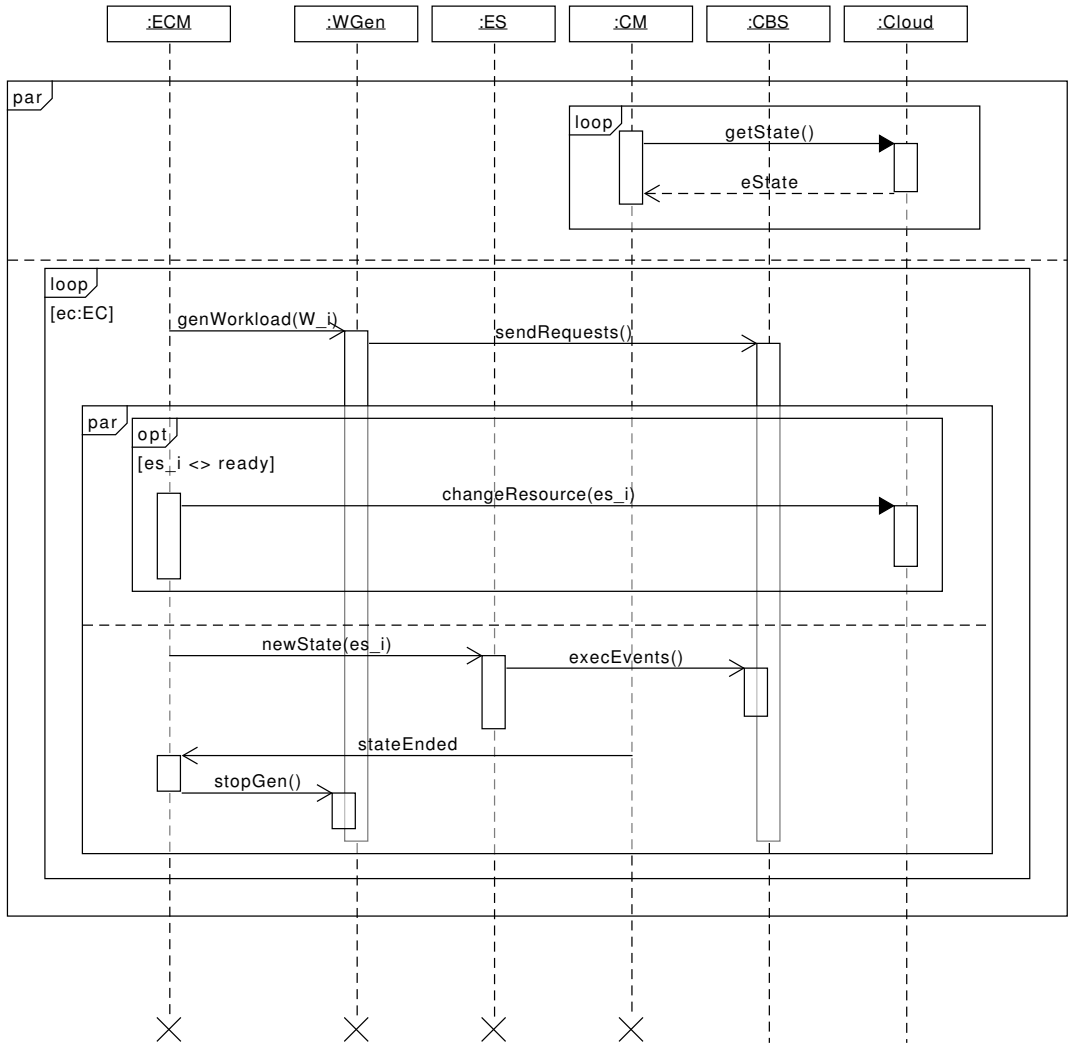


Figure 6.2 – Prototype Execution Sequence

Cloud Computing Infrastructure

We execute all the experiments on the cloud provider Amazon EC2, where the *scale-out* and *scale-in* thresholds are $60\,\%$ and $20\,\%$ of CPU usage.

In the experiment with MongoDB, the MongoS instance is deployed on a large machine (*m3.large*), while the other instances are deployed on medium machines (*m3.medium*). In the experiments with ZooKeeper, every node is deployed on a medium machine (*m3.medium*) [4].

Workload Tools

The Yahoo! Cloud Serving Benchmark (YCSB) [30] is the workload generation tool that generates the workload for the experiment with MongoDB. For the experiments with ZooKeeper, the workload generation tool is an open-source benchmark tool [50].

Selected Bugs: Requirements for Reproduction

Table 6.4 summarizes the requirements for the three selected bug reproductions. Those bugs cover all the possible combinations of requirements, constrained by the mandatory presence of *elasticity control*, and the need of at least one of the others requirements. We do not aim at reproducing any bug that only requires *elasticity control*, since one could reproduce the required elastic behavior using the elasticity driver (Section 4), and then, meets elasticity control requirement only.

	FEATURE		
BUG	**Elasticity Control**	**Selective Elasticity**	**Event Scheduling**
$MongoDB-7974$	*Yes*	*Yes*	*Yes*
$ZooKeeper-2164$	*Yes*	*Yes*	*No*
$ZooKeeper-2172$	*Yes*	*No*	*Yes*

Table 6.4 – Requirements for Reproducing the Three Selected Bugs

MongoDB bug 7974

This bug affects the MongoDB versions $2{,}2{,}0$ and $2{,}2{,}2$, when a secondary component leaves a MongoDB replica set [11]. Indeed, MongoDB elects one replica set component as primary member, which works as a coordinator, while the others remain as secondary members.

This bug reproduction requires 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, this bug reproduction needs to meet all the requirements considered in this chapter: *elasticity control*, *selective elasticity*, and *event scheduling*.

ZooKeeper bug 2164

This is a ZooKeeper bug that concerns the leader election. According to the bug report [13], in an ensemble with three nodes, when the node running the leader shuts down, a new leader election starts and never ends.

This bug reproduction must follow a precise sequence: initialization (allocation of the first node), followed by two nodes allocation and the deallocation of the leader node. The main difficulty of reproducing this bug is that for a three nodes ZooKeeper *ensemble*, the deallocated node is not necessarily the leader.

In conclusion, this bug reproduction needs to meet two requirements: *elasticity control* and *selective elasticity*.

ZooKeeper bug 2172

This is a ZooKeeper bug that introduces the dynamic reconfiguration. According to the bug report [14], when a third node enters a ZooKeeper *ensemble*, the system enters an unstable state and cannot recover.

The analysis of available logs identifies the bug occurs when a leader election starts right after a third node allocation. More precisely, when a new node joins the ensemble, its synchronizes the configuration data with the leader. Therefore, if the data synchronization does not end before the leader election, the bug occurs.

This bug reproduction requires a simple elastic behavior: initialization and two node allocations. However, this sequence alone does not reproduce the bug: the leader election must start before the end of the data synchronization process. Increasing the data amount through an event synchronized with the third node allocation can ensure this bug reproduction.

This bug reproduction needs to meet two requirements: *elasticity control* and *event scheduling*.

6.3.2 Bug Reproduction

This section describes the use of the elasticity test reproduction approach to reproduce the three bugs, and compare the results to the manual reproduction attempts. We do not explain in details the setup of manual reproductions. We assume that one can manage the *control elasticity* and meet this requirement. Indeed, reproducing elasticity is a native feature of cloud computing infrastructures, and we just drive CBS through required elastic behavior using the elasticity driver (Section 4).

MongoDB-7974 Bug Reproduction

To reproduce MongoDB bug 7974 using the elasticity test reproduction approach, we first manually create the MongoDB replica set, composed of three nodes. Then, we

configure 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 this bug reproduction requires to deallocate a secondary member of MongoDB replica set, an *elasticity selection* (ese_1) discovers a secondary member. Then, this ese is associated to the scaling-in state, i. e., elasticity change ec_2 ($\langle si_1, \langle 1500, r\rangle\rangle$). Listing 8 presents this ese algorithm, which first calls the external procedure $getAllMember$ that returns all the members of MongoDB replica set. Then, the algorithm uses MongoDB API to query the first member of the replica set for the replica set status ($getReplicaSetStates$). To discover a secondary member, the algorithm queries the replica set status whether each member is the master, and returns the first member that is not the masters, i. e., is a secondary member.

Listing 8 – Elasticity Selection that Discover MongoDB Secondary Members

```
@Selection{name="ese1", elasticity_change="ec2"}
public ServerAddress discoverMongoDBSecondary() {
ArrayList<ServerAddress> members = getAllMembers();
ReplicaSetStatus replSetStatus = new ReplicaSetStatus();
replSetStatus = members.get(1).getReplicaSetStatus();
for (ServerAddress member : members) {
  if (!replSetStatus.isMaster(member)) {
        return member;
  }
}
}
```

Two events implement the user interactions with the MongoDB during the test reproduction. Event $e1$ creates a unique index, while event $e2$ inserts 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 that deallocates a secondary MongoDB member. The $e2$ is associated to elasticity change ec_5, the last ready state.

Elasticity Change	Event ID	Execution Sequence	Wait Time
ec_3	$e1$	1	0 s
ec_5	$e2$	1	0 s

Table 6.5 – MongoDB-7974 Event Schedule

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

Table 6.6 shows the result of all the three executions, either using the elasticity test reproduction approach or manually. All the attempts using the approach reproduce the bug, while none of the manual attempts do it.

Reproduction	Reproduced	Not Reproduced
Elasticity Test Reproduction Approach	3	0
Manually	0	3

Table 6.6 – MongoDB-7974 Bug Reproduction Results

In the executions without the approach for elasticity test reproduction, we force MongoDB to elect the intermediate node (in the order of allocation) as primary member [3], what can occasionally occur in a real situation. In this scenario, independent of scale-in settings, cloud computing infrastructure always deallocate a secondary member, since Amazon EC2 only allows to deallocate the oldest or newest nodes. Even though cloud computing infrastructures may reproduce the required elastic behavior, this bug still needs the event executions. This is the reason why the execution without the elasticity test reproduction approach does not reproduce the bug .

ZooKeeper-2164 Bug Reproduction

To reproduce this bug, we translate and complete the scenario (Section 6.3.1) 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 ZooKeeper leader node, a ese is associated to the scaling-in state, i. e., elasticity change $e6$ ($\langle si_1, \langle 5000, r\rangle\rangle$). Listing 9 presents this ese. Since ZooKeeper API does not provide a method to query which is the ZooKeeper leader, the elasticity selection algorithm connects to each ZooKeeper node, and executes the ZooKeeper command $stat$. This command returns statistical information, which includes the node execution mode, i. e., leader or follower. Then, the algorithm returns the node that is the leader. We use the Java library *JCsh* [7] to implement the procedures that connect to the ZooKeeper nodes ($sshSession$) and execute remote commands ($sshCommand$).

Listing 9 – Elasticity Selection that Discover ZooKeeper Leader

```
@Selection{name="ese1", elasticity_change="ec6"}
public String discoverZooKeeperLeader() {
String user = 'atlanmodels';
```

```
String password = 'secret';
ArrayList<String> hosts = getAllHosts();
String leaderHost = null;
String outputStream = null;
for (String host : hosts) {
 Session session = sshSession(host,user,password);
 session.connect();
 outputStream = sshCommand('echo stat | nc localhost 2181 | grep Mode
    ↪ ');
 if (outputStream.equals('Mode: leader')) {
   leaderHost = host;
   session.disconnect();
   break;
 }
 session.disconnect();
}
return leaderHost;
}
```

The sequence of elasticity states, which includes a selective elasticity, should reproduce the bug. To verify whether the failure occurs, we implement a JUnit [43] test oracle. This test oracle executes after the last elasticity change ($\langle si_1, \langle 5000, r\rangle\rangle$), and repetitively searches for a leader until finding it or the timeout is over. In the first case, the verdict is *pass*, what means the approach reproduces the bug. Otherwise, the verdict is *fail*.

As well as in the first experiment, this experiment is executed in two ways: using the elasticity test reproduction approach, and manually. This experiment is repeated three times for each setup.

Since the selective elasticity is one of this bug reproduction requirements, when executing without the reproduction approach, we try to reproduce a real scenario, where a leader election can ellect any node as the leader. Therefore, we force the ZooKeeper to elect a different node as the leader at each execution: the newest, the oldest, then the intermediate node. Then, Amazon EC2 elastic controller deallocates 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 the first two executions we can configure Amazon EC2 to deallocate the leader, but not during the last one.

Table 6.7 summarizes the results. When using the reproduction approach, all the three test executions pass, which demonstrates the ability of the approach to reproduce the bug. In contrast, only two manual executions pass, the ones where the leader is the newest or the oldest node. Therefore, manual attempts to not reproduce the bug.

ZooKeeper-2172 Bug Reproduction

This bug reproduction (Section 6.3.1) requires the following sequence of elasticity changes:

Reproduction	Pass Verdicts	Fail Verdicts
Elasticity Test Reproduction Approach	3	0
Manually	2	1

Table 6.7 – ZooKeeper-2164 Bug Reproduction Results

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

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 $10\,\mathrm{s}$ according to the log files. The event $e1$ requests a data increasing to an amount that should take longer than $10\,\mathrm{s}$ to synchronize. Table 6.8 describes this event association with the elasticity state so_2. Since this experiment uses Amazon EC2 $m3{,}large$ machines, which have a bandwidth of $62{,}5\,\mathrm{MB/s}$, the data amount must be $\approx 625\,\mathrm{MB}$ of data.

Elasticity Change	Event ID	Execution Sequence	Wait Time
ec_4	$e1$	1	$0\ s$

Table 6.8 – ZooKeeper-2172 Event Schedule

We use the test oracle as for the bug 2164 by executing it during the last *ready* elasticity state which should identify ZooKeeper leader before a timeout. Table 6.9 summarizes the experiment execution. In all the three executions, the test verdict is *pass*, which means that the testing approach reproduces the bug. Since the Amazon EC2 cannot manage events synchronized with elasticity states natively, the three execution do not reproduce the bug.

Reproduction	Pass Verdicts	Fail Verdicts
Elasticity Test Reproduction Approach	3	0
Manually	0	3

Table 6.9 – ZooKeeper-2172 Bug Reproduction Results

6.4 Conclusion

In the experiments, we use the prototype proposed in this chapter to reproduce bugs of two popular CBSs. The prototype reproduces all the bugs, while the attempts without it cannot reproduce the bugs. We repeat each bug reproduction multiple times, where

without the proposed prototype each bug reproduction fails at least once. This indicates that the proposed prototype reproduces elasticity-related bug in a deterministic manner. This part of the work has been published at [23].

Since testing is not only about reproducing, but also diagnosing them, an evolution for the reproduction approach is to integrate it to the approach for test sequences generation. Then test configurations will cover the approach features, such as selective elasticity, and event scheduling. Finally, we plan to investigate how fast a test execution can be executed without compromising the CBS behavior.

7

A Domain-Specific Language for Elasticity Test Deployment, Configuration, and Execution

Managing CBS during elasticity testing is complex, laborious, and requires the tester to master cloud computing and its particularities.. For this management, testers must interact with cloud providers multiple times, requesting computing resources, deploying CBS components, and configuring system elasticity and test parameters. The laboriousness increases when testers need to execute a test over different cloud providers, since each provider has its own procedures and language.

Previously in Section 3.4, we presented some research efforts that propose high-level languages for CBS deployment, making this process less complex and laborious. However, none of them addresses the system elasticity configuration nor elasticity testing. To fill this gap, in this chapter, we propose a Domain-Specific Language (DSL) for deploying, configuring, and executing elasticity tests, which abstracts the complexities of elasticity testing by using a friendly syntax. Furthermore, programs expressed in this DSL are less verbose than other languages. Along with the DSL, we propose a way to translate DSL programs into cloud provider specific CLI calls. This translation allows a test execution over different cloud providers with no program change.

This chapter is divided as follows. Section 7.1 illustrates the main activities and concepts in elasticity test configuration. Section 7.2 proposes a DSL and the way it is translated into executable code. Section 7.3 presents a configuration case study, its translation into executable code, and discusses the results. Finally, Section 7.4 concludes,

and gives directions of future work.

7.1 Elasticity Test Deployment and Configuration

Managing a CBS during elasticity testing includes three main activities:

1. CBS component deployment: specification of the CBS under test, its dependencies, as well as its operating system and hardware requirements.
2. Elasticity parameter configuration: specification of different cloud provider parameters: thresholds, monitored resources, etc.
3. Testing parameter configuration: required system states, history of state transitions, association between system states and test cases, etc.

7.1.1 CBS Component Deployment Concepts

Figure 7.1 depicts the activities a tester must perform to test a CBS component. First, she must connect to the cloud provider. Then, she must select the resources required for the component deployment, (i. e., an Operating System (OS) image and a VM type). An image is a copy of a OS state, while a VM type describes the hardware configuration, such as CPU, storage, and memory. Testers must also grant external access to the CBS by configuring its ports on the Cloud firewall. Then, the tester launches an instance with the selected resources, deploys and configures the component on it.

Figure 7.2 presents a UML Class Diagram representing the deployment configuration of a CBS. In this diagram, a *Provider* (Cloud Provider) may provide several *Instances*, which implements an *Image*, a *MachineType*, and one or more *SoftwareComponents*. A *MachineType* has a name, a storage capacity, a quantity of CPU, and a memory capacity. An *Image* has an identifier and implements an *OS* (Operating System), which has a distribution, a version, and an architecture. A *SoftwareComponent* has a name and describes the manner to install it (i. e., replication or installation package). It has four *Source* files: a pre- and a post-installation scripts, an installation file, and a configuration file. A *Source* file has a location and a destination. A cloud *Provider* has at least one *SecurityGroup*, which groups one or more *PortConfigurations*. The last describes a port and a reachability.

While there are different manners to install software components, we consider so far that they are either installed by direct folder replication, or using a package manager. The source file associated to a software component allows its initial installation and configuration, as well as calling scripts before and after the component execution.

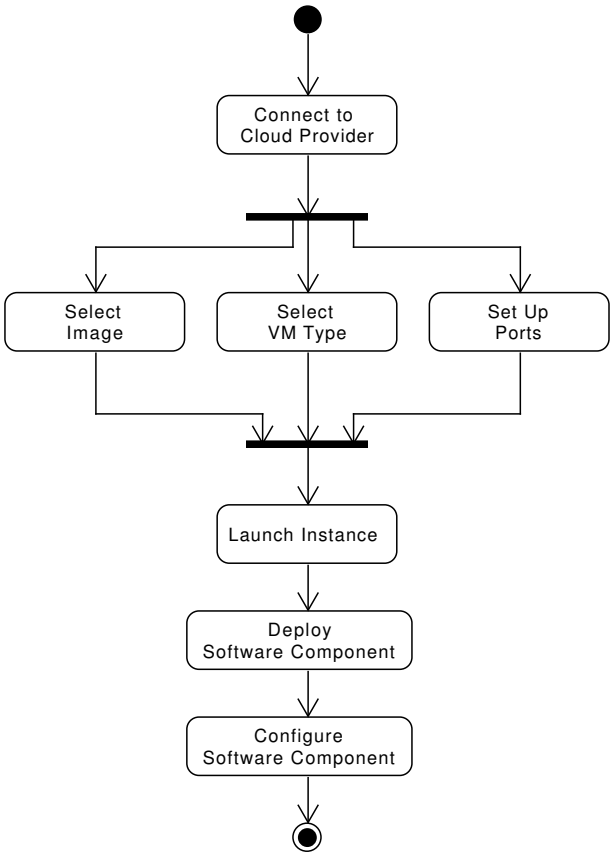


Figure 7.1 – CBS Deployment Activities

7.1.2 Elasticity Configuration Concepts

Enabling elasticity in a cloud system requires the configuration of different parameters. Figure 7.3 illustrates tester activities when enabling cloud system elasticity. After connecting to the cloud provider, the tester sets up a resource pool, where he/she describes the resource that should scale. Then, he/she sets up scale-out and scale-in thresholds, and the policies to scale the resource. Finally, the tester creates an auto-scaling group that implements such setups.

Figure 7.4 depicts a UML Class Diagram representing the concepts related to elasticity configuration. In this diagram, an elasticity *Policy* defines an action (i. e., add or remove a resource), the quantity of concerned resources, and the reaction time needed to apply the action, after a threshold breach. Each policy is associated to a pool, at least one scale-in threshold, and at least one scale-out Threshold. A *Threshold* corresponds

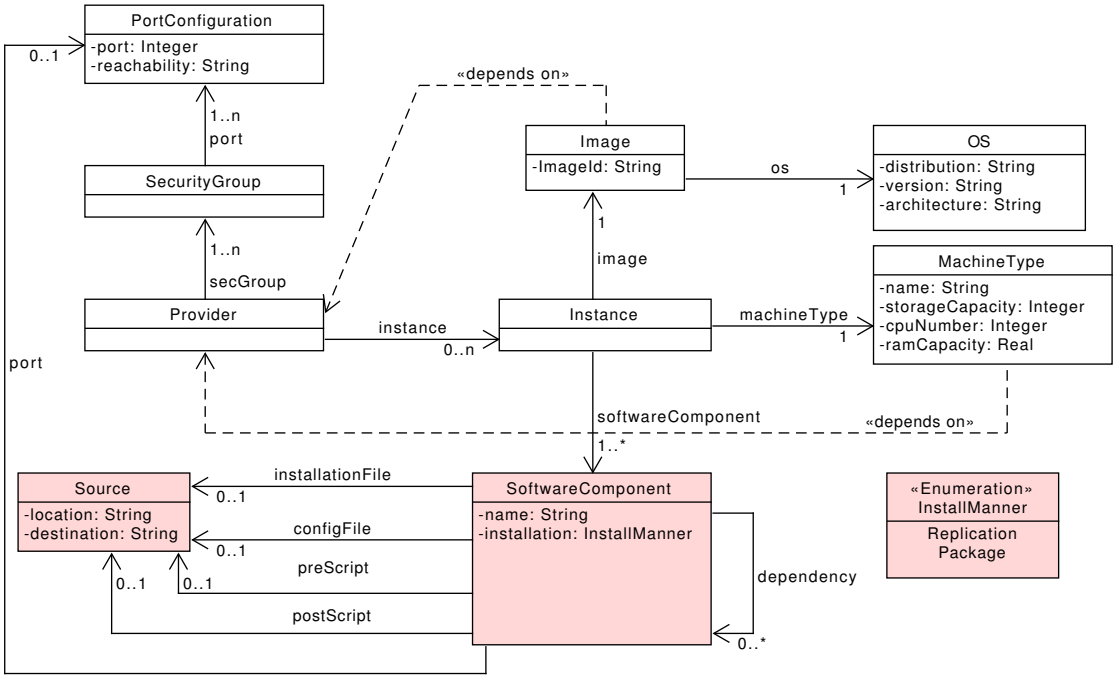


Figure 7.2 – CBS Deployment Setup Model

to a resource, the kind of measure (i. e., average, minimum, etc.), a comparator, and a reference value. A *ResourcePool* has a minimum and a maximum size and refers to a cloud resource, an *Instance* (see Figure 7.2).

7.1.3 Elasticity Test Configuration Concepts

The elasticity test configuration depends on the properties the tester wants to verify. In this section, we recapitulate the concepts related to the elasticity testing approaches described in the previous chapters.

Elasticity Driving Configuration

Figure 7.5 depicts the model of elasticity driving configuration, based on the approach described in Chapter 4. In this model, an elasticity *Driving* refers to two *SoftwareComponents* (see Figure 7.2). The first one is the component that receives the workload leading to the required elasticity behavior. The second one is the tool that generates the workload. A driving also defines an intensity when profiling the resource usage, and is associated to one or more *RequiredElasticityStates*, which has elasticity states, and the workload that leads to such states.

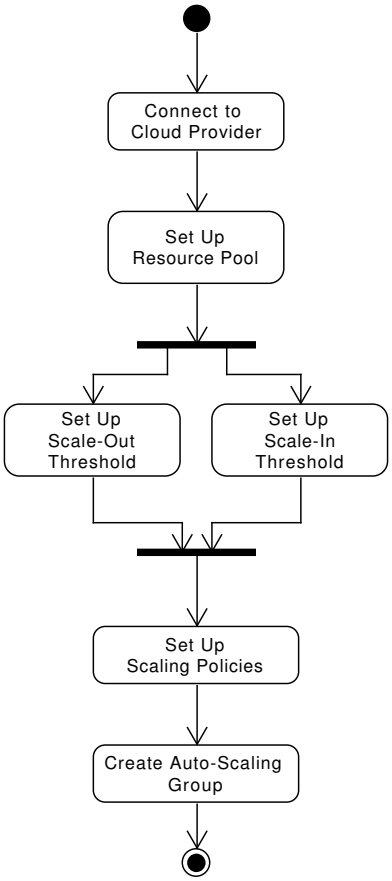


Figure 7.3 – Elasticity Configuration Activities

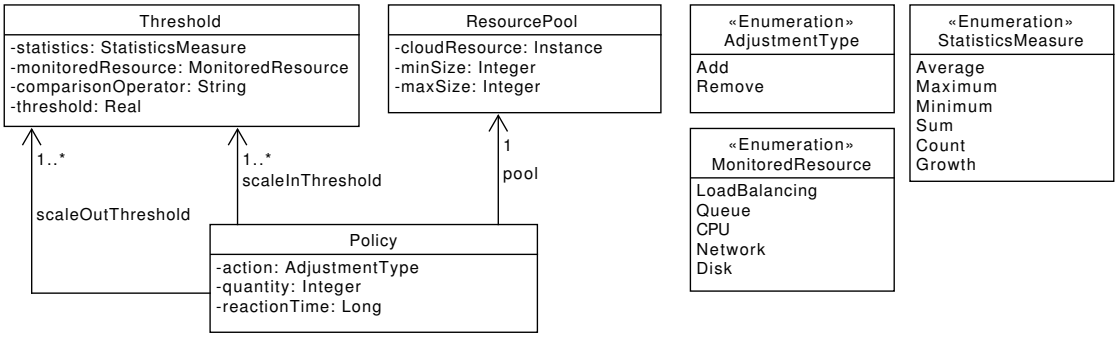


Figure 7.4 – Elasticity Setup Model

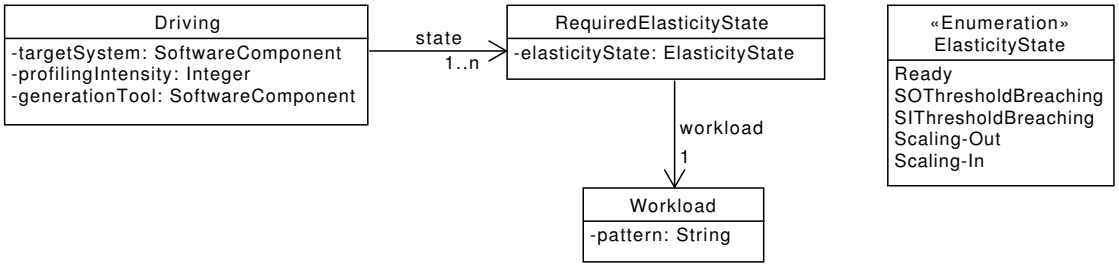


Figure 7.5 – Elasticity Driving Configuration Model

Elasticity State-Based Test Configuration

Figure 7.6 depicts the model of elasticity state-based test configuration, based on the approach described in Chapter 5. In this model, a *TestSchedule* has an *ElasticityState* and an execution priority. A test schedule refers to a *Test*, which has a name, a pre-execution delay, and a repetition choice.

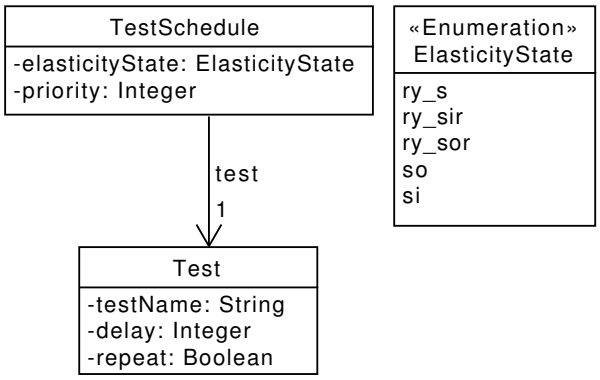


Figure 7.6 – Elasticity State-Based Configuration Model

Event Scheduling Configuration

Figure 7.7 depicts the event scheduling configuration model, based on the approach described in Chapter 6. In this model, an *EventSchedule* has an execution priority and references an *Event* and one or more *ElasticityChange*. Events has a name and a pre-execution delay. Elasticity changes use a unique sequence number to keep a historical trace of elasticity state changes.

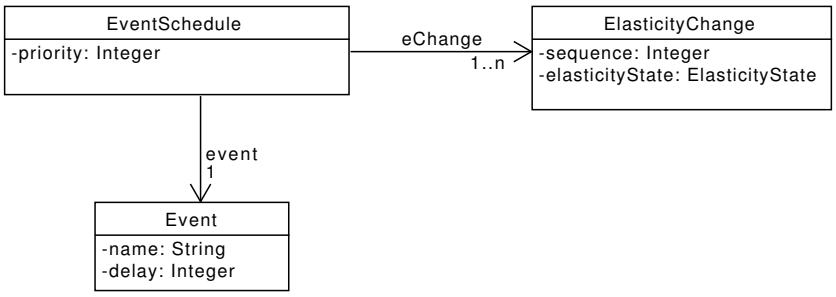


Figure 7.7 – Event Scheduling Setup Model

Selective Elasticity Configuration

Figure 7.8 presents the selective elasticity configuration model, which is based on the approach described in Chapter 6. In this model, a procedure for resource *Selection* has a name and is associated to one or more elasticity changes.

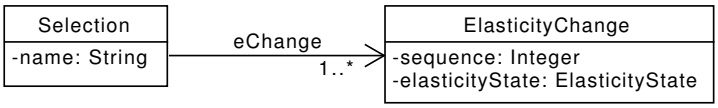


Figure 7.8 – Selective Elasticity Setup Model

7.2 A DSL for Elasticity Test Deployment and Execution

After presenting the main concepts of elasticity testing deployment and execution, we present in this section a DSL based on these concepts. Programs written in this language can configure and execute elasticity tests independently of the cloud provider. To make this independence possible, we base this approach on Thiery et al. work [72].

7.2.1 DSL to Configure Elasticity Testing

The DSL is divided into three parts, according to the tester actions when managing elasticity tests: deployment of CBS components (*Deployment*), elasticity configuration

(*Elasticity*), and elasticity testing configuration (*ElasticityTesting*). We also propose a fourth part (*Provider*), which is cloud provider-dependent, that lists available resources on a cloud provider.

In order to reduce the DSL syntax presentation, we propose a simple case study, which consists of the following activities:

— Deploy on a Cloud a distributed Web application, composed of native Apache HTTP servers and a HTTP proxy.

— Deploy the software components for elasticity driving, elasticity state-based testing, and workload generation.

— Configure elasticity configuration parameters on a cloud provider.

— Configure elasticity driving and elasticity state-based test configuration.

We use this case study to exemplify the DSL in the next sections.

Deployment Part

Deployment part is sub-divided into two sub-parts: *RequiredRes* that describes the required resource for each CBS component, and *CBSComponents* specifies the required CBS components. Looking back at Figure 7.2, *RequiredRes* corresponds to the parts in white, while *CBSComponents* corresponds to the parts in pink.

Despite *RequiredRes* refers to cloud provider resources, its configuration is cloud provider-independent. That is, the tester specifies the required resources regardless of their availability in the cloud provider. Listing 10 shows an example of *RequiredRes* configuration. As required resources, the tester describes machine types, OSs, and ports. We do not describe *Image* and *Instance* (see Figure 7.2) in this part. They concern the required resources implementation, whose configuration is generated in the compilation phase by matching the required resources to those available in the cloud provider.

Listing 10 – Example of *RequieredRes* Specification

```
RequiredRes {
MachineTypes {
 medium {
  storageCapacity = 3;
  cpuFrequency = 1.7;
  ramCapacity = 3;
 }
 large {
  storageCapacity = 20;
  cpuFrequency = 3.5;
  ramCapacity = 5;
 }
}
```

```
OSs {
 ubuntu32 {
  distribution = Ubuntu;
  version = 10;
  architecture = i386;
 }
}
Ports {
 http {
  port = 80;
  reachability = '0.0.0.0';
 }
}
}
...
```

Listing 11 shows an example of a *CBSComponents* setup. First, it describes the external files (*Sources*) necessary to install and configure the CBS components. Then, it describes their installations, where each installation associates a software component to the required resources and external files. More precisely, the HTTP proxy installation, a Web server, a workload generator, an elasticity driver, and state-based tests. Installations are analogous to *Instances* from the model presented in Figure 7.2. However, since in *CBSComponents* we only consider required resources, not existing ones, in the remainder of this thesis we refer to installations as *abstract instances*.

Listing 11 – Example of *CBSComponets* Specification

```
...
CBSComponents {
Sources {
 addServer {
  location = './add-server.sh';
 }
 removeServer {
  location = './remove-server.sh';
 }
 haproxy {
  location = './haproxy-1.7.3/';
  destination = '~/haproxy/';
 }
 haproxy_config {
   location = './haproxy.cfg';
   destination = '~/haproxy/';
 }
```

```
 elasticity_driver {
  location = './elasticity-driver-1.0.0/';
  destination = '~/elasticity-driver/';
 }
 state_based_testing {
  location = './state-based-testing-1.0.0/';
  destination = '~/state-based-testing/';
 }
}
install http_proxy on MachineType large {
 os = ubuntu32;
 installation = Replication;
 installationFile = haproxy;
 configFile = haproxy_config;
 port = http;
}
install web_server on MachineType medium {
 softwareComponent = 'apache2';
 os = ubuntu32;
 installation = Package;
 preScript = addServer;
 postScript = removeServer;
 dependency = haproxy;
}
install workload_generator on MachineType large {
 softwareComponent = 'httperf';
 os = ubuntu32;
}
install e_driver on MachineType large {
 os = ubuntu32;
 installation = Replication;
 installationFile = elasticity_driver;
}
install s_based on MachineType medium {
 os = ubuntu32;
 installation = Replication;
 installationFile = state_based_testing;
}
}
...
```

Elasticity Part

Listing 12 presents an example of an *Elasticity* configuration, where a resource pool contains from 1 to 10 web server abstract instances, from Listing 11. The example defines two threshold levels, the first for when the maximum CPU usage is higher than 60 % (*highCPU*), and the second for when the maximum CPU usage is lower than 30 % (*lowCPU*). It also defines two policies, the first one defines that the resource pool should expand when the first threshold is breached for more than 60 s and the second one defines that the resource pool should shrink when the second threshold is breached for more than 60 s.

Listing 12 – Example of *Elasticity* Specification

```
...
Elasticity {
ResourcePools {
 webserver_pool = web_server[1,10];
}
Thresholds {
 highCPU { Maximum CPU > 60 };
 lowCPU { Maximum CPU < 30 };
}
Policies {
 add_srv {if highCPU during 60 then Add 1 in webserver_pool };
 rm_srv {if lowCPU during 60 then Remove 1 from webserver_pool
    ↪ };
}
}
...
```

Elasticity Testing Part

We sub-divide the *ElasticityTesting* part into four sub-parts, each one addressing different aspects of elasticity tests: *CBS Driving* describes elasticity driving, *State Based Testing* describes the state-based tests, *Events Scheduling* describe event scheduling, and *Selective Elasticity* describes the selective elasticity.

In the case study, only the *CBS Driving* and the *State Based Testing* parts are necessary. Listing 13 shows a driving configuration, that configures the application *web_server* driving through the following sequence of states: *ry_s*, *ry_sor*, and *so*. The workload has a *read* pattern and a profiling intensity of 100, generated by the previously defined generation tool (*workload_generator*).

Listing 13 – Example of *CBSDriving* Specification

```
...
ElasticityTesting {
Driving {
 RequiredElasticityStates {
  res1 = {(ry_s, 'read'), (ry_sor, 'read'), (so, 'read')};
 }
 drive web_server through res1 with workload_generator {
    ↪ profilinfIntensity = 100 };
}
...
```

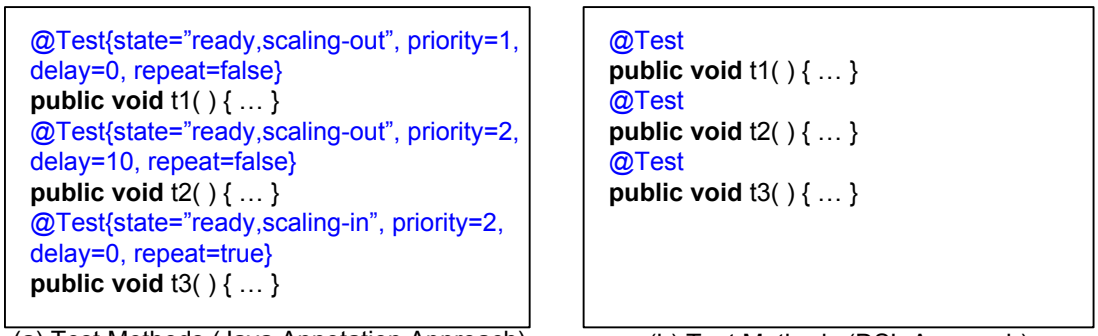
Listing 14 shows the *State Based Testing* configuration for three tests, *t1*, *t2* and *t3*, and associate then to the states *ready*, *scaling-out*, and *scaling-in*. In the execution configuration, a comma means that tests are in a sequence, while a pipeline means they are parallel. This expresses test priority, represented in the model of Figure 7.6, where tests executed in sequence have incremental priority, while tests executed in parallel have the same priority. For instance, through *ready* state, *t1* has priority 1, while *t2* and *t3* have priority 2.

Listing 14 – Example of *StateBasedTesting* Specification

```
...
StateBasedTesting {
 Tests {
  t1 {
   delay = 0;
   repeat = false;
  }
  t2 {
   delay = 10;
   repeat = false;
  }
  t3 {
   delay = 0;
   repeat = true;
  }
 }
 execute t1, t2 | t3 through ry_s;
 execute t1, t2 through so;
 execute t3 through si;
}
}
}
```

Using Java annotations to configure test cases, such as proposed in Section 5.2.1, is a straightforward solution. However, testers must write all the test specification using annotation parameters, which may be confusing due to the low level of abstraction. Furthermore, as previously explained, it is not possible to configure the priority according to elasticity state. With the proposed DSL, the test execution specification has a higher level of abstraction.

Figure 7.9 illustrates the difference between both specifications, where (a) shows the configuration with annotations, and (b) shows the configuration in the proposed DSL. Note that when using the DSL, method declarations are annotated with **@Test**, all the other specifications are in a separated file. Furthermore, using the DSL allows to divide the configuration among different staff. For instance, given the complexity of the CBS management, one tester could write the test code, while other tester could schedule and manage test executions. This improves the collaboration among software testers.



(a) Test Methods (Java Annotation Approach)

(b) Test Methods (DSL Approach)

Figure 7.9 – Example of Test Methods Source Code

Provider Part

A *Provider* is divided into two sub-parts: *Resources* that describe the cloud provider available resources, and *Commands* that parameterize the CLI commands necessary to interact with cloud provider while leading elasticity testing. Similarly to *RequiredRes*, *CloudResources* specification also corresponds to the white parts in Figure 7.2. However, while *RequiredRes* specification is cloud provider-independent, *CloudResources* is specific for each cloud provider, which describes existing resources.

Listing 15 shows an example of *CloudResources* setup. In the example, we describe resources of Amazon EC2 cloud provider: an image with hypothetical identifier *ami-1234*, which runs the OS distribution *Ubuntu*, with version *7.04* and architecture *i386*, and machine types *m3.medium* and *m3.large*.

Listing 15 – Example of *CloudResources* Specification

```
Provider EC2 {
CloudResources {
 Images {
  ami-1234 {
   os {
    distribution = 'Ubuntu';
    version = '7.04';
    architecture = 'i386';
   }
  }
 }
 Machines {
  m3.medium {
   storageCapacity = 4;
   cpuFrequency = 2.6;
   ramCapacity = 3.75;
  }
  m3.large {
   storageCapacity = 32;
   cpuFrequency = 5.2;
   ramCapacity = 7.5;
  }
 }
}
...
```

Listing 16 shows an example of a parameterized command, which corresponds to the Amazon EC2 CLI command to create an instance. Each command specification has a fixed name, a CLI command, and several command arguments. The command name never changes, even when a *Command* specification is (re-)written for a further cloud provider. Conversely, the CLI command and its arguments dependend on the cloud provider. Furthermore, argument value is a reference to a parameter from other configurations. In the example, the values of arguments *–image-id* and *–instance-type* refer to an image and a machine type from *CloudResources*, respectively.

Listing 16 – Example of *Commands* Specification

```
...
Commands {
 instantiationCommand 'aws ec2 run-instances' {
  arguments {
   --image-id Image;
   --instance-type MachineType;
```

```
    }
  }
  ...
}
}
```

7.2.2 Compilation of Elasticity Test Configuration

Figure 7.10 depicts a methodology to compile DSL programs into executable code. In the figure, the cloud provider-dependent and the cloud provider-independent parts are in separated files. If the same test is executed on different cloud providers, the only file that the tester must change is the *Provider Dependent* one. Furthermore, this file is only written once, since a *Provider Dependent* file can be re-used from other executions. The compilation has two steps: *resource matching* and *script generation*. In the following sections, we detail these steps.

7.2.3 Resource Matching

The first compilation step consists in matching the required resources to the available resources from a cloud provider and in generating a resource matching file. This file contains the resource descriptions from the *Required Res* part, updated with the values of matched resources from *Cloud Resources*.

The matching function considers the minimum distance between resources described in both files, which is repeated for every required resource (res) in *RequiredRes*. Algorithm 4 illustrates this function. It is a brute-force algorithm that calculates the distance between a required resource ($res \subset RequiredRes$) and every available resource of the same type ($cr \subset CloudResources$). At the end, the algorithm returns the cloud resource (cr) with the least distance to the required resource (res). For numeric values we consider their arithmetic difference as distance metric, while for literal values we consider the Levenshtein distance [57].

Listing 17 shows an example of a resource matching file, where an instance named as *mediumUbuntu* runs image *ami-1234* on a machine of type *m3.medium*. Note that attribute names correspond to *instance* attributes in the model of Figure 7.2, while their values refer to resources of cloud provider Amazon EC2 described in *CloudResources* (Listing 15).

Listing 17 – Example of Resource Matching File (I_{EC2})

```
instance mediumUbuntu {
 image := ami-1234,
 machineType := m3.medium,
 geographicZone := eu-west-1
```

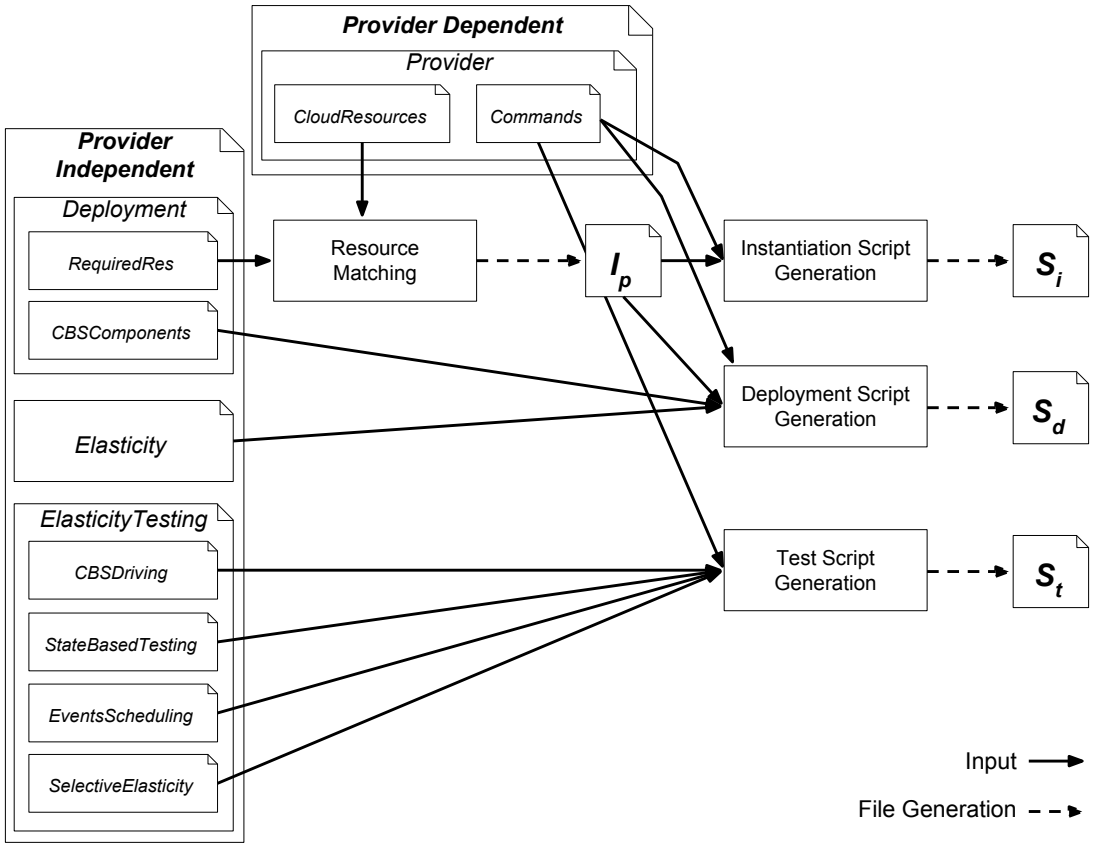


Figure 7.10 – Compilation of Elasticity Testing Setup into Executable Code

```
}
```

7.2.4 Script Generation

The second compilation step consists in generating all the scripts necessary to execute elasticity tests: resource instantiation (S_i), CBS deployment (S_d) and elasticity test (S_t) scripts. These scripts contain CLI commands to interact with the cloud provider.

Instantiation Script Generation

The instantiation script (S_i) commands instantiate the required resources on the Cloud. This script is generated by replacing argument values of the corresponding command in the specification by the attribute values from the resource matching file.

Listing 18 shows an example of the generated instantiation script (S_i). Since a

Algorithm 4: Resource Matching Function

```
Data: Required Resource req, Available Resources R
minDist ← infinity;
for i = 1 to lenght(R) − 1 do
    cr ← R[i];
    if dist(res, cr) < minDist then
        minDist ← dist(res, cr);
        matchedRes ← cr;
    end
end
return matchedRes;
```

resource instantiation code may be reused to deploy more than one CBS component, each instantiation code is placed in a block named *instantiation*. Note that there is exactly one instantiation block for each instance in the resource matching file. Instantiation scripts execute along with deployment scripts.

Listing 18 – Example of an Instantiation Script

```
instantiation mediumUbuntu {
aws ec2 run-instances --image-id ami-1234 --instance-type
    ↪ m3.medium --region eu-west-1
}
```

Deployment Script Generation

The deployment script (S_d) deploys CBS components on the Cloud, and configures the elasticity parameters. The deployment of a CBS component is done in three steps: launching an instance that meets the resource requirements, deploying the components, and configuring the provider elasticity parameters. The script generator gathers information from the *CBSComponents*, *Elasticity* and *Commands* parts and from the resource matching file (I_p).

Test Script Generation

Test script contains the commands necessary to execute test suites. The first task of the generation is to transform test specifications into inputs for testing tools. For instance, in state-based testing, this consists in transforming the *StateBasedTesting* specification into a Java file with annotated methods. The second task is to generate the commands to send such inputs to the testing tools, and to orchestrate their executions.

7.3 Experiment

This section presents an experiment that aims at measuring the tester effort when writing elasticity tests by using the DSL proposed in this chapter. As case studies, we consider two elasticity tests, previously presented in Chapters 5 and 6: *CS1*) the MongoDB bug 7974 reproduction (Section 6.3.1), and *CS2*) the performance testing of a distributed Web application throughout a preset elastic behavior (Section 5.3.2).

For each case study, we write the elasticity test by using the DSL proposed in this chapter, and translate it into three cloud provider CLIs: Amazon EC2, Google CP, and OpenStack. Then, we compare the tester effort to write elasticity tests in both, the DSL and the cloud provider CLIs. We measure the tester effort in number of words: *total of words*, and *cumulative words*.

7.3.1 Total of Words

The *total of words* is the amount of words necessary to write the tests. Table 7.1 presents the results for the case studies. The tests written in the DSL result in fewer words for all the cloud providers and case studies. They are the same for all the cloud providers, whereas tests written in CLIs are specific to each provider. The DSL reduces considerably the tester effort to write tests: Amazon EC2 ($CS1 \approx -24\%$, $CS2 \approx -22\%$), Google CP ($CS1 \approx -38\%$, and $CS2 \approx -36\%$), and OpenStack ($CS1 \approx -43\%$, and $CS2 \approx -39\%$).

Cloud Provider	**CS1**	**CS2**
DSL		
$All\ Cloud\ Providers$	213	192
CLI		
$Amazon\ EC2$	264	238
$Google\ CP$	292	265
$OpenStack$	298	275

Table 7.1 – Total of Words in Case Studies

Figure 7.11 depicts the tester effort to write the case studies. In the figure, the dashed line connects *CS1* efforts, while the solid line connects *CS2* efforts.

7.3.2 Cumulative Words

The *cumulative words* (CW) is the sum of the new words necessary to re-write an existing elasticity test to execute it in a further cloud provider infrastructure. We use the

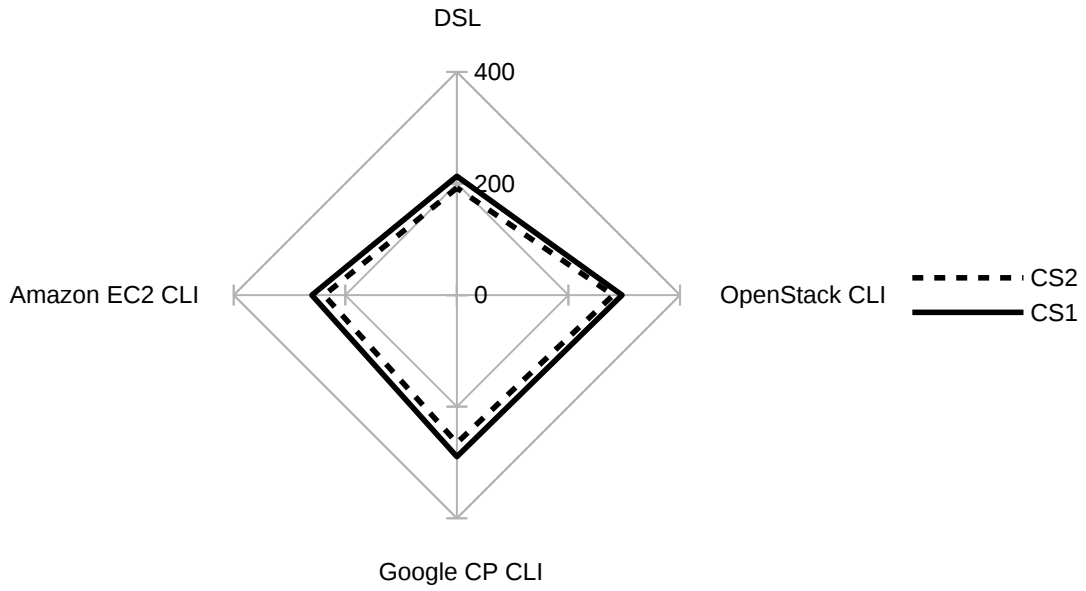


Figure 7.11 – Tester Effort to Write the Case Studies

following formula to calculate the CW:

$$\sum_{i=1}^{n} CW_{i-1} + (S_{i-1} \odot S_i) \tag{7.1}$$

The i denotes the sequence to write the elasticity test, and $S_{i-1} \odot S_i$ denotes the new words from the previous (S_{i-1}) to the next test (S_i).

The graph of Figure 7.12 illustrates the CW as the case studies are (re-)written for a given sequence of cloud provider infrastructures: Amazon EC2, Google CP, and OpenStack. The solid lines illustrate the CW for tests written in the DSL, while the dot-dashed lines illustrate the CW for tests written in the cloud provider CLIs. While the CW for tests written in the DSL is stable, it almost triples from the first to the last elasticity test for tests written in the cloud provider CLIs. This is because the tests written in the DSL are portable over cloud providers, while when using the cloud provider CLIs the test must be completely re-written.

7.4 Conclusion and Future Work

In this chapter, we proposed a DSL-based approach to configure the elasticity tests implementation. Its major contributions are the tester effort reduction to write elasticity

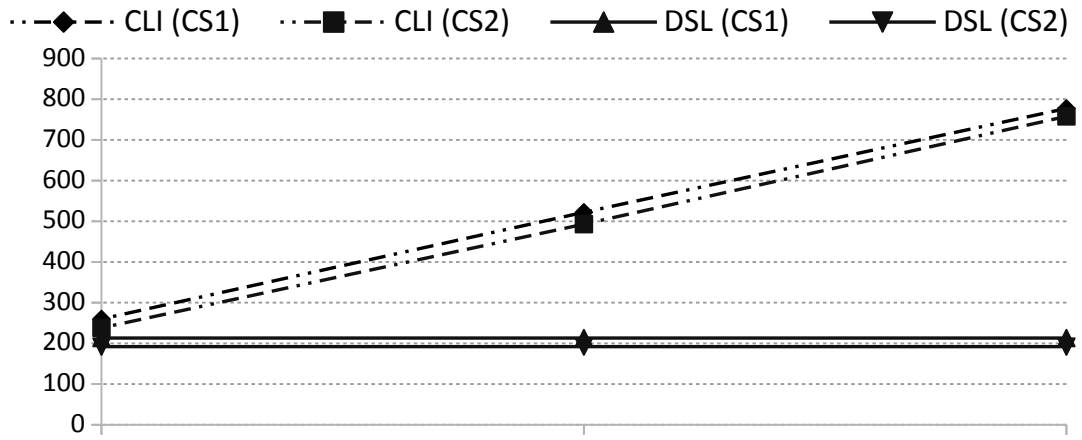


Figure 7.12 – Cumulative Words

tests, and the portability of tests over cloud providers. In the experiments, the tests written in the DSL contain less words than the tests written in the cloud provider CLIs. However, the major effort reduction is for elasticity tests that must be re-written to execute over different cloud provider infrastructures, where the same test can execute without any change This part of the work has been published at [20].

The compilation phase of the DSL-based approach is so far theoretical, which we wish to implement as part of a future work. The next challenge will be to tune this approach to find cloud computing resources automatically, where testers must not write cloud provider-dependent specifications. Another plan is to extend the DSL to test code level, in a way testers use it to write their tests rather than other languages, and test code calls testing approaches directly. Finally, a future plan is to change the DSL syntax, and make it closer to a programming language than a configuration file.

8

Generation of Test Sequences for Elasticity Testing

Elasticity testing admits many parameters. For instance, the CBS deployment requires to list the required resources, while different workloads drive the CBS throughout elasticity. Ideally, elasticity tests should cover all the combinations of elasticity testing parameters, i. e., test configurations. However, given the number of parameters, executing all the test configurations is time and cost prohibitive. In contrast, a random generation of test configurations may miss the critical parameter combinations that cause the CBS issues. Therefore, testers must generate a set of test configurations that is small enough to execute in a feasible time, and covers critical combinations of elasticity testing parameters. Furthermore, test configurations should be in a sequence, which mimics CBS reconfigurations and allows them to execute in a single run. Then, testers can implement different test oracles to test the CBS throughout this sequence.

This chapter presents an approach based on CIT to generate small sets of test configurations, and to arrange them as executable test sequences. We conduct a systematic experiment by using the proposed approach, which reveals several performance degradations in MongoDB, a CBS case study.

The remainder of this chapter is organized as follows. Section 8.1 presents the test sequence generation approach. Section 8.2 presents an experiment. Finally, Section 8.3 concludes and gives directions of future work.

8.1 Approach for Test Sequence Generation

This section presents an approach to generate elasticity tests and test sequences for elasticity testing. Elasticity testing parameters have a large configuration space, where testing all the configurations is prohibitive. Therefore, the tester needs to reduce the number of test configurations, and at same time, to ensure that they are relevant. Since in the literature CIT presents convincing results [63], we base the approach on it.

Figure 8.1 depicts the approach workflow, which is divided into three steps:

1. *Elasticity Modeling*: modeling the *elasticity testing parameters*, such as elasticity states, thresholds and workload, into a CTM (Section 8.1.1).
2. *Test Configurations Generation:* leveraging CIT algorithms to generate a set of test configurations covering all the T-wise interactions among the elasticity testing parameters (Section 8.1.2).
3. *Test Sequences Generation:* applying a graph traversal algorithm to generate test sequences covering all the possible transitions between test configurations (Section 8.1.3).

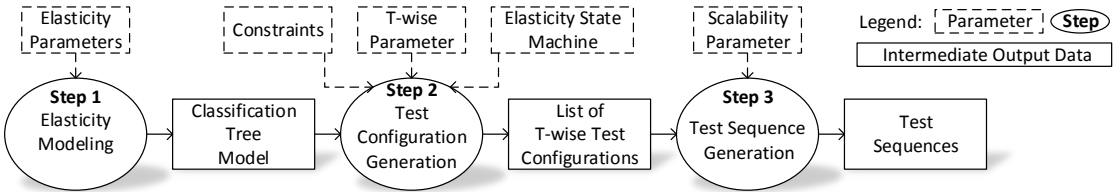


Figure 8.1 – Approach for Test Sequence Generation Workflow

8.1.1 Elasticity Modeling

In the first step of our approach, we use a CTM to represent the elasticity testing parameters. CTM structures system features into a hierarchical tree [67], where the root element is the system, and the branches (*compositions*) represent its components. Non-decomposable components are defined as *classifications*, structured in *classes*, which implement them. In a comparison between the CTM and the SUT model described in Section 3.3 ($M_{SUT} = \{P, V, C\}$), CTM classifications correspond to the parameters set P, while CTM classes correspond to the parameters values set V. The constraint set C is configured in the next step.

Figure 8.2 shows an example of CTM representing the controlled elasticity testing parameters. The root of the CTM is the elasticity property, i. e., the CBS characteristic the tester wants to verify. We decompose elasticity testing parameters into two main

compositions, namely the *cloud infrastructure*, which encompasses the parameters pertaining the CBS deployment, and the *benchmark*, which models the workload of the system.

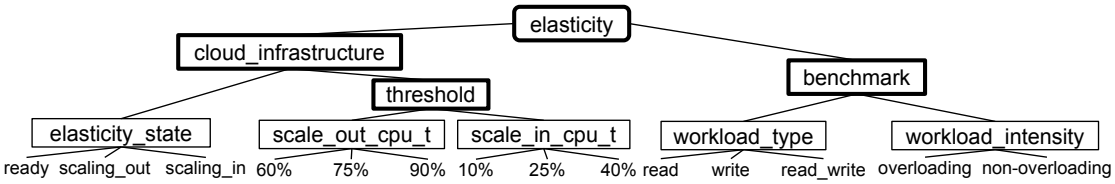


Figure 8.2 – CTM of Elasticity Testing Parameters

The cloud infrastructure is decomposed into the *elasticity_state* classification, and the *threshold* sub-composition containing classifications that represent scale-out (*scale_out_cpu_t*) and scale-in (*scale_in_cpu_t*) thresholds. The elasticity state classes represent the CBS states (see Figure 4.1). The classes in the classifications *scale_out_cpu_t* and *scale_in_cpu_t* are the usage percentages of the allocated resources. For example, a $60\,\%$ scale-out threshold entails that the system switches to the scaling-out state when the CPU usage exceeds $60\,\%$.

The *benchmark* composition is decomposed into the *workload_type* and *workload_intensity* classifications. In particular, the *workload_type* classes represent the three basic workload profiles consisting of read, write, or read and write operations. The *workload_intensity* classes represent two ways to drive the CBS through a scaling-out state: using a workload intensity that attempts to exhaust the allocated resource (*overloading*), or any other workload intensity (*non-overloading*).

8.1.2 Test Configuration Generation

In the second step of our approach, we propose a CIT-based methodology to generate test configurations. Before introducing the methodology for test configuration generation, we define the concept of test configuration.

A *test configuration* is a tuple

$$conf\_i = \langle v_1, v_2, \ldots, v_n \rangle$$

where $v_1 \in V_1, v_2 \in V_2, \ldots, v_n \in V_n$ and $V \in M_{SUT}$ (see Section 3.3). Considering a CTM, a test configuration is composed of one *class* of each *classification*. For instance, *conf_i* can refer to the first class of each classification:

$$conf\_i = \langle ready, 60\%, 10\%, read, overloading \rangle$$

The methodology consists in generating a *test configuration set*, where each test configuration covers T-wise combination of elasticity testing parameters and satisfies the constraints (C). This methodology only considers one *cross-tree* constraint, which models a particular aspect of the elasticity testing domain: a test configuration associated to the classes *ready* or *scaling-in* cannot be associated to *overloading*. Otherwise, an overloading workload could unexpectedly trigger a resource scale-out while the tester is testing the CBS in those two states. For instance, the *conf_i* is an invalid test configuration since it combines the *ready* and *overloading* classes.

This methodology reduces the number of test configurations while ensuring variety in the CTM classes, in a way that the number of test configurations and their variety increase with the value of T. For instance, the CTM in Figure 8.2 has 5 classifications, where the T value can vary from 2 to 5. Despite the proposed methodology is independent of the T value, previous studies on CIT present convincing results when using pairwise ($T = 2$) [55]. Therefore, we believe that pairwise should be sufficient to test elastic CBSs.

Table 8.1 lists all the test configurations satisfying pairwise interactions of elasticity testing parameters and the constraint. Note that only considering pairwise interactions, the number of test configurations reduces from 162 ($\prod_{i=1}^{N} |V_i|$) to 12.

	elasticity _state	**scale _out _cpu _t**	**scale _in _cpu _t**	**workload _type**	**workload _intensity**
2w-conf_0	*scaling_in*	*90%*	*40%*	*read_write*	*non_overloading*
2w-conf_1	*scaling_out*	*90%*	*25%*	*write*	*overloading*
2w-conf_2	*scaling_out*	*75%*	*10%*	*read*	*non_overloading*
2w-conf_3	*ready*	*60%*	*25%*	*write*	*non_overloading*
2w-conf_4	*scaling_out*	*60%*	*40%*	*read*	*overloading*
2w-conf_5	*scaling_out*	*60%*	*10%*	*read_write*	*overloading*
2w-conf_6	*scaling_in*	*75%*	*25%*	*read_write*	*non_overloading*
2w-conf_7	*scaling_in*	*60%*	*10%*	*write*	*non_overloading*
2w-conf_8	*ready*	*90%*	*10%*	*read_write*	*non_overloading*
2w-conf_9	*ready*	*75%*	*40%*	*read*	*non_overloading*
2w-conf_10	*scaling_in*	*90%*	*25%*	*read*	*non_overloading*
2w-conf_11	*scaling_out*	*75%*	*40%*	*write*	*overloading*

Table 8.1 – The Twelve Pairwise Test Configurations

8.1.3 Test Sequence Generation

The third step of the proposed approach concerns the test sequences generation. A test sequence is an ordered list of test configurations covering all the possible *reconfigurations*, i. e., transitions between test configurations. This mimics a real CBS behavior, where the CBS receives parameters that vary during its execution, such as the resource allocation and the workload. The test sequences generation is divided into three sub-steps: generation of a list of reconfigurations, generation of a reconfiguration tree, and selection of a set of test sequences.

Reconfiguration List Generation

Reconfigurations follow the transitions from the *elasticity states* model (Figure 4.1), where scaling-out and scaling-in states cannot succeed to one another, a ready state always precedes or follows them. Since each reconfiguration is a pair of test configurations, they are composed of a *previous test configuration* associated to the *elasticity_state* class *ready* and a *next test configuration* associated to the *elasticity_state* class *scaling_out* or *scaling_in*.

Table 8.2 shows an excerpt of the 54 reconfigurations among the 12 pairwise test configurations. This table reports the previous and next test configurations, and the change in the amount of resources. The change in the amount of resources regards the classification *elasticity_state* of the *next test configuration*. For instance, the next test configuration of reconfiguration $2w\text{-}reconf\_0$ is associated to the *elasticity_state* class *ready*, when the number of resources does not change ($=0$). In contrast, the next test configuration of reconfiguration $2w\text{-}reconf\_9$ is associated to the *elasticity_state* class *scaling_in*, when a resource deallocation occurs (-1). In the reconfiguration $2w\text{-}reconf\_10$, the next test configuration is associated to the *elasticity_state* class *scaling_out*, when a resource allocation occurs ($+1$).

	previous test configuration	next test configuration	changes in the amount of resource
2w-reconf_0	2w-conf_0	2w-conf_3	0
...	...	...	...
2w-reconf_3	2w-conf_1	2w-conf_3	0
...	...	...	...
2w-reconf_9	2w-conf_3	2w-conf_0	-1
2w-reconf_10	2w-conf_3	2w-conf_1	+1
2w-reconf_11	2w-conf_3	2w-conf_2	+1
...	...	...	...

Table 8.2 – Excerpt of the 54 Reconfigurations Between Pairwise Test Configurations

Reconfiguration Tree Generation

One can generate test sequences of any length by chaining reconfigurations in the reconfiguration list. Figure 8.3 illustrates this in a graph of the reconfiguration list excerpt in Table 8.2. The nodes of the graph represent test configurations, while the edges represent reconfigurations. Each edge has a number of resources allocated or deallocated during the reconfiguration, which corresponds to the last column of Table 8.2. A test sequence can generated covering a path over this reconfiguration graph.

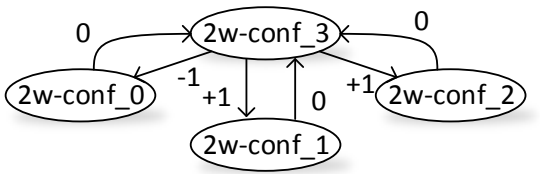


Figure 8.3 – Excerpt of the Pairwise Reconfigurations Graph

Since the graph in Figure 8.3 is cyclic, a test sequence could be infinite. To create finite test sequences, this graph is transformed into a tree, which we call *reconfiguration tree*. The tree root can be any test configuration associated to the *elasticity_state* class *ready*, regarding to the elasticity state at the FSM beginning in Figure 4.1. Furthermore, the tree branches cannot contain repeated reconfigurations.

As shown in the fourth column of Table 8.2, reconfigurations are associated to changes in resource allocation. Thus, particular paths over the graph could lead to resource exhaustion when resource is continuously de-allocated, while other paths could allocate too many resources. Therefore, the following constraints bound the amount of resources (a): the initial number of resource (i), the minimum number of resources (Min), and the maximum number of resources (Max).

Figure 8.4 illustrates a *reconfiguration tree* transformed from the graph in Figure 8.3. This tree root corresponds to a test configuration associated to the ready state ($2w$-$conf\_3$). Let us assume that $i = 1$, $Min = 1$ and $Max = 2$. On the first level (diamonds 1 and 2), $2w$-$conf\_0$ is discarded since it is associated to the *elasticity_state* class *scaling_in*, which would lead to an amount of resource lower than Min. Then, only $2w$-$conf\_1$ and $2w$-$conf\_2$ can occur. On the third level of the left branch (diamond 3), $2w$-$conf\_1$ and $2w$-$conf\_2$ are discarded since they are associated to the *elasticity_state* class *scaling_out*, which would lead the amount of resource upper the maximum. Finally, the two sequences end on the sixth level (diamonds 4 and 5) since all the reconfigurations (the six edges of each branch) were already used.

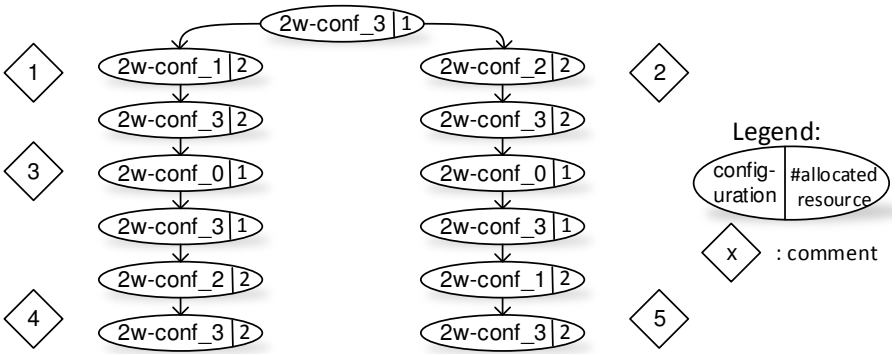


Figure 8.4 – Reconfiguration Tree from the Excerpt of Pairwise Reconfigurations Graph

Test Sequences Selection

Creating an optimal sequence that covers all the reconfigurations without repeating them is a NP-complete problem. Therefore, the approach creates several sequences, where each sequence covers a subset of unique reconfigurations, and then selects several sequences covering together all the reconfigurations.

Each branch of the reconfiguration tree (Figure 8.4) is a possible test sequence. However, a branch may not cover all the reconfigurations in the graph. In contrast, the longest branches maximize the chances to cover all the reconfigurations. Therefore, if a single branch does not cover all the reconfigurations, the approach uses as test sequences the longest branches that together cover all the reconfigurations. In the case of the tree in Figure 8.4, any branch covers all the reconfigurations.

8.2 Experiment

This section presents an experiment that aims at investigating the efficiency of test sequences generated by the approach proposed in this chapter to reveal CBS issues. In the experiment, the we generate test sequences by covering pairwise combination of elasticity testing parameters, and use the following constraints to bound the amount of resources: $i = 1$, $Min = 1$, and $Max = 2$. This results in two test sequences: test sequence 1 (*2w-TS1*) that contains 50 reconfigurations, and test sequence 2 (*2w-TS2*) that contains 47 reconfigurations. These test sequences cover together all the allowed reconfigurations between pairwise test configurations.

Equation 8.1 shows the test sequence *2w-TS1*:

$$\begin{aligned} 2w-TS1 = \{&conf\_3, conf\_11, conf\_8, conf\_10, conf\_8, conf\_11, conf\_3,\\ &conf\_7, conf\_3, conf\_4, conf\_8, conf\_0, conf\_3, conf\_2, conf\_9, conf\_0,\\ &conf\_8, conf\_4, conf\_3, conf\_10, conf\_3, conf\_5, conf\_3, conf\_0, conf\_9,\\ &conf\_1, conf\_9, conf\_7, conf\_9, conf\_11, conf\_9, conf\_6, conf\_9, conf\_4,\\ &conf\_9, conf\_10, conf\_9, conf\_2, conf\_8, conf\_7, conf\_8, conf\_2, conf\_3,\\ &conf\_6, conf\_3, conf\_1, conf\_3\} \end{aligned} \tag{8.1}$$

Equation 8.2 shows the test sequence *2w-TS2*:

$$\begin{aligned} 2w-TS2 = \{&conf\_8, conf\_2, conf\_8, conf\_10, conf\_8, conf\_11,\\ &conf\_3, conf\_10, conf\_3, conf\_5, conf\_9, conf\_7, conf\_3, conf\_11,\\ &conf\_9, conf\_0, conf\_9, conf\_5, conf\_8, conf\_0, conf\_3, conf\_2,\\ &conf\_9, conf\_6, conf\_3, conf\_4, conf\_9, conf\_10, conf\_9, conf\_4,\\ &conf\_8, conf\_6, conf\_9, conf\_1, conf\_3, conf\_7, conf\_9, conf\_2,\\ &conf\_3, conf\_0, conf\_8, conf\_4, conf\_3, conf\_6, conf\_8, conf\_1,\\ &conf\_8, conf\_7, conf\_8, conf\_5\} \end{aligned} \tag{8.2}$$

8.2.1 Experimental Setup

As a CBS case study, we use the document database MongoDB [9] deployed as a sharding cluster [16]: a *configuration server*, a *mongos* instance, and several *shard* instances. The configuration server stores meta-data, *mongos* instance works as a coordinator and a load balancer (routing queries and write operations to the shards), and *shard* instances store and process the data in a distributed manner.

We execute all the experiments on Amazon EC2. The *mongos* and the *configuration server* are deployed on the same VM ($m3$,$large$ type) [4], and each shard instance on a dedicated VMs ($t2$,$small$ type). The initial MongoDB configuration consists of one shard instance, and additional instances are allocated/deallocated according to the test configurations in the test sequences. Further software artifacts for workload generation and CBS driving are deployed in another VM ($c3$,$large$ type). As a benchmark tool, we use the YCSB, where the *elasticity driver* (Chapter 4) leads this tool to generate the workload during the experiment.

8.2.2 Test Oracle

In this experiment, we use a customized test oracle for performance testing, which we execute by using post-execution scripts. In the following, we explain the whole process of building and using the oracle.

As a performance metric, the test oracle calculates the median CBS throughput ($\tilde{t}$) for each test configuration over the 30 experiment executions. This discards executions where unpredictable conditions, such as bandwidth issues or concurrent processes, affect the CBS throughput.

Since a perfect performance is unrealistic, for each test configuration, the test oracle calculates the *performance deviation* (D_i) from the workload (w_i), defined as $D_i = \frac{\tilde{t}_i - w_i}{w_i}$. The test oracle also allows to configure several *tolerance levels* (L), where the higher is the tolerance, the higher is the tolerated performance deviation. Since the throughput is either less than or equal to the workload, the performance deviation is either 0 or negative, and hence we consider its absolute value ($|D|$). Then, for each tolerance level, the test oracle assigns performance verdicts (pv) to the test configurations by comparing the $|D|$ and L, as following:

$$pv_i \begin{cases} \textit{pass} & \textit{if } |D_i| \leq L \\ \textit{fail} & \textit{if } |D_i| > L \end{cases}$$

If the $|D_i|$ is less than or equal to L, then the verdict is *pass*. Otherwise, the verdict is *fail*.

8.2.3 Experimental Results

Executing the 97 reconfigurations (summing the configurations in *2w-TS1* and *2w-TS2*) takes ≈6 h on the Amazon EC2, while repeating them for 30 times takes ≈180 h, i. e., ≈7,5 d. Even if several test sequences execute in parallel, which reduces the execution time, this does not reduce the execution cost. The total time is expressive, where a wider coverage of elasticity testing parameters may be prohibitive. For instance, 3-wise coverage, only one combination wider than pairwise, results in 40 test configurations, and ≈2674 reconfigurations. Each execution of such reconfigurations takes ≈7 d if we consider the average time per configuration when executing pairwise reconfigurations (≈3,7 min). If they are re-executed 30 times, such as pairwise reconfigurations, it takes ≈7 months. We configure 7 tolerance levels in the test oracle: 0, 0,05, 0,10, 0,15, 0,20, 0,25, and 0,30. Figure 8.5 depicts the percentage of *fail* verdicts according to the tolerance level. All the verdicts are fail when the tolerance is at lowest level (0, i. e., no tolerance). Therefore, no test configuration achieves the ideal performance in the experiment, which is comprehensible since we are testing a distributed system under a massive sequence of reconfigurations. We also note that the percentage of fail verdicts decreases as the tolerance level increases, and that at tolerance level 0,35 there is no fail verdict. The most severe performance degradation happen at tolerance level 0,3.

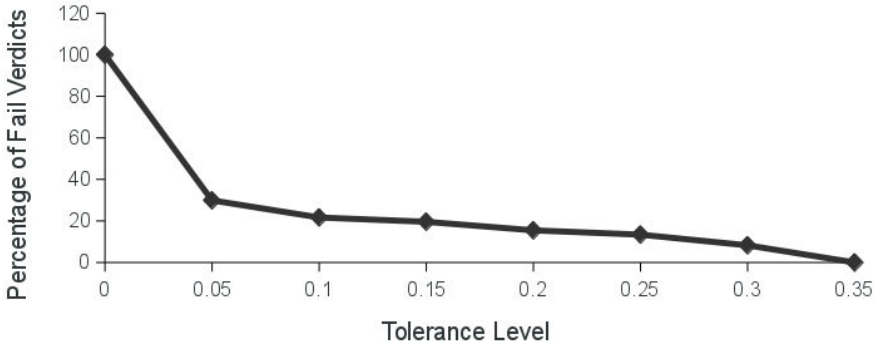


Figure 8.5 – Percentage of Fail Verdicts by Tolerance Level

Observations

Table 8.3 ranks the test configurations that fail (i. e., *unstable* test configurations) by the number of fail verdicts grouped by tolerance level, where we omit unstable test configurations at the lowest level (0%) since all the test configurations fail. All the unstable test configurations in this table are associated to the *elasticity_state* class *ready*. Furthermore, we do not find a pattern among the other classes despite the *workload_type* class *non-overloading*, which is always in test configurations associated to the *elasticity_state* classes *ready* and *scaling_in* due to cross-tree constraints.

Tolerance	Fail Verdicts	Unstable Test Configurations
0,30	8 (8%)	*2w-conf_8, 2w-conf_9*
0,25	13 (13%)	*2w-conf_8, 2w-conf_9*
0,20	15 (15%)	*2w-conf_8, 2w-conf_9*
0,15	19 (20%)	*2w-conf_8, 2w-conf_9*
0,10	21 (22%)	*2w-conf_8, 2w-conf_9, 2w-conf_3*
0,05	29 (30%)	*2w-conf_8, 2w-conf_9, 2w-conf_3*
0	97 (100%)	*

Table 8.3 – Unstable Configurations Classified by Tolerance

The test configurations in Table 8.3 are unstable when they are preceded by specific test configurations, which are highlighted in Table 8.4. In this table, each level of tolerance includes the unstable test configurations of the higher level, where tolerance 0,05 corresponds to all the reconfigurations of the table. Therefore, for each tolerance level, the table only shows new unstable reconfigurations, which the higher levels do not reveal. Note that tolerance level 0,25 does not include any new unstable reconfiguration compared to level 0,30.

In Table 8.4, all the previous test configurations are associated to the *elasticity_state*

Tolerance	Reconfiguration	
	Previous Test Configuration	**Next Test Configuration**
0,30	*2w-conf_1*	*2w-conf_8, 2w-conf_9*
	2w-conf_2	*2w-conf_8, 2w-conf_9*
	2w-conf_4	*2w-conf_8, 2w-conf_9*
	2w-conf_5	*2w-conf_8*
	2w-conf_11	*2w-conf_8*
0,25	–	–
0,20	*2w-conf_5*	*2w-conf_9*
	2w-conf_11	*2w-conf_9*
0,15	*2w-conf_4*	*2w-conf_3*
	2w-conf_11	*2w-conf_3*
0,10	*2w-conf_5*	*2w-conf_3*
0,05	*2w-conf_1*	*2w-conf_3*
	2w-conf_2	*2w-conf_3*

Table 8.4 – unstable Reconfigurations Classified by Tolerance

class *scaling_out*, and all the next test configurations are associated to the *elasticity_state* class *scaling_out*. Besides the *elasticity_state* classes, we do not find further patterns among those test configurations.

The test configuration *2w-conf_3* is only unstable when the tolerance level is higher than or equal to $0{,}2$, while the test configurations *2w-conf_8* and *2w-conf_9* are only unstable when the tolerance level is lower than or equal to $0{,}15$. This concerns the parameter scaling-out threshold (*scale_out_cpu_t*), given it is higher for the test configurations *2w-conf_8* and *2w-conf_9* (90% and 75% of CPU) than for the test configuration *2w-conf_3* (60% of CPU). Thus, CBS receives a workload with higher intensity, which makes the performance failure more severe.

Therefore, the highest tolerance levels ($0{,}30$-$0{,}20$) has the following reconfiguration pattern, where $*$ means any possible parameter value:

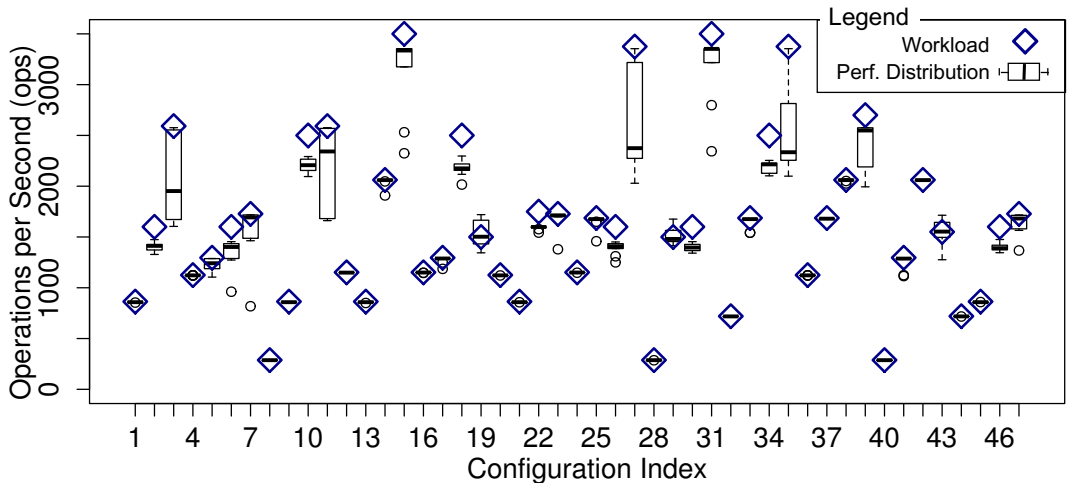
$$conf_{i-1} = \{scaling\text{-}out, *, *, *, *\}$$
$$conf_i = \{ready, \geq 75\%, *, *, *\}$$

In contrast, the lowest tolerance levels ($0{,}15$-$0{,}05$) has the following pattern:

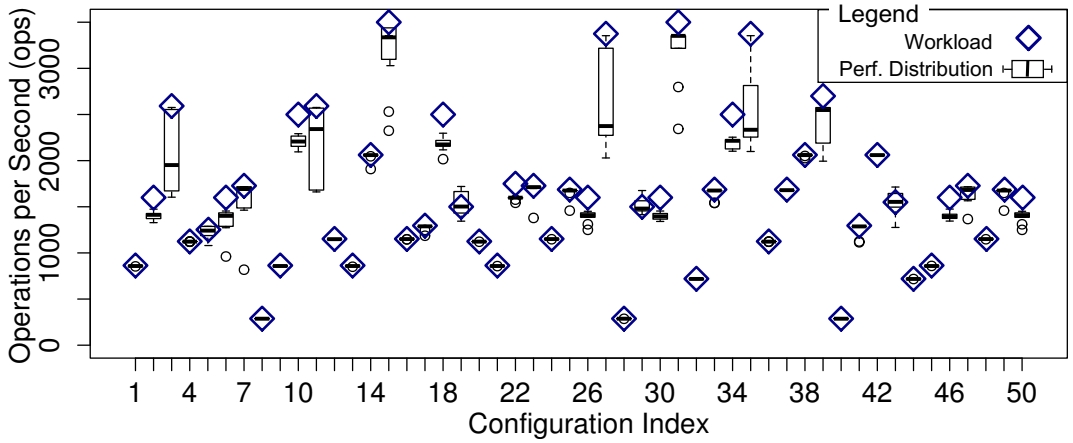
$$conf_{i-1} = \{scaling\text{-}out, *, *, *, *\}$$
$$conf_i = \{ready, \geq 60\%, *, *, *\}$$

Figure 8.6 depicts the CBS throughput of each test configuration in the pairwise test sequences *2w-TS1* (Figure 8.6a) and *2w-TS2* (Figure 8.6b). The diamonds represent the workload, while box-and-whisker plots represent the CBS throughput distribution for each test configuration over 30 executions. At some test configurations, the performance

variation is high, such as for the test configuration at the index 3. Such test configurations have their median values distant from the workload, what mean they are unstable. Those test configurations match to the ones listed in the column *next test configuration* of Table 8.4, when the tolerance level is high (0,30-0,20).



(a) Throughput of Each Test Configuration in *2W-TS1*



(b) Throughput of Each Test Configuration in *2W-TS2*

Figure 8.6 – Measured Throughput for Pairwise Test Sequences

Discussing the Performance Testing Fail Verdicts

All the performance degradations occur due to a load balancing problem. The newest shard, added during the scaling-out state that precedes the unstable test configuration, does not receive as many requests as the existing one (oldest shard), which exhausts the oldest shard.

Pairwise Adequacy

One could argue that pairwise coverage may not be effective, and a wider combination of elasticity testing parameters could find further CBS issues. However, this would be cost and time-prohibitive. As we already discussed, executing test sequences execution generated with one combination wider than pairwise (i. e., 3-wise) would take $\approx 7\,\mathrm{months}$. Furthermore, preliminary results of an experiment where we only execute part of 3-wise reconfigurations, suggest that pairwise test sequences reveal all the performance degradations of the case study.

The three-wise test sequences we executed cover all the 3-wise test configurations, though they only cover a small part of their reconfigurations. Those test sequences do not reveal new unstable test configurations or reconfigurations, where all of the unstable elasticity test configurations found using three-wise follow the same pattern of pairwise. However, we leave new tests considering larger combination of elasticity testing parameters for future work, when we also plan to execute further test oracles.

8.3 Conclusion

In the experiments, we applied the approach proposed in this chapter to create pairwise test sequences for testing the MongoDB NoSQL database. These test sequences reveal several significant performance degradations, where we identify a pattern for unstable reconfigurations. Preliminary comparison to results of partial three-wise test sequences suggests that pairwise identifies all the performance degradations of our case study. This part of the work has been published at [21].

The test oracle we use to test the MongoDB in the experiments is specific for performance testing. However, the elasticity test generation approach is independent of the test oracle.

Our work is the first step into generating test sequences for elasticity testing, and the preliminary results encourage further investigations. As a future work, we plan to conduct a complete experimentation that identifies which is the minimum coverage of elasticity testing parameters to reveal most of or all the relevant elasticity-related issues. This also requires a deeper evaluation of elasticity testing parameters, such as further workload configurations and a scalability larger than two allocated resources. A further investigation should also consider a larger test suite, which covers other aspects besides the CBS performance, and further CBS case studies.

9

Conclusion

In this chapter, we present this thesis conclusion. First, we recapitulate the requirements in elasticity testing. Second, we summarize the gaps in the state of the art. Then, we summarize the contributions of this thesis. Finally, we present the perspectives of future work.

9.1 Elasticity Testing Main Problems

In this thesis, we focus on five elasticity testing requirements, which we recapitulate here:

1. CBS driving throughout elasticity,
2. elasticity tests synchronization with CBS states,
3. elasticity test reproduction,
4. elasticity test implementation, and
5. elasticity test generation.

One of the elasticity testing needs is to drive the CBSs throughout preset elastic behaviors. However, estimating the workload variations is a real challenge, since the same workload may impact different CBSs in distinct manners. This needs emphasize the lack of a procedure to estimate and control workload variations according to a preset elastic behavior.

Since CBSs can go through different elasticity states, triggering CBS adaptations, these adaptions must be be state-based. For this, testers must identify the elasticity states at real-time, and coordinate test execution accordingly.

Testers must execute regression testing regularly to detect and correct bugs. The elasticity test reproduction requires to repeat the previous CBS elastic behavior in parallel with further conditions, such as user interactions with the CBS, which we call *time-based events*. In addition, the elastic behavior control requires to manage specific CBS component variations. Therefore, we need an approach that reproduces CBS elastic behavior by regarding the management of specific CBS components, and coordinates *time-based events*.

Elasticity testing is complex and laborious, given the quantity of parameters involved, and requires the tester to master cloud computing concepts and tools. The literature introduces some work based on high-level languages that make elastic CBS setup less complex and laborious. However, none of them addresses the setup of elasticity or elasticity testing. This requires an approach that abstracts the complete writing of elasticity testing.

Finally, elasticity testing configuration space is large, and test cases generated randomly may miss important configurations. In contrast, generating elasticity tests that cover all the possible configurations is time and cost prohibitive. Therefore, it is necessary to generate a minimum number of elasticity tests that still reveal relevant elasticity-related issues.

9.2 Contributions

In this thesis, we propose five contributions to tackle the presented problems. Here, we recapitulate these contributions.

For the CBS driving, we propose an approach that profiles the impact of workloads in CBSs, and calculates the necessary workload variations to drive the CBS through a given elastic behavior. Then, the approach controls the workload generation along with the test execution. The preliminary results show that the approach can drive CBS throughout preset elastic behaviors in a short time.

We tackled the elasticity state-based testing problem with an approach that identifies the different CBS elasticity states at run-time, and switches among tests accordingly. Thank to this approach, we discovered the causes of performance degradations of a case study. For such case study, the approach identifies the elasticity state where performance degradations occur, and tests the CBS during the this state.

We also proposed a prototype that tackles the elasticity test reproducibility problem. This prototype also accelerates the tests reproduction. We used this prototype to reproduce 3 representative bugs.

We proposed a DSL-based approach to specify elasticity tests, and a way to execute these tests in a cloud provider-independent manner. This approach reduces the amount of words needed to write cloud provider-independent elasticity test specifications.

Finally, aiming at minimizing the number of test configurations, we proposed an

approach based on CIT. This approach consists in generating test configurations that cover all T-wise combinations of elasticity test parameters. Then, the approach generates sequences of test cases that mimics a realistic elastic behavior. In the preliminary results, this approach encourages further investigations, where test configurations generated by covering pairwise combination of parameters reveal several elasticity-related issues in a case study.

9.2.1 Future Work Perspectives

As a global goal, we intend to reassemble all the contributions in a testing framework, and make it available for the research community. However, before this step, we aim at improving each contribution separately.

The elasticity driver considers that the CBS scales linearly, which may not be true depending on the CBS or the allocated resource amount. Therefore, we first plan to propose an strategy that considers non-linear scalability. Second, we plan to profile more generic workloads, such as workloads based on log traces. Finally, we plan to adapt the elasticity driver to address predictive elasticity policies since so far it only works with reactive ones.

The elasticity state-based testing approach reduces the efforts in executing elasticity tests by executing them during the CBS elasticity states. However, we can still reduce this effort by identifying more specific CBS states. Since this approach main idea is to test CBS adaptations, we plan to improve the CBS states monitoring to identify CBS states based on adaptations rather resource variation. This will reduce the time within each test executes, and as a consequence, will reduce their execution efforts.

Testing is not only about reproducing existing bugs, but also diagnosing them. Therefore, an evolution for the reproduction approach is to generate elasticity tests rather than writing tests to repeat previous executions. We plan to model this approach features as part of the CTM for the elasticity test generation approach. Then, we can generate elasticity tests that cover the selective elasticity and time-based events. For instance, for a client-server CBS architecture, we could model a selective elasticity classification with two classes: *server* and *client*. Then, an elasticity test sometimes deallocates a *server*, while other times it deallocates a *client*. Another future investigation of the elasticity test reproduction approach is to investigate how fast a test execution can be executed without compromising the CBS behavior.

The compilation phase of the DSL is so far theoretical, which we wish to implement as part of a future work. The next challenge will be to tune this approach to find cloud computing resources automatically in a way that testers must not write the cloud provider-dependent specifications. Another plan is to extend the DSL to test code level. Then, testers could use it to write their tests rather than other languages, and test code can have direct interactions with testing approaches. Finally, a future plan is to change the DSL syntax, and make it closer to a programming language than to a configuration file.

As a future work on elasticity test generation, we plan to conduct a complete experimentation that identifies which is the minimum coverage of elasticity testing parameters to reveal most of or all the relevant elasticity-related issues of different CBSs. This also requires a deeper evaluation of elasticity testing parameters, such as further workload configurations and a scalability larger than two allocated resources. A further investigation should also consider a larger test suite, which covers other aspects besides the CBS performance.

List of Terms

Amazon EC2 Amazon Elastic Compute Cloud. 6, 26, 50, 61, 62, 72, 74, 78–80, 95–97, 100, 101, 110, 111, 121

API Application Programming Interface. 19, 33, 49, 71, 77, 78, 121

AR Allocated Resource. 46, 47, 51, 121

ARU Average Resource Usage. 43, 44, 51, 121

AWS Amazon Web Services. 121

CBS Cloud-Based System. 5–14, 17–23, 25, 27, 33–35, 37–39, 41–53, 55, 57, 58, 60–64, 66–69, 71–73, 80, 81, 83, 84, 89–91, 95, 98, 99, 103–107, 109–111, 113, 115, 117–121, 125, 129, 138

CIT Combinatorial Interaction Testing. 11, 30, 103–106, 119, 121

CLI Command-line Interface. 10, 19, 83, 95, 96, 98, 100–102, 121

Cloud Provider Company that manages a public cloud infrastructure.. 121

CM Cloud Monitor. 13, 71–73, 121

CPU Central Processing Unit. 25, 84, 121

CT Combinatorial Testing. 121

CTM Classification Tree Model. 11, 21, 33, 104–106, 119, 121, 130

DBMS Database Management Systems. 121

DSL Domain Specific Language. 9, 14, 21, 22, 34, 35, 83, 89, 90, 95, 97, 100–102, 118, 119, 121, 138

EC Elasticity Changes. 71, 121

ECM Elasticity Controller Mock. 13, 70–73, 121

ES Elasticity States. 121

ETS Elastic Transition Sequence. 121

EvS Event Scheduler. 13, 71–73, 121

FM Feature Model. 33, 121
FSM Final State Machine. 33, 108, 121

Google CP Google Cloud Platform. 6, 26, 61–64, 66, 72, 100, 101, 121

HTTP Hypertext Transfer Protocol. 50, 61, 64, 65, 91, 121

IaaS Infrastructure as a Service. 23, 24, 26, 121, 129
IP Internet Protocol. 121
ISTQB International Software Testing Qualifications Board. 121

JAR Java Archive. 121

MDEs Minimum Distance Estimation. 121

OPS Operations per Second. 26, 51, 52, 121
OS Operating System. 84, 90, 95, 121

PaaS Platform-as-a-Service. 23, 24, 121
PAYG Pay-As-You-Go. 121
PGE Payment Gateway Emulator. 121
PI Profiling Intensity. 43, 44, 121

QoS Quality of Service. 6, 18, 33, 121

RBE Remote Browser Emulator. 121
RES Required Elasticity States. 44–47, 50, 53, 58, 61, 64, 72, 121
RMI Remote Method Invocation. 48, 121

SaaS Software-as-a-Service. 23, 24, 34, 121
SER Selective Elasticity Requests. 71, 121
SPL Software Product Line. 33, 121
SSH Secure Shell. 121
SUT System Under Test. 30, 38, 104, 121

TaaS Test-As-A-Service. 34, 35, 121
the Cloud cloud computing infrastructure. 17, 21, 23–25, 37, 43, 73, 84, 121

UML Unified Modeling Language. 42, 48, 84, 85, 121

URL Uniform Resource Locator. 121

vCPU Virtual CPU. 121

VM Virtual Machine. 25–27, 48, 50, 51, 61, 68, 84, 110, 121

WGen Workload Generator. 13, 70–73, 121

WIPS Web Interactions Per Second. 121

YCSB Yahoo! Cloud Serving Benchmark. 75, 110, 121

Contents

List of Tables

List of Figures

Bibliography

[1] Apache ZooKeeper Web site. https://zookeeper.apache.org/. [Online, Accessed 08-Feb-2017]. 73

[2] ClarkNet-HTTP - Two Weeks of HTTP Logs from the ClarkNet WWW Server. [Online, Accessed 10-Jun-2017]. 37

[3] Force a Member to Become Primary — MongoDB Manual 3.0. https://docs.mongodb.com/v3.0/tutorial/force-member-to-be-primary/. [Online, Accessed 08-Feb-2017]. 78

[4] Instances of Amazon EC2 – Amazon Web Services (AWS). http//aws.amazon.com/fr/ec2/instance-types/. [Online, Accessed 08-Feb-2017]. 74, 110

[5] ISO/IEC/IEEE 29119 Software Testing Standard. http://www.softwaretestingstandard.org/index.php. [Online, Accessed 17-May-2017]. 29, 30, 133

[6] ISTQB Glossary. http://astqb.org/glossary/. [Online, Accessed 13-Feb-2017]. 29

[7] JSch - Java Secure Channel. http://www.jcraft.com/jsch/. [Online, Accessed 01-06-2016]. 78

[8] Mongodb bug 7974: Suppress stack trace on replication errors. https://jira.mongodb.org/browse/SERVER-7974. [Online, Accessed 29-May-2017]. 19, 67

[9] MongoDB Web site. https://www.mongodb.org/. [Online, Accessed 21-Sep-2016]. 110

[10] Slashdot: News for nerds, stuff that matters. https://slashdot.org/?SetFreedomCookie. [Online, Accessed 08-Feb-2017]. 50

[11] Three Member Replica Sets — MongoDB Manual 3.2. https://docs.mongodb.com/v3.2/core/replica-set-architecture-three-members/. [Online, Accessed 08-Feb-2017]. 19, 67, 75

[12] TPC-W - Homepage. http://www.tpc.org/tpcw/. [Online, Accessed 10-Jun-2017]. 37

[13] Zookeeper bug 2164: Fast leader election keeps failing. https://issues.apache.org/jira/browse/ZOOKEEPER-2164. [Online, Accessed 08-Feb-2017]. 19, 67, 76

[14] ZooKeeper bug 2172: Cluster crashes when reconfig a new node as a participant. https://issues.apache.org/jira/browse/ZOOKEEPER-2172. [Online, Accessed 08-Feb-2017]. 76

[15] IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, Dec 1990. 8

[16] Amazon EC2 AutoScaling CoolDown. http://docs.aws.amazon.com/autoscaling/latest/userguide/Cooldown.html, 2016. [Online, Accessed 19-Sep-2016]. 110

[17] Chef Website. https://www.chef.io/chef/, August 2016. [Online, Accessed 08-Feb-2017]. 33, 34, 35

[18] Puppet Website. https://puppet.com/, August 2016. [Online, Accessed 08-Feb-2017]. 33, 34, 35

[19] Divyakant Agrawal, Amr El Abbadi, Sudipto Das, and Aaron J. Elmore. Database scalability, elasticity, and autonomy in the cloud. *DASFAA'11 Proceedings of the 16th international conference on Database systems for advanced applications*, pages 2–15, April 2011. 24

[20] Michel Albonico, Amine Benelallam, Jean-Marie Mottu, and Gerson Sunyé. A dsl-based approach for elasticity testing of cloud systems. In *Proceedings of the International Workshop on Domain-Specific Modeling, DSM@SPLASH 2016, Amsterdam, Netherlands, October 30, 2016*, pages 8–14, 2016. 102

[21] Michel Albonico, Stefano di Alesio, Jean-Marie Mottu, Sagar Sen, and Gerson Sunyé. A dsl-based approach for elasticity testing of cloud systems. In *Proceedings of the International Conference on Cloud Computing (CLOUD) 2016, Honolulu, USA*, 2017. 115

[22] Michel Albonico, Jean-Marie Mottu, and Gerson Sunyé. Controlling the Elasticity of Web Applications on Cloud Computing. In *Proc. of the 31st SAC*. ACM, 2016. 55

[23] Michel Albonico, Jean-Marie Mottu, Gerson Sunyé, and Frederico Alvares. Making cloud-based systems elasticity testing reproducible. In *CLOSER 2017 - Proceedings of the 7th International Conference on Cloud Computing and Services Science, Porto, Portugal, April 24-26, 2017.*, pages 495–502, 2017. 81

[24] Lee Badger, Tim Grance, Robert Patt-Comer, and Jeff Voas. *Draft Cloud Computing Synopsis and Recommend- ations*. Nist Special Publication 800-146, 2011. 5, 23, 24

[25] X. Bai, M. Li, X. Huang, W. T. Tsai, and J. Gao. Vee@Cloud: The virtual test lab on the cloud. In *2013 8th International Workshop on Automation of Software Test (AST)*, pages 15–18, May 2013. 37

[26] Marcello M. Bersani, Domenico Bianculli, Schahram Dustdar, Alessio Gambi, Carlo Ghezzi, and Srđan Krstić. Towards the Formalization of Properties of Cloud-based Elastic Systems. In *Proceedings of PESOS 2014*, New York, NY, USA, 2014. ACM. 5, 6, 12, 17, 24, 25, 26, 43, 133

[27] L. d Bousquet, M. Delahaye, and C. Oriat. Applying a Pairwise Coverage Criterion to Scenario-Based Testing. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 83–91, April 2016. 30, 33

[28] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and Regression Trees*. Wadsworth Publishing, Belmont, Calif, June 1983. 33

[29] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, July 1997. 30

[30] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM. 37, 75

[31] Emanuel Ferreira Coutinho, Danielo Goncalves Gomes, and Jose Neuman de Souza. An analysis of elasticity in cloud computing environments based on allocation time and resources. In *2nd IEEE Latin American Conference on Cloud Computing and Communications*, pages 7–12. IEEE, December 2013. 36, 37, 57

[32] Carlo A. Curino, Djellel E. Difallah, Andrew Pavlo, and Philippe Cudre-Mauroux. Benchmarking OLTP/Web Databases in the Cloud: The OLTP-bench Framework. In *Proceedings of the Fourth International Workshop on Cloud Data Management*, CloudDB '12, pages 17–20, New York, NY, USA, 2012. ACM. 37

[33] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based Testing in Practice. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 285–294, New York, NY, USA, 1999. ACM. 32

[34] Eduardo Cunha de Almeida. *Testing and Validation of Peer-to-peer Systems*. PhD thesis, University of Nantes, 2009. 29

[35] Edwin Schouten. Cloud computing defined: Characteristics & service levels. `https://www.ibm.com/blogs/cloud-computing/2014/01/cloud-computing-defined-characteristics-service-levels/`. [Online, Accessed 23-Jan-2017]. 24, 133

[36] Emelie Engstrom, Per Runeson, and Mats Skoglund. A systematic review on regression test selection techniques. *Information and Software Technology*, 52(1):14–30, January 2010. 8, 19, 67

[37] S. Gaisbauer, J. Kirschnick, N. Edwards, and J. Rolia. VATS: Virtualized-Aware Automated Test Service. In *Fifth International Conference on Quantitative Evaluation of Systems, 2008. QEST '08*, pages 93–102, September 2008. 37

[38] Guilherme Galante and Luis Carlos E. De Bona. A Survey on Cloud Computing Elasticity. *2012 IEEE Fifth International Conference on Utility and Cloud Computing*, (1):263–270, November 2012. 24, 25, 133

[39] A. Gambi, W. Hummer, H. L. Truong, and S. Dustdar. Testing Elastic Computing Systems. *IEEE Internet Computing*, 17(6):76–82, November 2013. 7, 8, 18, 20, 30

[40] Alessio Gambi, Waldemar Hummer, and Schahram Dustdar. Automated testing of cloud-based elastic systems with AUToCLES. In *2013 28th IEEE/ACM International Conference*

on Automated Software Engineering (ASE), pages 714–717. IEEE, November 2013. 33, 34, 35

[41] Alessio Gambi, Waldemar Hummer, and Schahram Dustdar. Testing elastic systems with surrogate models. In *Proc. of the 1st CMSBSE*, pages 8–11. IEEE, May 2013. 37, 38

[42] Alessio Gambi, Waldemar Hummer, Hong-Linh Truong, and Schahram Dustdar. Testing Elastic Computing Systems. *IEEE Internet Computing*, 17(6):76–82, November 2013. 12, 37, 38, 43

[43] Erich Gamma and Kent Beck. Junit: A cook's tour. *Java Report*, 1999. 79

[44] J. Gao, X. Bai, W. T. Tsai, and T. Uehara. SaaS Testing on Clouds - Issues, Challenges and Needs. In *2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*, pages 409–415, March 2013. 37

[45] J. Gao, X. Bai, W. T. Tsai, and T. Uehara. Testing as a Service (TaaS) on Clouds. In *2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*, pages 212–223, March 2013. 37

[46] Mats Grindal, Jeff Offutt, and Sten F. Andler. Combination testing strategies: a survey. *Software Testing, Verification and Reliability*, 15(3):167–199, September 2005. 30

[47] N. R. Herbst, S. Kounev, A. Weber, and H. Groenda. BUNGEE: An Elasticity Benchmark for Self-Adaptive IaaS Cloud Environments. In *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 46–56, May 2015. 37

[48] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in Cloud Computing: What It Is, and What It Is Not. *ICAC*, pages 23–27, 2013. 5, 17, 18, 24

[49] Kai Hu, Lei Lei, and Wei-Tek Tsai. Multi-tenant Verification-as-a-Service (VaaS) in a cloud. *Simulation Modelling Practice and Theory*, 60:122–143, January 2016. 37

[50] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In Paul Barham and Timothy Roscoe, editors, *2010 USENIX, Boston, MA, USA, 2010*. USENIX Association, 2010. 75

[51] Sadeka Islam, Kevin Lee, Alan Fekete, and Anna Liu. How a consumer can measure elasticity for cloud platforms. In *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering - ICPE '12*, page 85, New York, New York, USA, April 2012. ACM Press. 35, 37

[52] Sadeka Islam, Srikumar Venugopal, and Anna Liu. Evaluating the Impact of Fine-scale Burstiness on Cloud Elasticity. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 250–261, New York, NY, USA, 2015. ACM. 37, 38

[53] Cem Kaner, Jack L. Falk, and Hung Quoc Nguyen. *Testing Computer Software, Second Edition*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 1999. 30

[54] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990. 33

[55] D. Richard Kuhn and Michael J. Reilly. An Investigation of the Applicability of Design of Experiments to Software Testing. In *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02)*, SEW '02, pages 91–, Washington, DC, USA, 2002. IEEE Computer Society. 106

[56] R. Kuhn, R. Kacker, Y. Lei, and J. Hunter. Combinatorial Software Testing. *Computer*, 42(8):94–96, August 2009. 32

[57] VI Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966. 97

[58] Connie U. Smith Lloyd G. Williams. Web Application Scalability: A Model-Based Approach. pages 215–226, 2004. 46

[59] A. Milicevic, J. P. Near, E. Kang, and D. Jackson. Alloy*: A General-Purpose Higher-Order Relational Constraint Solver. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 609–619, May 2015. 33

[60] David Mosberger and Tai Jin. httperf - A Tool for Measuring Web Server Performance. *SIGMETRICS Perform. Eval. Rev.*, 1998. 61

[61] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. 29, 30

[62] Cu D. Nguyen, Alessandro Marchetto, and Paolo Tonella. Combining Model-based and Combinatorial Testing for Effective Test Case Generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 100–110, New York, NY, USA, 2012. ACM. 30, 33

[63] Changhai Nie and Hareton Leung. A Survey of Combinatorial Testing. *ACM Comput. Surv.*, 43(2):11:1–11:29, February 2011. 11, 21, 30, 104

[64] Gilles Perrouin, Sebastian Oster, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Traon. Pairwise Testing for Software Product Lines: Comparison of Two Approaches. *Software Quality Journal*, 20(3-4):605–643, September 2012. 30, 33

[65] S. Rajan and A. Jairath. Cloud Computing: The Fifth Generation of Computing. pages 665–667, 2011. 23

[66] S. Sen, S. D. Alesio, D. Marijan, and A. Sarkar. Evaluating Reconfiguration Impact in Self-Adaptive Systems – An Approach Based on Combinatorial Interaction Testing. In *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*, pages 250–254, August 2015. 30, 33

[67] Sagar Sen, Stefano Di Alesio, Dusica Marijan, and Arnab Sarkar. Evaluating reconfiguration impact in self-adaptive systems–an approach based on combinatorial interaction testing. In *Software Engineering and Advanced Applications (SEAA), 2015 41st Euromicro Conference on*, pages 250–254. IEEE, 2015. 104

[68] D.M. Shawky and A.F. Ali. Defining a measure of cloud computing elasticity. In *2012 1st International Conference on Systems and Computer Science (ICSCS)*, pages 1–5, August 2012. 36, 37

[69] M. Silva, M. R. Hines, D. Gallo, Q. Liu, K. D. Ryu, and D. d Silva. CloudBench: Experiment Automation for Cloud Environments. In *2013 IEEE International Conference on Cloud Engineering (IC2E)*, pages 302–311, March 2013. 37

[70] K. Sledziewski, B. Bordbar, and R. Anane. A DSL-Based Approach to Software Development and Deployment on Cloud. In *2010 24th IEEE International Conference on Advanced Information Networking and Applications (AINA)*, pages 414–421, April 2010. 33, 34, 35

[71] K. C. Tai and Y. Lie. A Test Generation Strategy for Pairwise Testing. *IEEE Trans. Softw. Eng.*, 28(1):109–111, January 2002. 32

[72] A. Thiery, T. Cerqueus, C. Thorpe, G. Sunye, and J. Murphy. A DSL for Deployment and Testing in the Cloud. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 376–382, March 2014. 33, 34, 35, 89

[73] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, December 2008. 23

[74] Martti Vasar, Satish Narayana Srirama, and Marlon Dumas. Framework for Monitoring and Testing Web Application Scalability on the Cloud. In *Proceedings of the WICSA/ECSA 2012 Companion Volume*, WICSA/ECSA '12, pages 53–60, New York, NY, USA, 2012. ACM. 37, 38

[75] Joe Weinman. *Time is Money: The Value of "On-Demand"*. 35, 36, 37, 133

[76] Le Xu, Dijiang Huang, Wei-Tek Tsai, and Robert K. Atkinson. V-Lab: A Mobile, Cloud-Based Virtual Laboratory Platform for Hands-On Networking Courses. *International Journal of Cyber Behavior, Psychology and Learning (IJCBPL)*, 2(3):73–85, 2012. 37

[77] N. Yigitbasi, A. Iosup, D. Epema, and S. Ostermann. C-Meter: A Framework for Performance Analysis of Computing Clouds. In *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 472–477, May 2009. 37

[78] D. Yokoyama, V. Dias, H. Kloh, M. Bandini, F. Porto, B. Schulze, and A. Mury. The Impact of Hypervisor Layer on Database Applications. In *2012 IEEE Fifth International Conference on Utility and Cloud Computing*, pages 247–254, November 2012. 5, 23

UNIVERSITE BRETAGNE LOIRE

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Thèse de Doctorat

ALBONICO MICHEL

Test de Système Élastiques Basés Sur le Cloud

Controlling Cloud-Based Systems for Elasticity Testing

Résumé
Les systèmes déployés dans être testés pendant l'élasticité, ce qui entraîne plusieurs problématiques. D'abord, l'exécution d'un test pendant l'élasticité peut exiger de conduire le CBS dans une succession de comportements élastiques spécifiques, càd., une séquence d'ajout/retrait de ressources, qui nécessite des variations précises de la charge des requêtes envoyées au cloud. Seconde, certaines adaptations du CBS ne sont réalisées qu'à un moment précis, par exemple après un ajout de ressources et, par conséquent, leurs tests doivent être synchronisés avec des états spécifiques du CBS. Troisième, les testeurs doivent rejouer les tests pendant l'élasticité de manière déterministe afin de déboguer et corriger le CBS. Quatrième, la création des tests pendant l'élasticité est complexe et laborieuse dû au large nombre de paramètres, et à la particularité du cloud computing. Enfin, seulement quelques combinaisons de paramètres peuvent causer des problèmes au CBS, que les cas de test créés au hasard peuvent manquer, alors qu'un jeu de tests couvrant toutes les combinaisons possibles serait trop grand et impossible à exécuter. Dans cette thèse, nous abordons toutes ces problématiques en proposant plusieurs approches : 1) une approche qui conduit les CBSs dans une suite de comportements élastiques prédéfinis, 2) une approche qui synchronise l'exécution des tests selon les états du CBS, 3) une approche qui permette la reproduction des tests pendant l'élasticité, 4) un langage spécifique à ce domaine (DSL, selon l'acronyme anglais) qui résume la mise en œuvre des test pendant l'élasticité, 5) une approche qui génère des petits ensembles de tests pendant l'élasticité tout en révélant des problèmes liés à l'élasticité.

Mots clés
Cloud Computing, Élasticité, Test Pendant l'Elasticité, Contrôlabilité.

Abstract
Systems deployed on elastic infrastructures deal with resource variations by adapting themselves, which may cause errors, or even degrade their performance. Therefore, we must test the Cloud-Based Systems (CBSs) throughout elasticity, which faces problematics. First, executing elasticity tests may require the lead of CBS throughout a specific elastic behavior, i. e., sequence of resource changes, which depends on an accurate workload generation. Second, CBS adaptations occur at a precise moment, such as after a resource scale out, which requires to test them during a specific CBS states. Third, testers must re-execute elasticity tests in a deterministic manner to debug and fix the CBS. Fourth, implementing elasticity tests is complex and laborious given the wide possibility of parameters, and the peculiarity of cloud computing. Finally, specific combinations of parameters may cause the system issues, where random tests may miss such combinations, while a test set that covers all the combinations may be large and impractical to execute. In this thesis, we tackle all these five problematics by proposing several approaches: 1) an approach to drive the CBS throughout preset elastic behaviors, 2) an approach to synchronize tests according to the CBS states, 3) an approach to enable reproducing elasticity testing, 4) a Domain Specific Language (DSL)-based approach to abstract the elasticity testing implementation, and 5) an approach to generate small sets of tests that reveal relevant elasticity-related issues.

Key Words
Cloud Computing, Elasticity, Elasticity Testing, Controllability.