Planification pour la gestion autonomique de l'élasticité d'applications dans le cloud

Loic Letondeur

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 Août 2006

Présentée par

Loïc LETONDEUR

Thèse dirigée par **M. Noël DE PALMA**
et codirigée par **Mme Fabienne BOYER et M. Thierry COUPAYE**

préparée au sein **d'Orange Labs et du Laboratoire d'Informatique de Grenoble**
et de **L'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Planification pour la gestion autonomique de l'élasticité d'applications dans le cloud

Thèse soutenue publiquement le ,
devant le jury composé de :

M. Daniel HAGIMONT
Professeur à l'Institut National Polytechnique de Toulouse, Rapporteur
M. Lionel SEINTURIER
Professeur à l'Université Lille 1, Rapporteur
M. Frédéric DESPREZ
Directeur de Recherche à l'INRIA, Examinateur
M. Alain TCHANA
Maître de Conférences à l'Institut National Polytechnique de Toulouse, Examinateur
M. Noël DE PALMA
Professeur à l'Université Joseph Fourier, Directeur de thèse
Mme Fabienne BOYER
Maître de Conférences à l'Université Joseph Fourier, Co-Directeur de thèse
M. Thierry COUPAYE
Directeur de Recherche Cloud Platforms chez Orange Labs, Co-Directeur de thèse
M. Xavier ETCHEVERS
Ingénieur de Recherche chez Orange Labs, Co-Directeur de thèse

Remerciements

Au terme de ce long parcours qu'est le doctorat, je tiens à adresser mes remerciements à l'ensemble des personnes qui par leurs contributions ont permis l'aboutissement de ce travail.

En tout premier lieu, je souhaite remercier l'ensemble des membres du jury qui m'ont fait l'honneur de participer à ma soutenance. Merci à M. Frédéric Desprez, directeur de recherche à l'INRIA, pour avoir assuré la présidence du jury. Merci également à M. Daniel Hagimont, professeur à l'Institut National Polytechnique de Toulouse, et M. Lionel Seinturier, professeur à l'Université de Lille, pour avoir accepté d'être les rapporteurs de mon travail en pleine période de vacances d'été et avoir apporté un jugement constructif. Je remercie également M. Alain Tchana, maître de conférences à l'Institut National Polytechnique de Toulouse pour avoir accepté d'être examinateur de cette thèse.

Je tiens également à remercier mes quatre encadrants de thèse pour la confiance qu'ils m'ont témoignée, pour leur soutien, ainsi que pour le climat de réflexion et la méthodologie dont ils ont su me faire bénéficier. Merci à M. Noël de Palma, professeur à l'Université Joseph Fourier, à Mme Fabienne Boyer, maître de conférences à l'Université Joseph Fourier, à M. Thierry Coupaye, directeur de recherche chez Orange Labs et à M. Xavier Etchevers, ingénieur de recherche chez Orange Labs.

J'adresse également de profonds remerciements à M. Alexandre Lefebvre pour sa contribution au sein de l'encadrement de ce doctorat et qui a largement participé à son évolution.

Je n'oublie pas toutes les personnes d'Orange Labs qui m'ont soutenu et aidé dans ce travail.

Enfin, je tiens à adresser un remerciement tout particulier à ma douce et tendre Mélanie. Merci pour sa compréhension à l'égard d'un compagnon très souvent fatigué, stressé et pensif en raison du rythme poussé de travail induit par le doctorat. Merci pour son soutien dans les moments difficiles et les instants de doute. Sa gentillesse, sa tendresse et son humour ont joué un rôle déterminant dans l'aboutissement de ce travail.

Résumé

Le Cloud Computing permet une optimisation des coûts de déploiement et de maintenance des applications. Grâce au cloud, chaque application peut être déployée et reconfigurée en l'espace de quelques minutes. La nécessité pour une application d'être en permanence à la juste taille afin que celle-ci maintienne continuellement une qualité de service suffisante, sans pour autant utiliser trop de ressources, requiert de mettre en place l'élasticité des applications dans le cloud. Cependant, l'état de l'art montre que les solutions actuelles de gestion de l'élasticité sont restreintes à des applications multi-tiers tout en limitant fortement les scénarios possibles. Le cloud est effectivement un paradigme jeune et les différents acteurs du marché ont rapidement cherché à offrir l'élasticité. Si l'élasticité ainsi proposée présente l'avantage indéniable d'être simple à mettre œuvre, celle-ci n'obéit pas aux besoins de nombreux utilisateurs.

Afin de repousser les limites actuelles de l'élasticité, les travaux présentés dans ce manuscrit proposent un outil de spécification de l'élasticité nommé Vulcan. A la fois simple et complet, Vulcan montre qu'il n'existe pas de corrélation entre la complétude de la solution et la complexité de son utilisation. Basé sur une approche autonomique, Vulcan apporte les contributions suivantes :

- un modèle d'applications élastiques : celui-ci permet de décrire selon un mode dit *par intention*, la façon dont l'application doit évoluer au cours des reconfigurations induites par l'élasticité. Le formalisme de ce modèle est à la base de la simplicité recherchée quant à l'utilisation de la solution.

- un algorithme de planification : cet algorithme permet la résolution des différents paramètres lors de l'élasticité tout en offrant une compréhension simple à l'utilisateur de Vulcan.

- un prototype qui met en œuvre l'ensemble des concepts mis en avant dans ce document.

Reposant sur des principes à la fois novateurs mais également issus de solutions éprouvées, des expérimentations ont démontré tant les capacités d'intégration de Vulcan vis-à-vis de l'existant, que la viabilité des concepts proposés ou le potentiel de l'approche. Il en ressort une solution générique, sans restriction quant aux applications gérées et qui repousse les limites actuelles en matière d'élasticité.

Mots-clés : informatique dans le nuage, élasticité, systèmes répartis, applications patrimoniales, planification, informatique autonome, modèle à composants.

Abstract

Cloud computing permits cost optimization of both deployment and maintenance applications. Thanks to the cloud, applications can be deployed and reconfigured in a few minutes. Each application can thus be continuously maintained at a fair size, so that it can continuously maintain a sufficient quality of service without using too many resources. This adaptation is achieved thanks to the feature named *elasticity* . However, the state of the art shows that current solutions for the management of elasticity are restricted to multi-tiers applications and do not manage all possible scenarios. The cloud is actually a young paradigm and the various market providers have managed to offer elasticity. If the provided elasticity has the undeniable advantage of being a fast and easy mean to manage basic cases, it does not address the needs of many users.

To tackle the current limits of elasticity, the work presented in this manuscript propose a tool for the specification of elasticity that is named Vulcan. Besides being simple and complete, Vulcan shows that there is no correlation between the completeness of a solution for the management of elasticity and the complexity of its use. Based on an autonomous approach, Vulcan brings the following contributions :

- a model for elastic applications : it is used to describe how an application should evolve over the reconfigurations induced during elasticity. This description is done at a high level said *by intension* thanks to an innovative formalism. The proposed formalism is the basis of the Vulcan simplicity of use.

- a scheduling algorithm : this algorithm resolves various parameters during elasticity while providing to the user of Vulcan an easy comprehension.

- a prototype that implements all concepts described in this manuscript.

Making use of both innovative concepts and principles from proven solutions, Vulcan has shown its capabilities to push the current limits of elasticity.

Keywords: cloud computing, elasticity, distributed systems, legacy applications, autonomic computing, component-based models.

Table des matières

Table des figures

Liste des tableaux

Listings

Chapitre 1

Introduction

Sommaire

Les travaux de thèse présentés dans ce manuscrit se situent dans le domaine du *Cloud Computing* [40, 105] (l'informatique dans le nuage), un nouveau paradigme d'accès à des infrastructures informatiques. Le cloud met à disposition de ses utilisateurs des ressources généralement virtualisées, capables d'effectuer des calculs, de stocker des données ou de faire transiter des informations. Ses ressources sont accessibles sous la forme de services et au travers du réseau, généralement internet. Leur utilisation est facturée suivant la formule "*Pay-as-you-go*" : l'utlisateur paie ni plus ni moins que ce qu'il consomme. Un avantage extraordinaire du cloud computing réside dans l'approvisionnement des services puisque celui-ci est *à la demande* : le client obtient les services qu'il demande en fonction de ses propres besoins. Ces besoins sont quant à eux dictés par ceux de son application, c'est-à-dire le programme qu'il exécute dans le cloud.

Le cloud computing présente plusieurs avantages majeurs comparativement aux infrastructures matérielles traditionnelles. En effet, avec les infrastructures traditionnelles, le temps d'approvisionnement de machines était de l'ordre de plusieurs semaines/mois là où le cloud fournit un service équivalent en seulement quelques minutes. Outre la rapidité, les démarches nécessaires sont également considérablement simplifiées : l'approvisionnement de ressources dans le cloud peut se faire au travers d'une simple interface web là où avec les infrastructures traditionnelles des formalités administratives devaient également être remplies (e.g. obtention d'un budget Capex suffisant, accord de la hiérarchie, appel d'offre, bon de commande). Le cloud permet donc une fourniture rapide et simple d'infrastructures informatiques. Outre la rapidité et la simplicité, le cloud permet aussi de maintenir une infrastructure à la taille nécessaire, à chaque instant, là où les infrastructures matérielles traditionnelles étaient marquées soit par un gaspillage constant de ressources, soit par une insuffisance des ressources allouées. En effet, face à la complexité

des démarches d'obtention de moyens matériels, le choix était souvent fait de surdimensionner les ressources de sorte à pouvoir absorber quelques rares pics de charge. Ce choix résultait dans une sous-utilisation quasi permanente des ressources matérielles en dehors des pics tout en ne garantissant même pas que ces fameux pics soient correctement gérés quand ils se produisaient. Le cloud promet quant à lui de pouvoir adapter dynamiquement les ressources consommées : il devient ainsi possible de non seulement absorber correctement les pics de charge en un temps très court, mais aussi d'éviter le gaspillage en ne consommant que les ressources requises. Cette notion d'adaptation dynamique de la consommation des services du cloud est connue sous le terme d'*élasticité*, une des promesses du Cloud Computing qui est au centre des préoccupations des travaux de thèse présentés.

1.1 Problématique

Face à l'intérêt manifesté par les utilisateurs, l'élasticité est très rapidement devenue un buzzword que toutes les plateformes de cloud se doivent d'afficher parmi leurs fonctionnalités. Il s'agit également d'un sujet de recherche très en vogue face à l'intérêt suscité mais également en raison des problématiques à adresser.

Au niveau industriel, la gestion de l'élasticité a vu naître un ensemble important de solutions très souvent propriétaires (e.g. Amazon CloudFormation, Microsoft Azure, Google AppEngine). Ces solutions permettent une gestion automatique de l'élasticité. Elles sont néanmoins marquées à la fois par des risques d'enfermement propriétaire mais aussi par des limites importantes qui empêchent un utilisateur d'avoir une maîtrise complète sur l'élasticité de son application. En outre, ces solutions ne gèrent bien souvent qu'un ensemble restreint d'applications.

Au niveau des solutions de recherche, celles-ci se focalisent dans l'ensemble, sur des problématiques de prises de décision relatives à des méta-modifications à opérer sur l'application. Ces décisions visent à adapter dynamiquement le profil de l'application et consistent en des ordres d'ajout, de suppression ou de reconfiguration de ressources. De nombreux travaux ont ainsi vu le jour, apportant des solutions capables de décider des méta-modifications durant l'élasticité. Ces solutions permettent donc d'adapter une application en fonction de sa charge. Néanmoins, si ces travaux sont absolument essentiels, ils ne suffisent pas à garantir que l'élasticité ne soit pas restreinte par manque de contextualisation. Pour comprendre ce point, il suffit de considérer par analogie l'expérience de la CIA nommée "chaton acoustique" (*Acoustic Kitty*). En pleine Guerre Froide, à la recherche de nouveaux moyens pour espionner le Kremlin et ses ambassades, la CIA a mené dans les années 1960 un programme consistant à équiper un chat d'un micro, d'une batterie et d'un émetteur. L'antenne était alors implantée par acte chirurgical le long de la colonne vertébrale de l'animal. La CIA avait même réussi à lui implanter un récepteur afin de lui donner des ordres de déplacement à distance. Coûtant des millions de dollars, ce programme fut un échec puisque dès le premier essai sur le terrain, l'expérience s'acheva par le décès tragique du chat. En effet, l'expérience s'est déroulée près d'un parc à Washington DC et pour y parvenir le chat devait traverser la route afin

d'obéir à l'ordre lui intimant d'aller tout droit. Malheureusement pour le pauvre animal, son aptitude à traverser une route avec des voitures n'avait pas été testée et celui-ci périt, heurté par un taxi. Au-delà de l'aspect tragique de la scène, cet exemple montre l'importance que revêt le fait de contextualiser des ordres. De même que l'ordre d'aller tout droit pour le chat aurait dû être contextualisé afin de tenir compte de la route, les décisions de méta-modifications pour l'élasticité doivent également l'être afin de tenir compte des limites et des possibilités relatives à un environnement. Les contributions des travaux de thèse exposés dans ce manuscrit consistent en une solution capable de lever les verrous actuels de l'élasticité en permettant une contextualisation intelligente des décisions d'élasticité.

La solution présentée permet d'adapter les décisions d'élasticité à l'environnement courant de sorte à établir un plan complet et pertinent des modifications de l'application. En rapport avec la précédente analogie, l'ordre "va tout droit" serait ainsi décliné en opérations élémentaires successives telles que : "cherche le passage piétons le plus proche", puis "attends le feu rouge", puis "traverse", puis "gagne le point à atteindre". Il s'agit là d'un scénario un peu surréaliste mais qui illustre bien la complexité et l'intérêt de cette problématique.

Les travaux de thèse doivent remplir les objectifs suivants :

1. Définir un modèle architectural et un formalisme permettant la représentation et la spécification d'applications élastiques déployées dans le cloud. Il s'agit de proposer un modèle qui permette à un utilisateur de décrire non seulement son application mais également les contraintes qui en régissent le comportement durant l'élasticité. Cette application étant élastique, elle est amenée à évoluer, à être reconfigurée : il faut donc offrir une maîtrise complète sur les différentes évolutions avec le moins de restrictions possibles provenant de leur expression.

2. Concevoir et prototyper les mécanismes permettant l'élasticité. Cet objectif vise à implanter un prototype à même de gérer l'élasticité d'une application tout en tenant compte du modèle issu du premier objectif. Ce prototype doit permettre des évolutions de l'application qui soient intégralement maîtrisées par l'utilisateur. Cette maîtrise se doit d'être simple mais ne doit également pas se faire au prix de restrictions quant à la couverture des évolutions possibles d'une application.

3. Valider l'approche par des cas d'usages réalistes. Pour remplir cet objectif, des applications doivent être identifiées. Chacune de ces applications doit correspondre à de vrais besoins utilisateurs. Pour l'ensemble de ces applications, des évolutions possibles durant l'élasticité doivent être identifiées et examinées. Chacune de ces évolutions correspond à un scénario qui doit justifier de son réalisme.

.

1.2 Contributions et Plan

Les contributions des travaux présentés sont au nombre de trois :

1. Un modèle d'applications élastiques. Celui-ci comprend deux modèles complémentaires l'un de l'autre. Le *Modèle par Extension* est une connaissance de l'état courant de l'application tandis que le *Modèle par Intention* permet la description de contraintes servant à contextualiser les décisions d'élasticité. Le *Modèle par Intention* fait usage d'un formalisme original et innovant.

2. Un algorithme permettant la réalisation de la contextualisation des décisions d'élasticité. Cet algorithme fait appel à des méthodes issues de l'intelligence artificielle et permet une gestion aisée et compréhensible de l'élasticité.

3. Un prototype implantant l'ensemble des concepts. Une évaluation permettant de valider l'approche en est issue. Celle-ci repose sur un sous-ensemble des cas d'usages réalistes qui ont été identifiés.

Les travaux présentés apportent une gestion de l'élasticité clairement novatrice en comparaison de l'état de l'art actuel. Ces travaux ont fait l'objet de deux diffusions scientifiques :

- L. Letondeur, X. Etchevers, T. Coupaye, F. Boyer and N. De Palma. Planification pour la gestion autonomique de l'élasticité d'applications dans le cloud. Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS 2014), Neuchâtel, Suisse, 2014.

- L. Letondeur, X. Etchevers, T. Coupaye, F. Boyer and N.De Palma. Architectural Model and Planification Algorithm for the Self-Management of Elastic Cloud Applications. IEEE International Conference on Cloud and Autonomic Computing (CAC 2014), Londres, Royaume-Uni, 2014.

De plus, une participation au projet collaboratif FSN OpenCloudWare[1] a été réalisée au travers de plusieurs présentations dont une en réunion plénière.

La présentation des travaux est réalisée dans ce document en trois parties successives.

1. La première partie consiste en la présentation de l'état de l'art actuel. Pour cela, dans le chapitre 2, une description du contexte est donnée en préambule d'une étude approfondie de solutions de gestion de l'élasticité dans le chapitre 3.

   Au cours du chapitre 2, une description de l'élasticité des applications est tout d'abord donnée avant une description du Cloud Computing. Cette seconde description vise à montrer à quel point cet environnement est changeant, ce qui influe directement sur les contributions de la thèse.

   Le Chapitre 3 propose l'étude d'un ensemble de solutions de gestion de l'élasticité. Au cours de cet état de l'art, chaque solution est présentée puis évaluée suivant une grille de critères. Ces solutions concernent à la fois des solutions de recherche et des solutions industrielles.

[1] http ://www.opencloudware.org

2. Une deuxième partie expose les contributions de la thèse et en détaille les concepts les plus importants. Le chapitre 4 expose une vue d'ensemble des contributions. Les chapitres suivants détaillent ces contributions en expliquant les problématiques que chacune adresse et l'approche retenue pour le faire. Le chapitre 5 détaille le *Modèle par Extension*, un modèle visant à offrir la connaissance de l'état de l'application de sorte à pouvoir en calculer les évolutions durant l'élasticité. Ces calculs sont guidés par les contraintes que l'utilisateur mentionne dans le *Modèle par Intention* que le chapitre 6 décrit. Ce chapitre présente notamment le formalisme original et innovant retenu. Le chapitre 7 aborde une contribution majeure de la solution proposée. Il s'agit d'un algorithme permettant une gestion avancée des calculs mais qui rend toutefois l'utilisation de la solution extrêmement simple et pratique. En fin de ce chapitre, une synthèse rappelle les concepts de la solution et en expose d'autres montrant que la solution proposée s'appuie sur des préoccupations plus étendues.

3. Une troisième partie présente l'implantation du prototype et les expérimentations qui ont permis de valider l'approche proposée. Pour cela, le chapitre 8 commence cette partie en décrivant l'implantation du prototype qui a été réalisé. Il explique et justifie les choix importants qui ont été faits. Ensuite, le chapitre 9 propose une évaluation qualitative de ce prototype au travers de l'identification de cas d'usages réalistes et positionne les travaux par rapport à l'existant. Cette partie s'achève par le chapitre 10 qui propose une évaluation des performances du prototype.

Enfin, une conclusion rappelle les objectifs de la thèse, les contributions et expose des perspectives.

Première partie

État de l'art

Chapitre 2

Contexte

Sommaire

Ce chapitre vise à offrir au lecteur une vision à la fois claire et concise du contexte de la thèse qui porte sur l'élasticité des applications dans le cloud. Pour cela, ce chapitre commence par décrire dans une première section les applications et leurs architectures de sorte à proposer un contexte minimal servant de base à la définition de l'élasticité. Dans une deuxième section, ce chapitre aborde le *Cloud Computing*, un domaine dans lequel l'élasticité est simplifiée mais également soumise à des spécificités.

2.1 Elasticité des applications

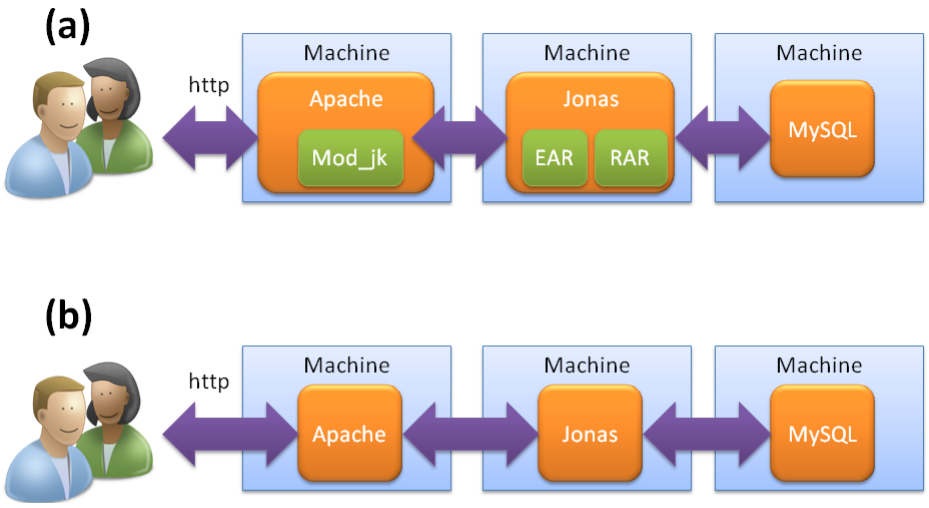
Cette section aborde la thématique de l'élasticité des applications. Pour cela, elle commence par exposer les concepts de base nécessaire à la compréhension de l'élasticité : les applications et les architectures applicatives. Cette première section vise à décrire la problématique de l'élasticité par une représentation à la fois compréhensible et intuitive.

2.1.1 Définition des applications et des architectures applicatives

Une application se définit comme un ensemble coopératif de briques logicielles pouvant être exécutées sur une ou plusieurs machines. Chaque brique logicielle est en charge de

fournir des fonctionnalités, soit directement à l'application, soit à destination d'autres briques logicielles servant l'application. Chacune de ces briques logicielles requiert des ressources en termes de capacités de calcul (CPU, GPU, nombre de threads), de stockage (disque dur, SSD), et de réseau.

L'architecture d'une application se définit quant à elle par l'ensemble des briques logicielles d'une application et des interactions existant entre ces briques et/ou avec le "monde extérieur". Il s'agit typiquement de canaux de communication et d'échanges de configurations. Le concept d'*architecture à base de composants* [146] modélise les applications avec des composants pouvant être soit des *composants primitifs* soit des *composants composites*. De façon générale, un composant est en fait une abstraction à granularité libre d'une ou plusieurs briques constitutives d'une application. La figure 2.1 donne un exemple d'architecture à base de composant pour Springoo, une application web JavaEE représentative de 80% du Système d'Information (SI) d'Orange.



Springoo est un application web de gestion de catalogues constituée de trois tiers : présentation, métier et persistance. (a) et (b) sont deux architectures présentant la même application avec des granularités différentes.

FIGURE 2.1 – Présentation de l'application Springoo

Cette application est constituée de trois tiers : un tiers présentation, un tiers métier et un tiers de persistance. Les parties (a) et (b) de la figure 2.1 représentent deux architectures de Springoo mais avec une granularité différente. Le tiers présentation est constitué d'un serveur Apache représenté par un composant *Apache*, le tiers métier comprend un serveur Jonas représenté par un composant *Jonas* et le tiers persistance un serveur de base de données MySQL représenté par un composant *MySQL*. La différence de granularité entre les deux parties de la représentation est la suivante : la partie (a) mentionne des composant primitifs *Mod_jk*, *EAR* et *RAR* alors qu'ils sont absents de la partie (b). Dans la partie (a), les serveurs Apache et Jonas sont ainsi représentés par des

composants composites puisqu'ils contiennent des composants primitifs, tandis que dans la partie (b), ces mêmes serveurs sont représentés par des composants primitifs. Comme le montre cet exemple, les architectures à base de composants permettent de modéliser une application suivant différentes granularités. Il aurait par exemple été possible de représenter les ressources matérielles au travers de composants. Cela souligne l'adaptabilité des architectures à base composants. Dans la suite de ce manuscrit, les exemples donnés s'appuient sur la granularité de la partie (b) de la figure 2.1, le terme *composant* faisant référence à des briques logicielles auxquelles des *ressources* sont allouées. Les *ressources* sont décrites par des paramètres des *composants* qu'ils soient primitifs ou composites.

2.1.2 Définition de l'élasticité

Le début de cette section a exposé les concepts permettant de poser les bases nécessaires à la définition l'élasticité que cette deuxième sous-section expose et illustre au travers de l'application Springoo.

La définition de l'élasticité qui est fournie dans cette sous-section est donnée du point de vue de l'architecture de l'application. C'est du reste ce point de vue qui est utilisé dans la suite du manuscrit. Ce point de vue est toutefois complété par une explicitation concrète.

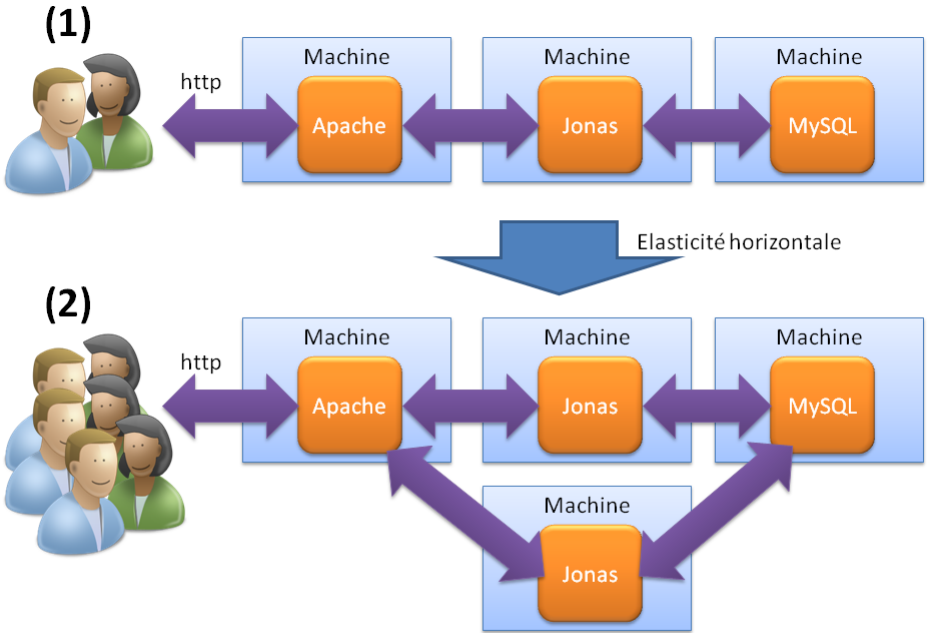
L'élasticité se définit comme un ensemble d'opérations venant modifier l'architecture d'une application par des ajouts/suppressions/reconfigurations de composants à des fin d'adaptation de l'application à la charge.

De façon concrète, au cours de l'élasticité, de nouvelles briques logicielles peuvent être ajoutées à une application de sorte à, par exemple, absorber une charge plus importante tout en maintenant de bonnes performances. De même, des ressources peuvent être ajoutées, soit pour permettre l'exécution de nouvelles briques logicielles, soit pour améliorer les performances de celles déjà présentes. Il s'agit par exemple d'ajouter de la mémoire vive et/ou des cœurs CPU alloués à une brique logicielle lorsque celle-ci parvient à saturer ses ressources. A contrario, l'élasticité concerne aussi la désallocation de briques logicielles et la libération de ressources matérielles. Une désallocation se traduit par la suppression d'une brique logicielle lorsque la charge de l'application est faible, tandis que la libération de ressources correspond à la fin de l'utilisation de ces mêmes ressources. La libération de ressources peut par exemple être réalisée lorsqu'une brique logicielle est désallouée ou lorsque la charge de l'une d'elle au cours de son exécution est faible.

L'élasticité comprend donc des opérations à la hausse comme à la baisse qui impactent les briques logicielles et/ou leurs ressources. En ce sens, deux types d'élasticité sont exposés dans la littérature scientifique : *l'élasticité horizontale* et *l'élasticité verticale*. Par définition, l'élasticité horizontale consiste en l'ajout ou le retrait de briques logicielles tandis que l'élasticité verticale ne modifie pas le nombre de composants mais uniquement leurs allocations de ressources. Un cas pratique d'opération d'élasticité verticale consiste en l'ajout de ressources matérielles supplémentaires : un serveur Jonas peut ainsi voir sa quantité de mémoire être augmentée en l'espace de quelques secondes. L'élasticité verticale est ainsi particulièrement réaliste en raison de sa rapidité d'exé-

cution par rapport au temps de nécessaire pour l'élasticité horizontale qui elle requiert plusieurs dizaines de secondes.

D'un point de vue architectural, l'élasticité horizontale ajoute ou retire des composants tandis que l'élasticité verticale modifie la configuration des composants-composites. La figure 2.2 illustre un cas d'élasticité horizontale pour l'application Springoo tandis que la figure 2.3 montre une opération d'élasticité verticale.

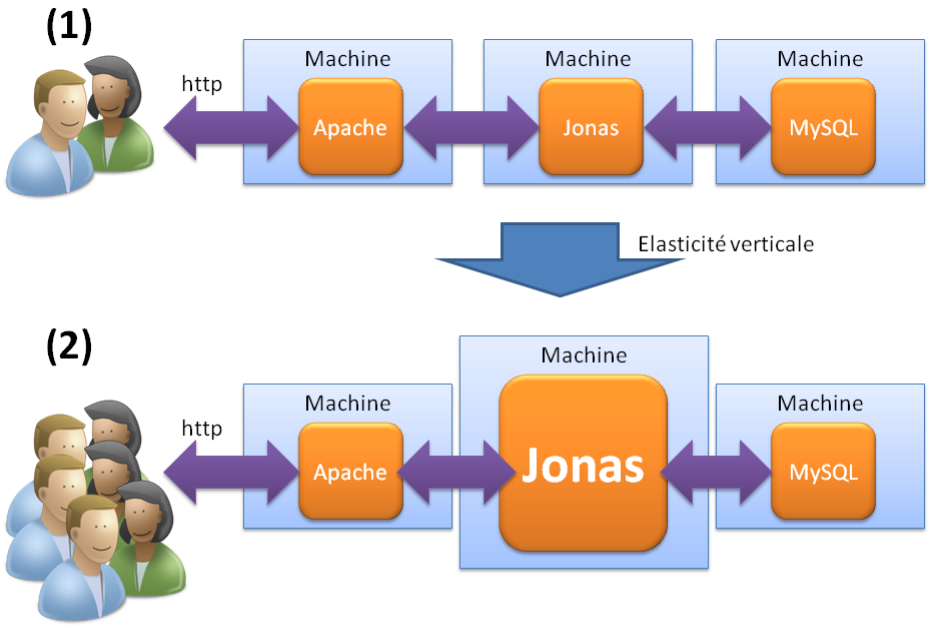


(1) est la situation intiale. En (2) le nombre de serveurs a augmenté : un nouveau serveur a été ajouté au tiers métier pour faire face à l'augmentation du nombre d'utilisateurs. Un équilibrage de charge est opéré sur le tiers présentation entre les deux serveurs Jonas.

FIGURE 2.2 – Exemple d'une opération d'*élasticité horizontale* pour l'application Springoo

Dans la suite de ce document, la terminologie des opérations d'élasticité[108, 155, 21, 9] est définie de la façon suivante :

- ***croissance verticale*** (scale up) : opération d'élasticité verticale consistant à ajouter des ressources à des briques logicielles. D'un point de vue d'une architecture applicative, il s'agit de reconfigurer à la hausse les composants d'une application sans qu'aucun composant ne soit ni ajouté ni retiré. Par exemple, une telle opération peut résulter en l'ajout de mémoire vive à une application en cours d'exécution.
- ***décroissance verticale*** (scale down) : opération d'élasticité verticale inverse de la *croissance verticale*.
- ***croissance horizontale*** (scale out) : opération d'élasticité horizontale qui consiste à ajouter de nouvelles briques logicielles à une application. Ce type d'opération



De (1) vers (2), le nombre de serveurs n'a pas évolué. Seules les ressources matérielles allouées au tiers métier (i.e. le serveur Jonas) ont été augmentées pour soutenir la charge.

FIGURE 2.3 – Exemple d'une opération d'*élasticité verticale* pour l'application Springoo

provoque donc l'ajout de composants dans l'architecture de l'application.

- ***décroissance horizontale*** (scale in) : opération d'élasticité horizontale inverse de la *croissance horizontale*.

Dans la suite de ce manuscrit, **l'élasticité est abordée du point de vue de l'architecture des applications élastiques**. Toutefois, toutes les applications ne permettent pas la réalisation de toutes les opérations d'élasticité. La suite de cette section aborde ce point.

2.1.3 Capacités d'élasticité des applications

Si l'élasticité ouvre de formidables horizons en matière d'adaptation à la charge pour les applications, la pratique montre qu'il ne s'agit toutefois pas d'un "remède miracle". En effet, toutes les applications ne sont pas aptes à être élastifiées. Il est ainsi courant de rencontrer des administrateurs voulant élastifier une application. Pensant détenir une pierre philosophale à même de répondre à tous leurs tracas, ceux-ci ne se rendent malheureusement pas compte qu'une application voit ses capacités élastiques être limitées par sa propre conception.

Pour comprendre ce point il est nécessaire de considérer l'informatique "avant le cloud". Comme le souligne [54], avant le cloud, l'informatique avait recours à des infrastructures physiques disponibles ça et là dans des centres de données de plus ou moins

petite échelle. Ces infrastructures physiques étaient administrées de façon locale, à des coûts élevés et étaient caractérisées par une faible autonomie nécessitant presque systématiquement le déplacement physique d'un administrateur. En plus d'être onéreuses, les infrastructures physiques traditionnelles n'étaient administrées ni de façon réellement automatisée ni de façon réactive. En conséquence, nombre d'applications destinées à ces infrastructures étaient conçues en tenant compte de ces limites et ne sont par exemple pas administrables de façon automatisée. Par ailleurs, comme les infrastructures traditionnelles étaient lentes à évoluer (plusieurs semaines pour obtenir une nouvelle machine), nombre d'applications n'étaient pas aptes à être un jour élastifiées, dans le sens où ni l'ajout ni le retrait de composants n'étaient possibles sans nécessiter un redémarrage. Pour aller plus loin, certaines applications requérant des matériels spécifiques ne sont carrément pas portables dans le cloud. Leur élasticité revient à manipuler des ressources matérielles spécifiques, ce qui n'entre clairement pas dans le cadre des travaux de thèse. Les travaux de thèse présentés dans ce manuscrit adressent donc les applications portables dans le cloud. De plus, tout comme une file de voitures voit sa vitesse être limitée par celle roulant le plus doucement, l'élasticité offerte par la solution proposée est bien évidemment limitée par les capacités de l'application. Dans le tableau 2.1 nous proposons une caractérisation des applications en fonction de leurs aptitudes face à l'élasticité.

Elasticité verticale	Elasticité horizontale	Caractéristiques observables de l'application
Oui	**Oui**	L'application est à même de gérer tout type d'opération élastique. Ses performances augmentent avec le nombre de composants. Individuellement, chaque composant est à même de tirer parti d'une augmentation des ressources qui lui sont allouées (e.g. CPU, RAM, nombre de threads)
Non	**Oui**	L'application permet d'ajouter ou de retirer des composants. Ces opérations influent clairement sur les performances de celle-ci tandis que les opérations de (dé)croissance verticale n'ont qu'un faible impact. Ceci est par exemple le cas des applications de minage de crypto-monnaies sur GPU pour lesquelles l'ajout de RAM ou de CPU n'a presque aucune influence.
Oui	Non	L'application ne permet ni d'ajouter, ni de retirer des composants que ce soit de façon dynamique ou non. En revanche, une opération de croissance verticale lui apporte une hausse de performances et réciproquement, une décroissance verticale effectuée dans des conditions satisfaisant son fonctionnement s'accompagne d'une baisse des performances.
Non	Non	L'application ne peut être pas élastifiée en raison d'une conception ne permettant ni de dupliquer des composants ni même de tirer partie d'une élasticité verticale à la volée (i.e. augmentation des ressources allouées sans redémarrage)

Ce tableau donne les caractéristiques observables des applications en fonction de leurs aptitudes élastiques. Une application ne permettant ni élasticité verticale, ni élasticité horizontale ne pourra pas devenir élastique sans une refonte de son architecture et/ou de son code.

TABLE 2.1 – Caractérisation des applications concernant leur gestion de l'élasticité

Ainsi l'élasticité est limitée par la conception des applications. Elle l'est également dans une infrastructure physique traditionnelle en raison du manque d'automatisation et de rapidité. La section suivante présente le cloud computing, un environnement qui permet d'élargir les possibilités de gestion de l'élasticité.

2.2 Cloud Computing

Cette section continue de présenter le contexte des travaux de thèse et se focalise sur le *Cloud Computing*, un paradigme jeune et en plein essor. Le but de cette section est d'offrir une compréhension complémentaire à celle des présentations usuelles puisque le *Cloud Computing* est sujet à une évolution rapide et permanente. Ce paradigme ne se résume pas à un simple modèle en couches ni à une façon d'accéder à des ressources et services. S'il est tout-à-fait possible d'établir "une photo" actuellement, tout indique que celle-ci ne sera pas valable très longtemps. Par analogie, si l'on considère la théorie du Big Bang, l'univers à T0+8 ans était radicalement différent à T0+10 ans. Le cloud computing peut ainsi être vu comme un écosystème darwinien complexe et en perpétuel changement : des "espèces" s'éteignent faute d'adaptations, là ou d'autres apparaissent voire prennent l'ascendance. Cette section décrit donc le *Cloud Computing* au travers de sa définition communément admise tout en l'augmentant de changements possibles. Ces deux aspects sont effectivement au cœur des préoccupations adressées par les contributions exposées dans la suite du manuscrit.

2.2.1 Définition usuelle du cloud

Les difficultés et les coûts liés au déploiement et à l'administration des applications dans les infrastructures physiques traditionnelles ont motivé l'émergence d'un nouveau paradigme : le *Cloud Computing*. Le cloud computing se caractérise par un ensemble infini de ressources accessibles au travers d'internet et facturées à l'usage [26, 65, 105]. Il est communément décrit au travers d'un modèle en trois couches [26, 65, 105, 131, 137].

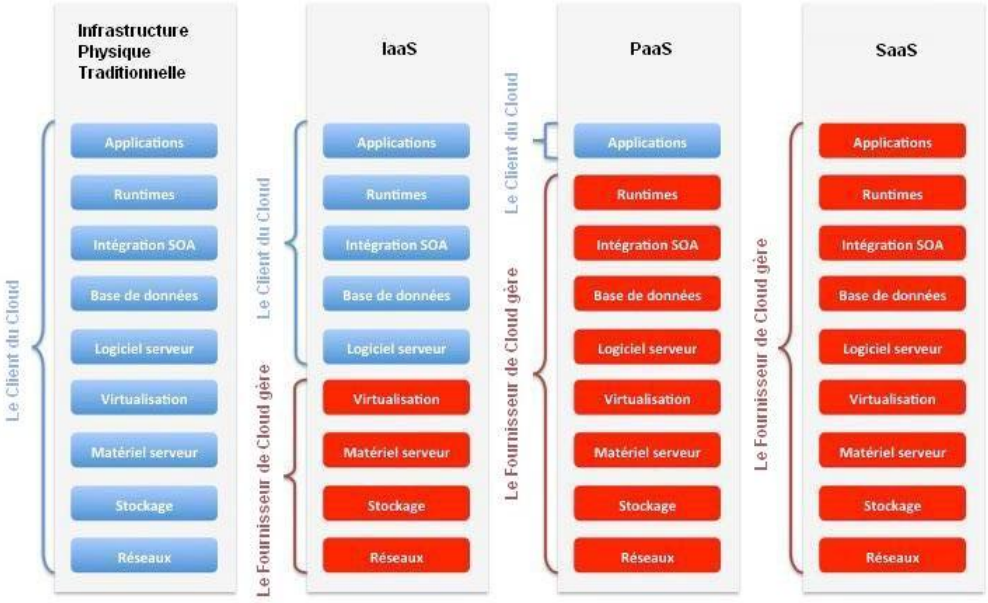
Un modèle en trois couches

Les ressources du cloud sont mises à disposition sous la forme de services qui peuvent être répartis en trois couches en fonction des préoccupations adressées :

- La couche *Infrastructure* constitue le plus bas niveau du modèle. Les services offerts concernent la fourniture de capacités de calcul, de stockage et de réseau. Il s'agit de préoccupations au niveau du matériel. Cette couche est généralement appelée *IaaS* pour *Infastructure as a Service*. Amazon EC2 [11] et OpenStack[10] sont, par exemple, des services de *IaaS*.

- La couche *Software as a Service* (*SaaS*) est la couche de plus haut niveau qui concerne des préoccupations visant à offrir des logiciels directement utilisables par les utilisateurs finaux. L'offre Office365 [13, 141] de Microsoft en est un exemple.

- La couche intermédiaire *PaaS* (*Platform as a Service*) est un liant entre la couche *Iaas* et la couche *SaaS*. Il s'agit d'offrir des intergiciels nécessitant les services de la couche IaaS pour permettre la réalisation des préoccupations de la couche SaaS comme Cloud Foundry [4] ou OpenShift [15] par exemple.



repris de : `http://www.technologies-ebusiness.com/enjeux-et-tendances/le-cloud-computing`

FIGURE 2.4 – Modèle en couches du cloud et préoccupations adressées par les différents acteurs

Le schéma 2.4 illustre ce modèle en trois couches ainsi que les préoccupations adressées. Ce schéma mentionne également les acteurs identifiés du cloud computing. Le *fournisseur de cloud* met à disposition des services tandis que les *clients du cloud* les consomment. Ces clients peuvent être des utilisateurs finaux (SaaS) et/ou des administrateurs d'applications (PaaS/IaaS) qui eux-mêmes déploient et gèrent des applications pour des utilisateurs finaux.

L'émergence du cloud computing a été étroitement liée au développement d'internet, ce qui permet à un client du cloud d'accéder simplement à n'importe quel cloud de par le monde. Outre les réseaux de télécommunications planétaires, le cloud computing repose également sur la virtualisation.

La virtualisation

La virtualisation est une technique permettant de reproduire l'intégralité du comportement d'une machine physique dans un environnement logiciel : il s'agit d'une machine dite virtuelle(VM) par opposition à une machine physique (PM). Avec une VM, un utilisateur a l'illusion de manipuler une machine physique alors que celle-ci n'est en réalité qu'une application intégralement logicielle ou presque. Chaque machine virtuelle exécute un système d'exploitation permettant à un utilisateur d'y héberger des logiciels : il s'agit d'un système d'exploitation qualifié d'*invité* pour le discerner de celui de la machine physique qui lui est qualifié d'*hôte*. Les VMs sont créées et gérées par des logiciels appelés

hyperviseurs. Les *hyperviseurs* ont recours à différentes techniques de virtualisation [123] ce qui influe sur certaines spécificités des VMs qu'ils créent :

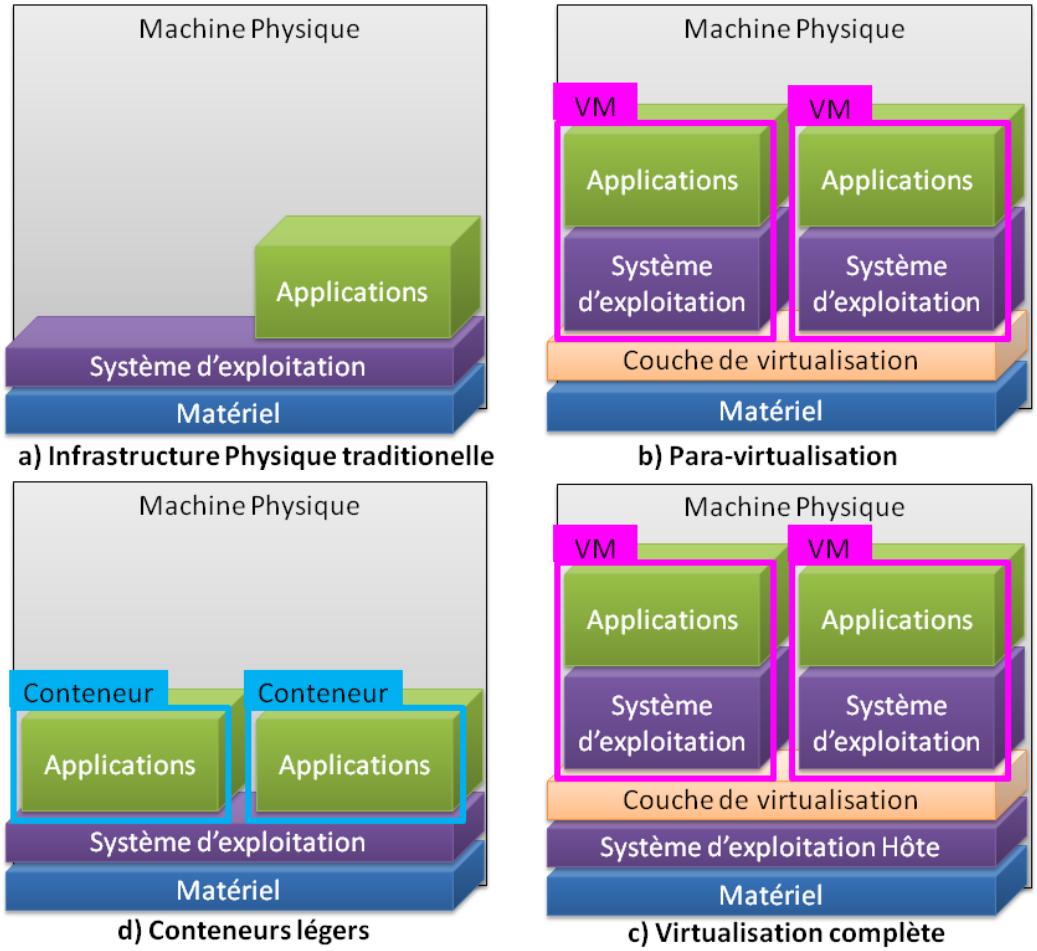
- La *para-virtualisation*. Il s'agit d'une couche de virtualisation directement installée sur du matériel. Chaque système d'exploitation invité a connaissance d'être une machine virtuelle. Cette technique est mieux adaptée aux systèmes d'exploitation libres puisqu'une modification est requise pour prendre en compte cette connaissance. Historiquement, Xen [28] est un hyperviseur utilisant cette technique. De base, la para-virtualisation offre d'excellentes performances par rapport à la *virtualisation complète*.

- La *virtualisation complète* permet l'exécution de VMs sans modification des systèmes d'exploitation invités. Toutefois, l'hyperviseur émule un processeur complet. Cette émulation étant particulièrement couteuse, il en résulte une forte dégradation des performances. VMWare Workstation et Virtualbox sont des exemples d'hyperviseurs utilisant cette technique.

Faces au succès de la virtualisation dans les serveurs, les fondeurs Intel et AMD ont apporté une amélioration matérielle des performances pour la virtualisation complète : on parle alors de *virtualisation complète assistée par le matériel*. Connues sous les noms d'*Intel-VT* et d'*AMD-V*, ces technologies ont permis de combler l'écart de performances entre la para-virtualisation et la virtualisation complète. L'ensemble des hyperviseurs actuels gèrent la virtualisation complète, dont Xen. La tendance actuelle en matière de virtualisation tend vers un déclin de la para-virtualisation au profit de la virtualisation complète.

La virtualisation permet donc d'obtenir des machines virtuelles *complètes* (également dites *lourdes*) par opposition à une autre approche appelée *conteneurs légers*. L'idée des *conteneurs légers* est non plus de créer une machine complète, mais de se baser sur un socle commun incluant un noyau et des bibliothèques, de sorte à avoir une approche plus simple, plus rapide et moins lourde par rapport aux VMs complètes [73]. En effet, les VMs complètes nécessitent de créer une image virtuelle, de paramétrer l'ensemble des caractéristiques techniques de la VM (nombre de cœurs CPU, quantité de RAM, réseau, disque durs, chipset, ...). Il s'agit d'un processus décrit comme étant complexe dans la littérature [58, 131]. Un conteneur léger est une sorte de *chroot* (i.e. un cloisonnement d'environnements d'exécution) plus évolué puisqu'il offre par exemple des liens réseaux séparés et un contrôle des ressources utilisées i.e. cpu, ram, disque). Un conteneur léger peut être déployé aussi bien sur des machines physiques que sur des machines virtuelles. Cette approche souffre toutefois d'une isolation moindre par rapport à des VMs lourdes. Des solutions comme l'univers Docker [17], Cloud Foundry [4] ou OpenShift [15] ont par exemple recours aux conteneurs légers.

La virtualisation et les conteneurs légers permettent donc l'exécution de multiples systèmes d'exploitation et environnements logiciels sur une même machine physique comme l'illustre la figure 2.5. Ce partage est à la base du Cloud Computing et plus particulièrement de ses apports. .



Comparaison de l'informatique sur infrastructure physique traditionnelle (a) avec les techniques de virtualisation (b et c) et les conteneurs légers (d)

FIGURE 2.5 – Virtualisation et conteneur légers

L'apport du cloud

Le cloud computing offre de nombreux avantages. Le premier d'entre eux est **la réduction des coûts**. Le partage des ressources matérielles permet, à partir d'une même ressource matérielle, de fournir des services à de multiples utilisateurs tout en les maintenant isolés : on parle de mutualisation des ressources. Ainsi, contrairement aux infrastructures physiques traditionnelles, les ressources physiques peuvent être concomittament partagées entre différents utilisateurs de façon sécurisée. Ce partage est rendu réaliste puisqu'en moyenne les serveurs physiques ne sont utilisés qu'entre 5 et 20% de leurs capacités [26]. De plus, un serveur sur six n'est tout simplement pas utilisé dans le monde. Le cloud permet donc une consolidation des serveurs tout en offrant un contrôle et des performances similaires à celles pouvant être obtenues au sein d'une infrastructure

physique traditionnelle. Pour un client, les coûts d'administration sont tirés vers le bas dans la mesure où la gestion du parc physique incombe au fournisseur de cloud.

Au-delà de la baisse des coûts, le cloud computing permet une **optimisation énergétique** [32, 53, 61, 162] : les machines physiques sont rassemblées en de grands ensembles et ne tournent plus à un niveau d'utilisation faible voire nul. Les grands ensembles ainsi créés permettent de tirer partie d'une meilleure expertise en matière de transport d'énergie, de refroidissement des serveurs et de valorisation de la chauffe de ces mêmes serveurs. Comparativement aux ensembles de petites et moyennes tailles issus des infrastructures physiques traditionnelles, le cloud computing se situe effectivement à la pointe des initiatives en la matière. A titre d'illustration, OVH utilise par exemple des refroidissements à eau et, à l'instar d'Orange, a recours à une architecture des bâtiments favorisant la dissipation passive des calories évacuées. Par ailleurs, google va jusqu'à étudier l'utilisation de la chaleur produite à destination du chauffage urbain. Le cloud computing permet donc une mise en place de meilleures pratiques environnementales [27, 81, 171].

Un autre avantage significatif du cloud réside dans la **haute-disponibilité**. Un utilisateur peut ainsi fiabiliser son application et ses données au sein de différents fournisseurs à un coût significativement bas par rapport aux infrastructures physiques traditionnelles.

Les possibilités **conjointes** de **scalabilité** et de **dynamicité** offertes aux utilisateurs [41] constituent des apports également importants.

- La **scalabilité** renvoie à la possibilité de déployer des applications ayant une taille adaptée à une charge donnée : il s'agit d'une mise à l'échelle statique de l'application. Le cloud présente une bonne scalabilité dans la mesure où il permet une gestion aisée des charges qu'elles soient faibles ou au contraire très lourdes, ce que les infrastructures traditionnelles ne permettent pas simplement. La scalabilité a d'ailleurs été un argument décisif retenu par Netflix, un acteur majeur de diffusion de vidéo à la demande qui a décidé de migrer l'intégralité de son infrastructure dans le cloud. NetFlix utilise actuellement 1 Pétaoctet de données, pour 1 milliard d'heures de vidéos diffusées par mois à destination de 48 millions d'utilisateurs se répartissant dans 40 pays.

- La **dynamicité** est liée à la possibilité de démarrer une machine en l'espace de quelques minutes seulement, là où la même opération, sur une infrastructure traditionnelle, nécessitait auparavant des jours voire des semaines.

Le cloud offre effectivement une gestion dynamique de la scalabilité : à chaque instant, chaque utilisateur du cloud peut consommer uniquement les ressources dont il a besoin. Une telle gestion dynamique de la scalabilité permet ainsi aux utilisateurs du cloud d'optimiser leurs dépenses tout en garantissant une gestion de forts pics de charges. Au niveau économique, il s'agit donc d'un avantage séduisant pour les entreprises qui sont ainsi assurées d'avoir une infrastructure à même de supporter de fortes hausses dans leurs activités commerciales. Un exemple souvent mentionné comme un cas d'école est Animoto, un service permettant la création de vidéos personnalisées à partir d'images, de clips vidéos et de musiques. En 2008, ce service hébergé dans le cloud lança son application Facebook, avec la possibilité pour chaque personne de créer gratuitement une

vidéo, à la seule condition de s'inscrire sur le site. La plateforme a alors dû faire face à une augmentation spectaculaire de son utilisation faisant passer son nombre d'utilisateurs de 25000 à 250000 en l'espace de trois jours. Ce succès a alors requis la gestion d'un pic d'inscriptions de 20000 nouveaux utilisateurs simultanés. D'un nombre de 50 VMs à l'origine, la plateforme est grimpée à près de 4000 comme l'illustre la figure 2.6. Grâce à la scalabilité dynamique offerte par le cloud computing, la plateforme a pu être

source : `http://www.rightscale.com/blog/enterprise-cloud-strategies/animotos-facebook-scale`

FIGURE 2.6 – Animoto : un cas d'école d'élasticité

adaptée dynamiquement à la hausse comme à la baisse. Ces deux types d'adaptation sont effectivement primordiaux. L'adaptation à la hausse permet aux applications de toujours être à même de gérer la demande, ce qui garantit une bonne qualité de service à tous les clients. Aucun client n'est ainsi rebuté par un service défectueux ou peu réactif : comparativement aux infrastructures traditionnelles, la perte de clients en raison d'une mauvaise qualité de service est minimisée. Réciproquement, l'adaptation à la baisse permet de maintenir une dépense optimale vis-à-vis de la demande : seuls les services nécessaires sont consommés.

Cependant, la gestion dynamique de la scalabilité nécessite que l'application soit elle-même élastique. Le site web d'Animoto constitue un excellent exemple d'application web élastique, qui a permis de pleinement exploiter les possibilités de gestion dynamique de la scalabilité offertes par le cloud. En outre, le cas d'Animoto montre l'importance primordiale que revêt **l'automatisation de l'élasticité**. Ce n'est effectivement qu'une fois qu'elle est automatisée, que l'élasticité devient à la fois réaliste en termes de coûts, réaliste au niveau de la main d'œuvre nécessaire, mais également beaucoup plus réactive qu'un opérateur humain ne peut l'être [114, 36]. Conscients de l'importance majeure de l'élasticité automatisée, différents acteurs du cloud proposent aujourd'hui des solutions pouvant la gérer, faisant de **l'élasticité automatisée un apport supplémentaire du cloud computing**.

L'automatisation de l'élasticité

L'automatisation de l'élasticité est un apport incontournable du cloud que diverses solutions existantes visent à offrir. La figure 2.7 en illustre l'approche générale qui consiste en une boucle composée de trois étapes : à partir de l'observation de l'application, des décisions d'élasticité sont prises de sorte à adapter son architecture, lesquelles résultent en des réactions qui vont modifier l'architecture courante. Il s'agit là d'une approche générique mentionnée pour expliquer l'approche générale, mais il est à noter que certaines solutions abordent l'automatisation de l'élasticité de façon plus fine avec la boucle MAPE-K [84] telle que décrite dans le chapitre 4.

Dans la boucle simple d'automatisation, grâce à une canevas de surveillance, l'Observation remonte des métriques montrant l'état de l'application : une métrique d'utilisation CPU élevée peut par exemple révéler une surcharge locale qu'il convient de corriger. Pour cela, une brique de prise de décision décide si une *décision d'élasticité* est nécessaire. Une telle opération est ni plus ni moins que la description d'une (dé)croissance verticale ou horizontale : il s'agit en fait d'une méta-modification de l'architecture de l'application à laquelle va correspondre un *plan de reconfiguration*. Ce plan consiste en une séquence d'opérations de reconfigurations telles que l'ajout d'une VM, la détermination du profil matériel de celle-ci. La déclinaison de la décision en un plan de reconfiguration est établie et réalisée dans la partie Réaction de la boucle.

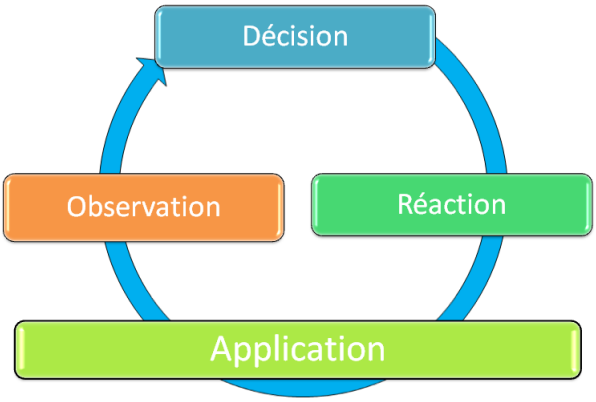


FIGURE 2.7 – Boucle simple d'automatisation de l'élasticité

L'élasticité automatisée est au cœur des travaux de thèse présentés dans ce manuscrit puisqu'il s'agit d'une fonctionnalité majeure et incontournable du cloud computing [26, 105, 131]. Plus particulièrement, ces travaux concernent une partie de la troisième étape comme décrit dans le chapitre 4. Après la description "usuelle" du cloud computing, la suite de cette section vise à montrer des problématiques inhérentes au cloud et qui ont été prises en compte dans les contributions exposées dans ce document. Il s'agit d'obtenir une gestion innovante, performante et évolutive de l'élasticité.

2.2.2 Un paradigme en constante évolution

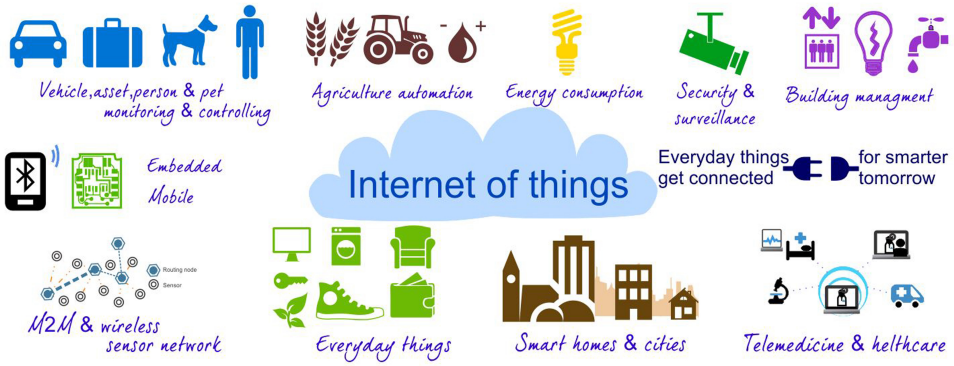
De façon globale, le cloud computing est constitué de grands ensembles répartis dans le monde et interconnectés. Toutefois, ce paradigme est en constante évolution ce qui a eu une incidence directe sur les contributions de la thèse. La suite de cette sous-section fait un point sur l'évolution actuelle du cloud computing.

Un facilitateur d'usages

Si le paradigme du cloud computing a émergé pour répondre à des besoins précis, un aspect amusant réside dans le fait qu'il a lui-même facilité certains de ses cas d'utilisation. Par exemple, l'Internet des Objets (Internet Of Things) ou le Big Data sont deux domaines pour lesquels le cloud computing fait office de catalyseur.

L'internet des objets L'internet des objets se caractérise par une multitude d'éléments connectés ayant des capacités matérielles réduites mais permettant de remonter de grandes quantités d'informations. Ces informations sont potentiellement créées dans le monde entier et le cloud computing permet d'offrir un stockage et un traitement répartis dans différents clouds. La figure 2.8 illustre des cas d'usages pour l'Internet des Objets et son rapport avec le cloud computing.

La répartition géographique est ici un facteur clé en raison des limites de bande passante induites par les longues distances [168]. Au regard des perspectives évoquées de développement de l'internet des objets, ce facteur de répartition prend tout son sens : de 13 millions en 2013, le nombre d'objets connectés devrait atteindre 130 millions en 2018 [165] pour un chiffre d'affaires de 19 milliards de dollars. Il s'agit donc d'un marché émergent qui nécessite un traitement réparti, élastique et multi-cloud [165, 168]



source : `http://www.satiztpm.it`

FIGURE 2.8 – Illustration d'applications de l'Internet des Objets et du lien avec le Cloud Computing

Comme le décrit [175], l'internet des objets avant le cloud computing était à la

fois coûteux et complexe à mettre en œuvre en raison de la nécessité d'administrer de vastes systèmes à la fois hétérogènes et répartis géographiquement. L'émergence du cloud computing est ainsi un facilitateur pour l'internet des objets et il en est de même pour le Big Data.

Le Big Data Le big data concerne des ensembles de données dont la taille empêche l'utilisation et la valorisation par des techniques usuelles de bases de données ou de gestion de l'information : on parle également de *datamasse* en français. Au-delà de la quantité de données, la nature même de ces données rend leur utilisation difficile. Celles-ci sont en effet non-relationnelles et peu structurées voire pas structurées du tout. Une partie importante du travail consiste donc à donner un sens à ces données de sorte à en permettre l'utilisation et donc les valoriser. Pour cela, le big data nécessite une répartition des calculs de même qu'une maîtrise des emplacements géographiques des nœuds de calcul par rapport à ceux des données de sorte à maximiser la proximité. Différents fournisseurs de cloud ont leur propre service de big data comme Google BigQuery, Microsoft Azure ou Amazon Web Services.

Conjointement à l'utilisation d'éléments à haute-scalabilité comme les bases de données NoSQL ou le canevas Hadoop, l'élasticité du cloud est un atout majeur pour le big data. Il est ainsi possible de créer rapidement des applications permettant un traitement d'une quantité importante de données en un temps très court. Toutefois, le big data requiert la maîtrise de l'emplacement géographique des données tant pour des problématiques de performances que pour des problématiques légales.

Le cloud computing permet donc l'émergence de nouveaux usages. Pour certains chercheurs, il s'agit même d'une opportunité de créer de nouvelles applications à la fois scalables, fiables, réparties et peu onéreuses pouvant proposer des services à faibles coûts capables de concurrencer des offres onéreuses et économiquement bien établies issues de l'infrastructure physique traditionnelle. Par exemple [37] propose une application répartie permettant l'acheminement de contenu statique pour des applications web aussi connue sous le terme de Content Delivery Network (CDN). Cette solution est définie par les auteurs comme étant une alternative performante et très accessible de par son prix face à ceux d'un ténor du marché comme Akamai, dont les tarifs le rendent uniquement accessible à de grosses sociétés. L'élasticité des applications de type CDN fait ainsi partie des cas d'usages retenus pour valider les travaux de thèse.

Tous ces usages ne montrent pas seulement l'intérêt de l'élasticité mais aussi sa nécessité. Plus particulièrement, les usages relatés dans cette première partie de sous-section soulignent plusieurs problématiques majeures que l'élasticité doit adresser. Celles-ci concernent la maîtrise des emplacements géographiques des briques applicatives et des données, ainsi que la nécessité d'adresser le *multi-cloud* comme le souligne [116]. Ces problématiques ne sont pas ou peu adressées actuellement lors de la prise en charge de l'élasticité. D'autres problématiques que doit adresser l'élasticité sont exposées dans la suite de cette sous-section.

Le cloud : un paradigme façonné par les évolutions

Le cloud computing a donc été rendu possible en partie grâce aux solutions matérielles fournies par Intel et AMD permettant une assistance matérielle de la virtualisation. Aujourd'hui encore, il continue d'être façonné par les différentes évolutions technologiques matérielles. Par exemple, les processeurs ARM sont souvent présentés comme de bons candidats pour le cloud une fois qu'un jeu d'instructions 64 bits sera réellement disponible : leur consommation maitrisée est effectivement un argument de poids. Parallèlement, l'essor des disques durs à l'état solide [61, 95, 142] (SSD) ou des calculs généralistes sur processeurs graphiques [98, 113, 145, 174] (GPGPU) est en train de modifier le visage du cloud. Des fournisseurs de cloud permettent par exemple de créer des machines virtuelles utilisant du stockage sur SSD et/ou permettant le GPGPU. Certaines initiatives de recherches profitent de cette évolution matérielle constante pour présenter des travaux innovants à base de FPGAs pour *Field Programmable Gate Array*, des circuits électroniques programmables. Par exemple, [44] propose une solution d'acheminement de contenu (CDN) utilisant ces circuits : l'approche proposée offre d'excellentes performances pour une consommation réduite. De même, [140] propose une solution économe en énergie à base de FPGAs permettant l'accès à distance à des machines virtuelles ou physiques.

Au-delà du matériel, des initiatives industrielles de PaaS comme Docker, Cloud Foundry et OpenShift montrent une évolution du modèle du cloud. Ces solutions de PaaS utilisent des conteneurs légers comme base de déploiement. Ce choix peut notamment être vu comme un élément de réponse aux constats de complexité de gestion du multi-cloud par divers travaux de recherches [33, 58, 131]. Les conteneurs légers permettent par exemple aux solutions de PaaS mentionnées ci-avant d'être déployées au sein de différentes VMs lourdes. Les VMs lourdes permettent une bonne isolation des utilisateurs de la plateforme vis-à-vis "du reste du monde" tandis que les conteneurs légers permettent une gestion rapide, simple et légère d'une mutualisation des VMs lourdes sous-jacentes (également montré par [170]). Ces initiatives démontrent que la vision du cloud *logiciel-serveur-VM* elle-même, est sujette à évolution.

2.2.3 Synthèse

Le cloud computing est un paradigme prometteur. Il s'agit d'un environnement d'exécution pour les applications qui offre des possibilités de mise à l'échelle et d'optimisation des coûts très nettement supérieures à celles des infrastructures physiques traditionnelles. Cette optimisation passe par la gestion de l'élasticité des applications au niveau du cloud et plus particulièrement sous sa forme automatisée. L'automatisation de l'élasticité peut être vue de façon générale comme une boucle *observation - décision - action*.

Le contexte actuel montre des applications nécessitant une maîtrise géographique de leur exécution et une répartition sur différents clouds. Ces besoins découlent de contraintes de passage à l'échelle, de haute-disponibilité et d'optimisation de performances. Cela illustre plusieurs enjeux que la gestion de l'élasticité doit adresser : l'élasticité ne doit pas se cantonner à une plateforme ou à un fournisseur mais elle doit per-

mettre de répartir une application au sein de plusieurs clouds. Par ailleurs, l'élasticité doit permettre de maîtriser l'emplacement géographique des éléments d'une application.

Un autre point important réside dans l'émergence de nouveaux matériels et le recours à des approches mêlant conteneurs légers et VMs lourdes : le cloud évolue régulièrement. L'élasticité des applications dans le cloud doit donc permettre d'adresser au mieux ces nouveaux éléments sans être limitée à des sous-ensembles correspondant à une "photo" du cloud à un moment donné.

Les contributions de la thèse tiennent compte de l'ensemble de ces points afin de repousser les limites de la gestion actuelle de l'élasticité des applications dans le cloud computing. Le chapitre suivant aborde justement ces limites afin de positionner les travaux exposés dans ce manuscrit.

Chapitre 3

Etude des solutions d'élasticité

Sommaire

Le chapitre précédent a décrit l'élasticité ainsi que les enjeux devant être adressés par sa gestion automatique dans le cloud computing. Le présent chapitre vise à dresser

un "état des lieux" en matière de gestion automatisée de l'élasticité. Afin de mieux appréhender les solutions décrites, il est nécessaire de comprendre leur positionnement par rapport au cycle de vie d'une application.

3.1 Cycle de vie d'une application et positionnement des solutions de gestion de l'élasticité

Le cycle de vie d'une application est défini par une succession précise de phases. Cette succession a pour but d'organiser et de stipuler le processus nécessaire à la création complète d'une application. Pour chacune des phases du processus, des préoccupations à la fois différentes mais globalement complémentaires sont stipulées par la méthode de gestion de projet utilisée. Les méthodes de gestion de projet sont fortement dissemblables [130, 103, 35, 92, 30] et préconisent des cycles dont les processus sont très différents. Cependant, toutes ces méthodes mentionnent des préoccupations similaires qui peuvent être regroupées en la succession de macro-phases suivantes [49, 54] :

- **La conception** vise à définir l'application à créer. Cette définition se fait en fonction des besoins exprimés par le client. La conception vise à établir un cahier des charges puis les spécifications techniques et fonctionnelles de l'application.

- **La réalisation** concerne la production effective de l'application qui doit couvrir l'ensemble des spécifications de la conception. C'est au cours de la réalisation que le code applicatif est développé, testé et documenté. En fonction de la méthode de gestion de projet et des méthodes de génie logiciel utilisées, des phases d'implantation puis de tests peuvent se succéder continuellement. Il est dès lors primordial pour les équipes de développement de pouvoir effectuer leurs tests de façon rapide et aisée.

- **La validation** est la vérification des caractéristiques de l'application. Il s'agit de s'assurer que l'application couvre de façon effective les exigences exprimées par le client. La validation comprend notamment des tests de performances et de gestion de la charge.

- **Le déploiement** correspond tout d'abord à l'installation et le démarrage de l'application sur une infrastructure en vue de son exploitation.

- **L'administration** regroupe l'ensemble des préoccupations autour de la maintenance de l'application dans le but de permettre son exploitation. Cette maintenance s'appuie sur de la supervision afin de connaître l'état de l'application, ainsi que des mécanismes ou des procédures de reconfigurations et de mises à jour. L'élasticité peut par exemple permettre l'exploitation d'une application en offrant des mécanismes d'adaptation à la charge, qui vont permettre de maintenir une bonne expérience utilisateur.

La prise en compte de l'élasticité durant ces différentes phases détermine l'aptitude finale de l'application à permettre son élasticité. Lors de la conception, l'établissement de l'architecture fonctionnelle et de l'architecture technique requiert de prendre en compte le comportement élastique de la future application. Cette prise en compte se traduit de façon pratique par des architectures logicielles adaptées pour la montée en charge. Lors de la réalisation, les équipes de développement sont amenées à effectuer des tests qui incluent notamment la vérification du bon fonctionnement de l'application durant l'élasticité. Lors de la validation, les tests de performances visent notamment à montrer la bonne montée en charge de l'application, ce qui concerne directement l'élasticité de l'application et sa mise en œuvre. Durant le déploiement, le choix de l'approche utilisée pour réaliser cette phase est cruciale pour l'élasticité : par exemple, une approche manuelle de déploiement permettra une élasticité vraisemblablement moins réactive en comparaison d'une approche automatisée telle que VAMP [54, 55].

Les solutions connues de gestion de l'élasticité adressent tout ou partie des macro-phases décrites. Par exemple, la solution Jelastic [5] a vocation à favoriser le développement (durant la réalisation) en offrant un environnement l'intégration d'outils d'intégration continue [143] tels que Hudson, Maven, Ant ou Jenkins [107]. D'autres concernent plus particulièrement l'administration voire uniquement cette macro-phase à l'instar de [89]. De façon globale, les solutions de gestion de l'élasticité se séparent en deux grandes catégories :

1. **Les solutions centrées sur la décision** et qui se destinent plus particulièrement à l'administration.

2. **Les solutions complètes** qui peuvent viser tout ou partie des macro-phases. Les solutions industrielles visent particulièrement la conception, le déploiement ou l'administration.

Ce chapitre dresse un état de l'art de la gestion de l'élasticité en détaillant chacune des deux catégories en deux sections. Étant donné que le positionnement de chaque catégorie est différent vis-à-vis des travaux exposés dans ce document, des critères différents ont été retenus pour les évaluer. Les critères retenus sont décrits au début de chacune des sections.

3.2 Solutions orientées décision

Cette section présente la première catégorie identifiée de solutions à destination de la gestion de l'élasticité. Ces solutions se focalisent sur la prise de décision qui se situe en entrée de la contribution exposée dans ce manuscrit. Toutes ces solutions sont issues de travaux de recherche, les solutions industrielles se focalisant sur une approche complète (Observation + Décision + Réaction) de sorte à offrir un service d'élasticité automatique de bout-en-bout à leurs clients. L'étude qui suit dans cette section a donc pour principal objectif l'**identification des différentes décisions possibles**.

Les sous-sections suivantes présentent des solutions étudiées de prises de décisions. Ces solutions sont regroupées en fonction du type d'élasticité visée par la prise de décision. Une description synthétique est donnée pour chacune de ces solutions. En fin de section, le tableau 3.1 synthétise les opérations d'élasticité admises par l'ensemble des solutions.

3.2.1 Solutions gérant la prise de décision pour de l'élasticité soit verticale soit horizontale

Les solutions présentées dans cette sous-section décident d'opérations concernant soit élasticité verticale soit l'élasticité horizontale.

La première de ces solutions [94] utilise un processus de décision reposant sur logique neuro-floue. Il s'agit d'une approche hybride alliant la logique floue au réseau neuronal. La solution gère la prise de décision pour l'élasticité horizontale.

CloudScale [139] utilise un modèle prédictif reposant à la fois sur une prédiction à base de transformations rapides de Fourier pour la détection des signatures des *bursts* (i.e montée en charge très rapide) et sur une prédiction à base de chaînes de Markov à temps discret. CloudScale adresse la décision pour de l'élasticité verticale. L'originalité de la solution réside dans sa gestion des fréquences des processeurs des nœuds physiques du IaaS et de leur tension d'alimentation (aussi appelée Vcore). Cette originalité permet de maîtriser l'énergie nécessaire aux applications.

Dans [62], les auteurs s'intéressent à l'utilisation de politiques d'élasticité décrites dans un précédent article [63]. L'approche se base sur la théorie du contrôle et des heuristiques en utilisant une prédiction par modélisation stochastique. La solution adresse la prise de décision d'opération d'élasticité horizontale.

La solution exposée dans [87] propose d'offrir l'élasticité à des applications non-élastiques puisque leur conception n'a pas prévu cette possibilité. Les auteurs adressent la prise de décision concernant l'élasticité horizontale en utilisant une approche originale à base de migrations de VMs. La migration de VMs vise à utiliser toutes les ressources matérielles libres des nœuds physiques sous-jacents.

VirtRL [167] utilise l'apprentissage par renforcement pour effectuer des prises de décisions à destination de l'élasticité horizontale. VirtRL utilise un processus de prise de décision markovien et offre un apprentissage accéléré grâce à un mécanisme d'initialisation innovant en plus d'un mécanisme de convergence accéléré.

Dans [112], la solution présentée décide d'opérations d'élasticité horizontale. Elle vise plus particulièrement l'élasticité des applications avec état. La solution garantit la haute-disponibilité grâce à la gestion de la réplication de sessions. Le cas d'usage présenté porte sur un réseau de télécommunications.

IACRS [72] offre une prise de décision par gestion de dépassement de seuils. Les seuils résultent de la composition de plusieurs métriques là où d'autres solutions ne proposent que des seuils basés sur une seule. L'approche s'intéresse notamment à l'élasticité des répartiteurs de charge en décidant quand il est nécessaire d'ajouter ou retirer un répartiteur en tête d'un ensemble de nœuds de calcul. IACRS vise la prise de décision pour des opérations d'élasticité horizontale.

Cette sous-section a décrit des solutions de gestion de l'élasticité orientées sur la prise de décision et ne gérant qu'un type d'élasticité (i.e soit verticale soit horinzontale). D'une manière générale, l'élasticité verticale est moins abordée que l'élasticité horizontale. Ce constat est sûrement à lier avec le fait que si l'élasticité verticale est bien gérée par les hyperviseurs, elle l'est moins au niveau des IaaS.

3.2.2 Solutions gérant la prise de décision pour de l'élasticité verticale et horizontale

Les solutions de prise de décision présentées dans cette sous-section adressent à la fois l'élasticité verticale et l'élasticité horizontale.

HybridScale [88, 89] est un gestionnaire de niveau de services (SLAs). Il s'agit d'une approche hybride tant au niveau des décisions (élasticité verticale et horizontale) qu'au niveau de la prise de décision elle-même en alliant les modes proactif et réactif [23].

Dans [164] les auteurs proposent une solution qui, en tenant compte du placement des VMs au sein de l'infrastructure du cloud, décide s'il faut plutôt prendre une décision d'élasticité horizontale ou bien une décision portant sur de l'élasticité verticale. Une comparaison des différentes stratégies est proposée.

La solution décrite dans [70] propose des algorithmes de prise de décision qui mixent à la fois de l'élasticité horizontale et de l'élasticité verticale. L'article propose une comparaison de ces algorithmes sur une application web multi-tiers.

DejaVu [160] offre une prise de décision reposant sur une classification de profils de charge. La décision est prise en fonction de l'établissement d'une correspondance entre l'état courant d'une application et un profil de charge. Une particularité de la solution est la méthode pour rendre possible l'établissement du profil : DejaVu effectue un rejeu des requêtes à destination des serveurs surveillés vers son *Profiler*. Pour cela, la solution nécessite l'exécution d'un proxy spécifique placé en tête des serveurs et qui communique ainsi les doublons des requêtes qui leur sont adressées vers le *Profiler*.

L'approche de [134, 23] vise à minimiser les coûts d'exécution d'une application. L'originalité de la solution réside dans la gestion de l'élasticité verticale par regroupement des VMs. Un ensemble de petites VMs peut par exemple être regroupé en une VM ayant un profil matériel plus élevé si cela minimise le prix final.

Les solutions décrites dans cette sous-section gèrent à la fois des décisions pour de l'élasticité verticale et des décisions pour de l'élasticité horizontale. [164] et [134, 23] soulignent l'importance de la gestion du placement et plus précisément, de la décision en tenant compte du placement des différentes ressources au sein de l'infrastructure.

3.2.3 Synthèse

Cette section a présenté des solutions orientées sur la prise de décision uniquement. D'un point de vue fonctionnel, ces solutions se situent en amont des contributions exposées dans ce manuscrit. Elles visent toutes à effectuer des prises de décision d'opérations d'élasticité à partir de l'observation des applications et plus particulièrement des métriques remontées (e.g. utilisation CPU/RAM, temps de réponse de l'application, nombre

de requêtes par seconde). Le tableau 3.1 présente une synthèse des différentes approches mentionnées ci-avant dans le but d'établir un inventaire des décisions possibles. Dans cet inventaire, les entrées sont le nom des solutions et les décisions d'opérations d'élasticité.

Solution	Elasticité verticale	Elasticité horizontale
[23, 134]	Oui	Oui
[62]	Non	Oui
[70]	Oui	Oui
[72]	Non	Oui
[87]	Oui	Non
[88, 89]	Oui	Oui
[94]	Non	Oui
[112]	Non	Oui
[139]	Oui	Non
[160]	Oui	Oui
[164]	Oui	Oui
[167]	Non	Oui

TABLE 3.1 – Inventaire des décisions possibles concernant l'élasticité.

Le tableau 3.1 montre que les solutions présentées adressent les deux types d'élasticité. Outre les deux types d'élasticité présents dans le tableau 3.1, la décision peut également concerner la migration des VMs au sein d'infrastructures à l'instar de certaines solutions décrites dans cette section. Par ailleurs, une solution comme DejaVu [160] requiert l'ajout de proxies propres à la solution mais pas à l'application. Il s'agit donc d'une problématique supplémentaire à prendre en compte pour les travaux de thèse qui se situent en aval de la décision. Cette problématique n'est pas spécifique à DejaVu puisqu'elle concerne également la mise en place de canevas de surveillance de l'application dont les autres solutions de gestion de l'élasticité font usage. De tels canevas de surveillance nécessitent notamment l'ajout et la configuration de sondes afin de remonter des relevés de métriques.

Cette section a permis d'établir un bilan des décisions possibles lors de la gestion automatique de l'élasticité. Ce bilan est primordial puisque comme le détaille le chapitre 4, les travaux de thèse doivent justement offrir un point d'entrée pour une brique spécialisée dans la prise de décision. La section suivante aborde les solutions proposant une boucle complète d'automatisation de l'élasticité.

3.3 Solutions de gestion complète de l'élasticité

Cette section décrit les solutions offrant une gestion de l'élasticité dite "complète". Cet adjectif ne signifie pas que ces solutions gèrent intégralement l'élasticité durant tout le cycle de vie de l'application puisqu'elles peuvent être spécialisées dans la gestion

d'une seule macro-phase. En revanche, ces solutions offrent une boucle complète en matière d'élasticité automatique. Il s'agit de solutions tout-en-un qui comprennent des mécanismes plus ou moins poussés de prise de décision. Ces mécanismes s'appuient sur des observations de métriques (e.g. occupation CPU/RAM, temps de réponse, nombre de requêtes par seconde) afin de décider quand il est nécessaire de réaliser une opération d'élasticité sur l'application. Toutes les décisions prises permettent une réaction qui consiste en des actions de reconfiguration de l'application. La fonctionnalité permettant la déclinaison des décisions en actions de reconfigurations de l'application constitue le cœur de la contribution de cette thèse. Les solutions décrites dans la suite de cette section seront donc abordées suivant cet aspect. Pour chaque solution, une description sera donnée ainsi qu'une évaluation en suivant des critères. La description et les critères qui ont été retenus sont explicités dans les deux sous-sections qui suivent.

3.3.1 Un point sur la description des solutions

La description des solutions complètes repose sur une explication de l'utilisation des différentes solutions selon le point de vue de leurs utilisateurs.Chaque solution fait l'objet d'une description en six temps.

1. Dans un premier temps, une introduction décrit les services de la solution ainsi que ses concepts.

2. L'exécution d'une application au sein d'une des solutions de cette section peut nécessiter un travail préparatoire afin d'adapter l'architecture de l'application à l'environnement du cloud. Pour cela, il est nécessaire de comprendre quels sont les services qui seront consommés par l'application et leurs interactions. Chaque solution peut en effet offrir ses propres services, avec des appellations spécifiques ainsi que des concepts spécifiques. L'exécution d'une application et son élasticité requiert donc d'identifier quels sont les services nécessaires à l'application et comment ces services doivent être utilisés. Pour chaque solution, l'identification de ces services est explicitée dans une sous-section intitulée *Modèle d'application.*

3. Une sous-section nommée *Opérations d'élasticité couvertes* décrit quelles sont les opérations admises par la solution : la gestion de l'élasticité verticale et de l'élasticité horizontale y est abordée.

4. La sous-section nommée *Architecture applicative* aborde la connaissance de l'application qui est offerte à l'utilisateur de la solution d'élasticité (i.e la personne qui exécute son application). Plus précisément, il s'agit de répertorier les descriptions offertes à l'utilisateur. Ces descriptions concernent tout d'abord les services identifiés pour l'application en sous-section *Modèle d'application* . Ces descriptions concernent également la connaissance de l'état de l'application lors de son exécution.

5. La sous-section *Gestion du placement* aborde la maîtrise qui est laissée à l'utilisateur de ces solutions quant à l'emplacement d'exécution des sous-ensembles de

son application. L'absence ou le manque de connaissance sur l'emplacement des données dans le cloud est effectivement une crainte récurrente de ses utilisateurs. Cet aspect concerne également des problématiques légales et de performances des services déployés.

6. La dernière sous-section appelée *Positionnement par rapport aux critères* décrit pour chaque critère donné ci-après, l'évaluation obtenue par la solution. Un graphe en toile d'araignée est fourni de sorte à identifier visuellement les caractéristiques de la solution. La surface n'est pas remplie pour ne pas fausser la perception du lecteur : une grande surface donnerait effectivement une impression de qualité là où la lecture de ce graphe doit être une perception de la carte d'identité des caractéristiques de la solution.

3.3.2 Critères d'évaluation

Chaque solution est évaluée au travers de huit critères. Pour chaque critère, la solution est notée sur une échelle *Faible - Moyen(ne) - Bon(ne) - Fort(e)*. La suite de cette section décrit et justifie les critères retenus.

Indépendance applicative

Ce premier critère permet d'évaluer l'aptitude de la solution à adresser une élasticité pour un ensemble peu ou non restreint d'applications. Par exemple, certaines solutions d'élasticité sont des mécanismes ad hoc propres à une application comme DELMA [56], CAP2020 [120], ou ElastiCLIF [148]. D'autres moins spécifiques peuvent ne gérer qu'un type d'applications : les applications web ou HPC [79] par exemple. Ce critère permet donc d'évaluer l'étendue des types d'applications pouvant être rendus élastiques par la solution.

- *Faible.* Une indépendance applicative évaluée à *Faible* traduit une forte adhérence de la solution à un type d'application. C'est par exemple le cas d'une solution ne gérant que les applications web.

- *Moyenne.* La solution est évaluée à *Moyenne* pour ce critère lorsqu'elle adresse un ensemble fini de types d'applications. Cet ensemble a une cardinalité strictement supérieure à 1 et chaque application ne peut correspondre qu'à un unique type. Par exemple, une solution ne permettant de rendre élastique des applications qu'uniquement de type web ou qu'uniquement de type MapReduce [67] sera évaluée à *Moyenne*. Pour une application donnée, celle-ci ne peut donc pas résulter de la composition de plusieurs types.

- *Bonne.* L'indépendance applicative d'une solution est évaluée à *Bonne* si la solution gère un ensemble de types d'applications avec une cardinalité strictement supérieure à 1, tout en permettant de composer entre eux les différents types d'applications gérés. Par exemple, une solution permettant de rendre élastique les

applications à la fois de type web et de type MapReduce, obtiendra une évaluation pour ce critère à *Bonne*.

- *Forte*. Cette évaluation est retenue si la solution n'a pas de limite connue concernant sa gestion d'applications.

Indépendance architecturale

Le critère d'indépendance architecturale traduit l'inverse de l'adhérence de la solution à un type d'architecture. La notion de "type d'architecture", réfère au gabarit (ou pattern) des éléments constitutifs de l'application. Par exemple, une application qui a une architecture organisée en tiers, a un type d'architecture multi-tiers. Pour ce type d'architecture, l'élasticité globale de l'application peut être opérée de façon simple au moyen de l'ajout ou du retrait d'éléments constitutifs d'un tiers. C'est un cas d'usage de base puisque très répandu dans les solutions connues de gestion de l'élasticité. Toutefois, la gestion de ce seul cas ne suffit pas à adresser tous les besoins : c'est pour cela qu'il est nécessaire d'évaluer chaque solution en fonction de sa non-adhérence à un type d'architecture, c'est-à-dire qu'il faut évaluer son indépendance face aux types d'architectures. Ce critère reflète ce dernier point et le détail de son évaluation est comme suit :

- Une indépendance *Faible* correspond au fait que la solution ne gère qu'un type d'architecture. Un exemple typique est une solution ne gérant que les applications multi-tiers.

- Une indépendance *Moyenne* correspond à une solution gérant plusieurs types d'architectures sans permettre de les composer.

- L'indépendance architecturale est évaluée à *Bonne* si la solution gère plusieurs types d'architectures et que la composition de ces architectures est possible.

- L'évaluation de ce critère est *Forte* lorsqu'il n'y a pas de limite connue à la solution. Celle-ci peut donc a priori gérer n'importe quel type d'architecture.

Il est à noter que ce deuxième critère n'est pas complètement orthogonal au précédent. Par exemple, une solution ne gérant que les applications web n'adressera vraisemblablement que des applications avec une architecture multi-tiers. En revanche, la réciproque n'est pas vraie. La lecture de ces critères doit donc tenir compte de ce fait.

Indépendance vis-à-vis de l'environnement

L'indépendance de l'environnement vise à évaluer si la solution de gestion de l'élasticité est spécifique à un ensemble de fournisseurs de cloud. Ce critère est primordial dès lors qu'il s'agit d'établir les risques d'enfermement propriétaire. Une indépendance forte est garante d'un risque minimum là où une indépendance faible indique que ce risque est maximal.

- L'indépendance vis-à-vis de l'environnement est évaluée à *Faible*, la solution ne permet d'accéder qu'à un unique IaaS.

- Evaluée à *Moyenne*, l'indépendance vis-à-vis de l'environnement indique que la solution peut adresser un ensemble fini et non-extensible de plusieurs IaaS.

- *Bonne* est l'évaluation retenue pour une solution qui peut gérer un ensemble fini et extensible de plusieurs IaaS. La seule limite provient du fait que le déploiement d'une application ne peut être réalisé sur plusieurs d'entre eux en même temps.

- Une indépendance *Forte* à l'environnement indique que la solution de gestion de l'élasticité est à même de gérer une application sans limite relative quant aux IaaS accessibles. L'ensemble de ces IaaS est extensible et le déploiement d'une application peut être réalisé sur plusieurs d'entre eux en même temps.

Gestion des applications patrimoniales

La gestion des applications patrimoniales est un critère-clé qui permet d'évaluer le support d'applications qui ne peuvent ni être recompilées ni être modifiées pour la plateforme. Comme le souligne [33], il s'agit d'une difficulté majeure à laquelle les entreprises font face. La migration des application patrimoniales dans le cloud constitue effectivement un frein à son adoption. Pour ce critère, plus une solution a une évaluation proche de *Forte*, moins cette solutions nécessitera de modifier l'application. De fait, la nécessité de recompiler une application constitue déjà une modification. En revanche, ce critère n'est absolument pas révélateur de la quantité de travail à fournir pour que l'application soit portée sur la solution. Une solution peut par exemple nécessiter une recompilation complète, sans que cela nécessite le moindre travail supplémentaire par rapport à un environnement en dehors du cloud. Cette solution sera évaluée comme ayant une gestion *Faible* des applications patrimoniales alors même qu'elle ne requiert aucun effort additionnel. Cet exemple a aussi sa réciproque : une solution peut obtenir une évaluation *Forte* tout en demandant le développement de pilotes spécifiques. Le présent critère doit donc être vu comme étant en lien avec les modifications à apporter à une application existante du point de vue de la recompilation et de la modification de l'architecture de l'application. Voici l'évaluation du critère de gestion des applications patrimoniales :

- *Faible* : la solution nécessite une recompilation de l'application et éventuellement des changements dans son architecture.

- *Moyenne* : la solution ne nécessite pas une recompilation complète de l'application mais uniquement de certains composants.

- *Bonne* : aucune recompilation n'est requise. En revanche, il est nécessaire d'effectuer des changements dans l'architecture de l'application. Par exemple, il est requis de changer le serveur métier d'une application web.

- *Forte* : aucune modification de l'application n'est requise. Le seul effort porte sur la création de pilotes des composants par la solution, notamment pour gérer leur cycle de vie.

Granularité des opérations

La granularité est un critère qui concerne les capacités de reconfiguration des applications que peuvent gérer les solutions durant l'élasticité. Plus précisément, ce critère a pour objectif l'évaluation de la finesse des opérations admises par la solution. Ce point est important car plus la finesse est faible plus les opérations possibles sont grossières. D'une granularité grossière découle une limitation au niveau du comportement élastique de l'application puisque certains paramètres sont alors cachés et ne peuvent donc pas être administrés. A titre d'illustration, une solution gérant uniquement l'ajout ou le retrait de VMs typées, toutes similaires et, avec l'usage implicite d'un répartiteur de charge, ne permet par exemple pas d'adresser les problématiques d'élasticité verticale. L'évaluation de ce critère est finalement :

- Granularité *Faible* : La solution a une granularité au niveau de la VM. Seuls des ajouts et retraits de VMs issues d'une même image virtuelle sont permis et avec un profil matériel fixe.
- Granularité *Moyenne* : la finesse des opérations offertes par la solution concerne des groupes de composants. La solution ne réalise en fait que des traitements par lots sur les composants sans possibilité de gestion individuelle. Tous les composants d'un même type sont donc administrés de la même façon. En outre, il n'y a pas de gestion du placement de ces composants.
- Granularité *Bonne* : la solution permet d'administrer le profil de VMs typées. De façon concrète, la solution gère l'élasticité verticale.
- Granularité *Forte* : la solution admet des opérations élastiques au niveau des composants. Les placements sont donc gérés par la solution et il est par exemple possible d'ajouter/de retirer des composants dans une VM existante. L'ensemble des paramètres de configuration de l'architecture est également administrable.

Modélisation de l'application à l'exécution

La modélisation de l'application à l'exécution est un critère permettant d'apprécier la connaissance que la solution donne à l'utilisateur de l'état courant de son application. Une telle connaissance doit être explicite et est bien entendu nécessaire afin de pouvoir évaluer le bon fonctionnement de l'application à des fins de tests de conception, de validation, de vérification du déploiement et bien évidemment pour pouvoir réaliser l'administration de cette application.

- *Faible* modélisation de l'application à l'exécution : une solution obtenant cette évaluation n'offre qu'une vue des services consommés de façon globale et les services consommés ne sont pas liés à une application précise.

- *Moyenne* modélisation de l'application à l'exécution : cette évaluation correspond à une modélisation reposant sur des consommations de services par application sans explicitation ni des composants ni des liaisons au niveau applicatif.
- *Bonne* modélisation de l'application à l'exécution : cette évaluation signifie que la solution décrit les composants avec leurs liaisons mais sans les placements.
- *Forte* modélisation de l'application à l'exécution : la solution mentionne à la fois les composants, leurs liaisons, leurs placements et leurs configurations.

Modélisation de l'élasticité

Ce critère est central pour une solution de gestion de l'élasticité. Il s'agit de la description du comportement élastique de l'application. En d'autres termes, cette modélisation sert à déterminer les modifications possibles de l'application durant l'élasticité. Suivant la solution, cette modélisation va consister à interdire des évolutions non voulues par exemple en spécifiant un minimum et maximum pour chaque ensemble de composant. Cette modélisation peut aussi expliciter comment doivent être construites les liaisons entre composants. Cependant toutes les solutions n'offrent pas le même degré de modélisation et certaines n'en permettent même pas du tout. Chaque solution a donc été évaluée suivant ce critère de la façon qui suit :

- *Faible*. Cette évaluation indique qu'aucune modélisation n'est offerte par la solution. Celle-ci s'appuie donc uniquement sur un comportement implicite de l'application.
- *Moyenne*. Lorsque la solution ne permet que de lister des services de la solution (ou bien des types de VMs), l'évaluation retenue est *Moyenne*.
- *Bonne*. La solution permet de modéliser une application au travers de l'expression de ses composants mais sans notion **explicite** soit de placement, soit de liaison, soit de configuration.
- *Forte*. L'évaluation de la solution est forte pour ce critère lorsque le modèle gère à la fois les notions de composants, de placements, de liaisons et de configurations. Ce type de modèle permet l'expression de l'entièreté de l'évolution d'une architecture applicative durant l'élasticité.

Expression de contraintes

L'expression de contraintes est un critère d'évaluation de la possibilité que la solution offre à ses utilisateurs de décrire des exigences portant sur le comportement élastique de l'application. Il ne s'agit non plus seulement d'évaluer l'expression de la modification d'une architecture applicative, mais d'évaluer s'il est possible d'exprimer des contraintes qui vont régir à la fois les liaisons, les placements, les cardinalités de composants et les configurations de tous les éléments de l'architecture. La grille d'évaluation retenue pour ce critère est donc :

- *Faible* : aucune contrainte n'est exprimable au travers de la solution, pas même des cardinalités minimale et maximale sur les composants.
- *Moyenne* : il est possible d'exprimer des contraintes simples portant sur des cardinalités de composants ou de VMs. Les placements, les liaisons ou les configurations ne sont pas contraignables.
- *Bonne*. L'expression des contraintes est évaluée à *bonne* lorsque la solution permet d'exprimer des contraintes plus complexes que des cardinalités.
- *Forte*. Cette évaluation est retenue si la solution laisse la possibilité d'étendre les contraintes gérées.

3.3.3 Heroku

Créé en 2007 puis racheté en 2010 par Salesforce.com [16], Heroku [6] est un service de PaaS spécialisé pour les applications écrites en langages Ruby, Node.js, Java, Python, Clojure et Scala. Il gère également certains canevas de programmation basés sur ces langages. Il fut originellement destiné au développement rapide d'applications web. Cette préoccupation caractérise encore aujourd'hui cette offre.

Le déploiement dans Heroku s'effectue au travers du logiciel de gestion de version git. Il suffit de pousser le code source sur le dépôt distant d'Heroku à partir du dépôt local. De façon préalable, l'utilisateur doit renseigner un fichier mentionnant les commandes admises par chaque conteneur d'exécution ainsi que d'éventuelles dépendances extérieures, telles que les plugins d'un serveur Node.js requis pour le fonctionnement de l'application par exemple. L'approche de Heroku repose sur des *Buildpacks*, des paquets de construction permettant à la fois de compiler le code mais aussi de l'exécuter. Cette approche est suivie par une large communauté et les *Buildpacks* ainsi créés sont largement utilisés par d'autres solutions décrites dans ce manuscrit.

Modèle d'application

Si Heroku repose sur l'infrastructure d'Amazon, celle-ci n'est pas directement visible par l'application. Les VMs d'EC2 et le stockage de S3 sont ainsi utilisés de façon quasiment transparente. L'application est exécutée dans des conteneurs légers appelés *dynos*. Chaque dyno peut exécuter un certain nombre de processus :

- Des processus orientés présentation de pages web.
- Des processus pour les traitements métier appelés *workers*.
- Des processus systèmes qui permettent notamment la gestion de l'interface en ligne de commandes, de l'administration et de l'ordonnancement.

L'ensemble des dynos requis pour l'exécution d'une application est appelé *slug*. Lors du déploiement, le BuilPack nécessaire est automatiquement détecté et va déterminer les dynos requis pour l'application.

Opérations d'élasticité couvertes

Chaque slug est élastique au travers d'opérations d'élasticité horizontale portant sur le nombre de dynos. Les dynos ne sont pas élastiques en eux-mêmes. Ils provoquent notamment des notifications d'erreurs lorsque leur consommation de mémoire vive excède le quota fixé par leur profil. Une fois ce quota dépassé, la mémoire supplémentaire est prise sur le swap d'où une dégradation importante des performances du dyno. Si ce rapport parvient à cinq, le dyno incriminé est redémarré automatiquement.

Heroku met à disposition différents profils de dynos. Ces profils sont caractérisés par le nombre de cœurs virtuels et la quantité de RAM. Les profils des dynos peuvent

modifiés mais, pour un slug donné, un seul profil est admis par type de dyno. Le redimensionnement d'un dyno entraine successivement le redimensionnement de l'ensemble des dynos de même type pour le slug auquel il appartient, puis leur redémarrage.

Architecture applicative

L'architecture de l'application est implicite et répartie en dynos dont le type détermine le comportement au sein de l'application. Plus précisément le type d'un dyno induit les liaisons vers les autre(s) type(s) de dynos. Chaque type de dyno est présent dans un catalogue fixe.

Certains types de dynos doivent respecter le pattern singleton : à chaque instant, un seul et unique dyno de ce type doit exister pour une application. C'est notamment le cas du dyno de type *clock* qui est une horloge pour l'application. Il appartient à l'utilisateur de ne pas commettre l'erreur de l'instancier plusieurs fois, ce sans quoi des dysfonctionnements pourraient apparaître.

Gestion du placement

Le placement est transparent pour l'utilisateur au niveau des dynos. Ceux-ci sont répartis au sein de l'infrastructure physique par la plateforme de sorte à ce qu'au sein d'un même slug, tous les dynos d'un même type soient placés sur des machines différentes.

Au niveau des slugs, l'utilisateur maitrise leur emplacement d'exécution. Les machines en elles-mêmes sont disponibles dans deux régions correspondant à deux datacentres d'EC2 :

- La région USA correspond au datacentre *us-east-1*.
- La région Europe en cours de test correspond au datacentre *eu-west-1*.

Par défaut, les applications sont déployées aux USA, sauf demande explicite lors du déploiement. Un slug est entièrement localisé dans une région.

Positionnement par rapports aux critères

- L'*Indépendance applicative* est *Faible* puisque Heroku ne gère que des applications Web. Il s'agit d'une limite due à l'utilisation de routeurs http/https qui empêchent tout autre trafic.
- *Indépendance architecturale.* Heroku ne permet que le déploiement d'applications ayant une architecture multi-tiers. L'indépendance architecturale est donc évaluée à *Faible.*
- *Indépendance vis-à-vis de l'environnement.* Heroku s'appuie exclusivement sur le IaaS Amazon EC2 alors même que les conteneurs légers permettent potentiellement une couverture plus vaste. Ce critère est en conséquence évalué à *Faible.*

- La *Gestion des applications patrimoniales* est évaluée à *Faible* puisque leur déploiement dans Heroku requiert de les recompiler.
- L'évaluation concernant la *Granularité des opérations* est *Moyenne* car Heroku offre une granularité au niveau du groupe de composants. Même s'il est possible de rajouter des composants individuellement (chacun étant exécuté dans un conteneur), l'élasticité verticale ne peut s'appliquer qu'à l'ensemble des dynos de même type.
- Concernant la *modélisation à l'exécution*, celle-ci est évaluée à *Moyenne* en raison de l'absence d'un modèle ayant un niveau de détail plus fin qu'une liste des dynos utilisés par l'application.
- La *modélisation de l'élasticité* évaluée à *Moyenne*. Heroku n'offre effectivement qu'une approche explicite mentionnant une liste des types de dynos que l'application utilise.
- Enfin, l'*Expression de contraintes* est évaluée à *Faible* : aucune expression n'est possible. L'exemple du pattern singleton est révélateur de ce point.

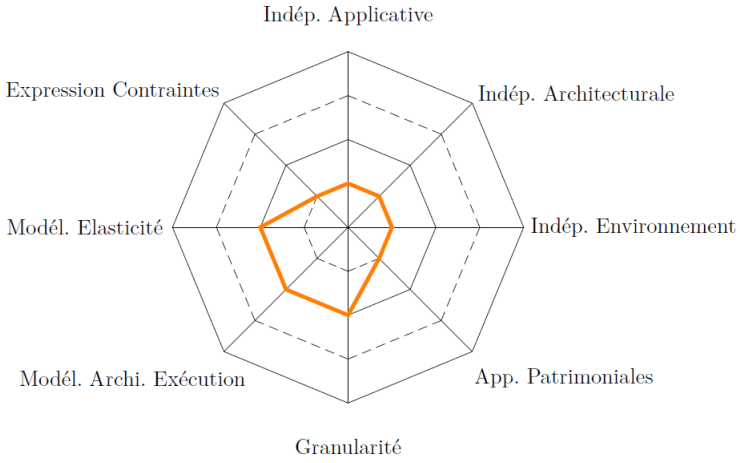


FIGURE 3.1 – Evaluation d'Heroku par rapport aux critères d'étude

3.3.4 OpenShift

OpenShift [15] est l'évolution d'une solution de PaaS initialement développée par la société Makara, rachetée en 2010 par la société RedHat. Cette solution permet le déploiement et l'élasticité d'applications au travers de la composition de services élastiques. L'offre comprend à la fois des versions gratuite et payante pour un cloud privé et une offre en ligne.

OpenShift a pour première préoccupation d'accélérer et faciliter le développement d'applications web. Pour cela, OpenShift propose une interface avec le logiciel de gestion de versions git. OpenShift est en outre interfacé avec l'IDE Eclipse et gère un environnement d'intégration continue composé de Maven et Jenkins. Pour le déploiement, OpenShift utilise des conteneurs légers reposant sur SELinux. Ces conteneurs sont exécutés sur des *nodes* à la demande d'un orchestrateur hébergé sur le *broker*. Le broker et les nodes communiquent entre eux au travers d'un bus à messages utilisant le protocole AMQP. Il s'agit d'une solution de PaaS très en vogue dans l'industrie et également étudiée dans des travaux de recherche [101].

Modèle d'application

OpenShift propose un modèle d'application reposant sur la composition de services. Ces services sont issus de modèles appelés *cartridges*. Les cartridges sont à la fois un élément de modélisation de l'application élastique, mais aussi des composants exécutés. Chaque cartridge est notamment constitué d'un fichier de manifeste décrivant l'ensemble des services utilisés. Ce fichier de manifeste renseigne également l'arborescence des fichiers d'exécution. Une fois instanciée, l'application est composée d'un ensemble de conteneurs légers disposant chacun d'un profil matériel. Ces conteneurs sont appelés *gears* et sont exécutés sur les nodes. Chaque gear exécute un cartridge.

Le modèle d'applications proposé par OpenShift est exposé à l'utilisateur soit au travers de la console web hébergée par le broker, soit au travers d'une interface en ligne de commande.

L'ensemble des cartridges est disponible au sein d'un catalogue exposé par les interfaces d'OpenShift. OpenShift offre la possibilité de rajouter de nouveaux cartridges.

Une contrainte importante est que toute application doit au moins avoir un cartridge web (e.g. PHP, Node.js).

Opérations d'élasticité couvertes

L'élasticité d'une application est réalisée au travers de l'addition, le retrait ou le maintien des gears. Ces gears ont un profil matériel correspondant à un quota de ressources de leur node d'exécution. Ce profil est déterminé par le cartridge qu'ils exécutent et demeure fixe. Il est important de noter que seul le profil *small* est accessible avec l'offre gratuite tandis que des profils *medium* et *large* existent. Pour chaque cartridge constitutif de son application, l'utilisateur peut déterminer les paramètres suivants :

- La gestion par le cartridge de l'élasticité.

- Cardinalités minimale et maximale de gears.

Lors de l'exécution de l'application, OpenShift permet de mettre à l'échelle le nombre de gears soit de façon manuelle, soit de façon automatique. Pour cela, la solution se base sur son propre système de collecte de métriques pour déterminer le nombre de gears nécessaires à la gestion du trafic à destination de l'application.

Architecture applicative

L'architecture d'une application provient, avant le déploiement, de la composition de services offerts par les cartridges. Durant l'exécution, l'utilisateur peut accéder à une vue architecturale simple mentionnant les gears exécutés, leur profil, et une adresse pour un accès direct via SSH. Il n'y a pas de mention de liaisons architecturales mais il y a une cohérence applicative. En effet, l'utilisateur peut choisir d'afficher uniquement les gears appartenant à une application précise.

Gestion du placement

De façon automatique, OpenShift distribue les gears d'un service au sein des différents nodes. Il n'y a pas de façon de maîtriser le placement des gears au sein des nodes. Il n'est par ailleurs pas possible de déterminer des zones groupant des nodes par exemple.

Positionnement par rapports aux critères

- L'*Indépendance applicative* est *Faible* étant donné que seules les applications web sont gérées.
- L'*Indépendance architecturale* est également évaluée à *Faible* en raison de la spécificité d'OpenShift aux architectures n-tiers.
- *Indépendance vis-à-vis de l'environnement.* OpenShift voit ce critère avoir une évaluation *Forte* grâce à la portabilité des conteneurs légers : cela permet effectivement à la solution d'être déployée au sein de n'importe quel IaaS.
- La *Gestion des applications patrimoniales* est *Moyenne* puisque la gestion des applications par OpenShift dépend de la présence d'un cartridge adapté. S'il est possible d'en rajouter, rien n'indique par exemple que ce puisse être possible avec une vieille version d'un serveur ce qui entraîne donc potentiellement des modifications dans l'application.
- La *Granularité des opérations* est évaluée à *Moyenne*, celle-ci se situant au niveau des groupes de composants.
- L'évaluation retenue pour la *Modélisation à l'exécution* est *Moyenne* : seule une liste des services consommés est disponible.

- Au niveau de l'évaluation de la *Modélisation de l'élasticité*, celle-ci est *Faible* car le comportement élastique de l'application est presque intégralement implicite.
- L'*Expression de contraintes* est évaluée à *Faible* car seules des cardinalités simples sont exprimables.

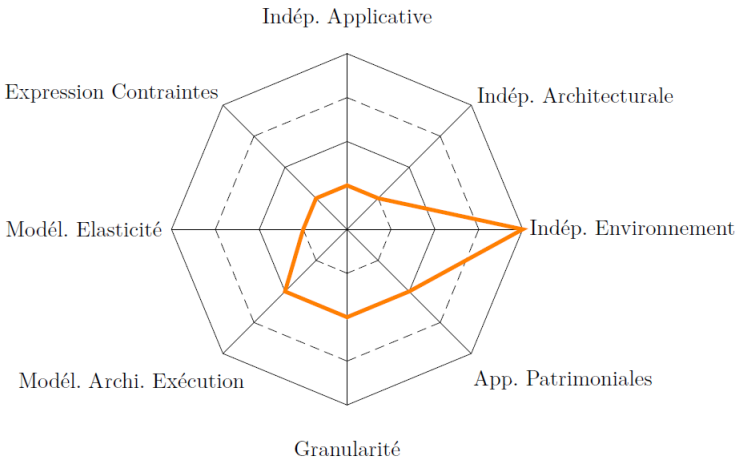


FIGURE 3.2 – Evaluation d'OpenShift par rapport aux critères d'étude

3.3.5 Jelastic

Jelastic [5] est une société créée en 2010 qui fournit une offre de PaaS à destination des applications web écrites en Java, en PHP ou en Ruby. La plateforme de Jelastic permet un déploiement rapide pour les développeurs grâce à sa gestion d'outils d'intégration continue tels que Hudson, Maven, Ant ou Jenkins. La plateforme permet de déployer une application de deux façons différentes :

- Au travers de l'interface graphique fournie par la plateforme.
- Au travers de plugins pour différents environnements intégrés de développement (Netbeans, Eclipse, IntelIj IDEA).

La plateforme s'interface également avec les logiciels de gestion de versions Git et SVN. En ce qui concerne la virtualisation, Jelastic repose sur Parallels Virtuozzo Containers, une solution propriétaire de virtualisation à base de conteneurs légers linux. Les applications sont hébergées dans des conteneurs dont la consommation en ressources physiques est exprimée en *Cloudlets*. Chaque Cloudlet correspond à 200MHz de processeur et 128Mo de RAM. Les clients de la plateforme sont facturés en fonction de leur consommation en Cloudlets.

Modèle d'application

Jelastic propose un ensemble de services dont la composition permet de déployer et exécuter une application. Comme Jelastic vise des applications web, la plateforme propose différents serveurs se répartissant dans les catégories qui suivent :

- Répartition de charge : Nginx, Apache.
- Serveur d'application (tiers métier) : Tomcat, TomEE, GlassFish, Jetty.
- Base de données : mongoDB, MariaDB, MySQL, CouchDB, PostgreSQL.
- Services annexes d'intégration continue.

.

L'utilisateur explicite son application en commençant par spécifier les services qui la composent. Ces services sont ensuite dimensionnés et configurés. La plateforme permet un accès total aux paramètres de configuration des différents services.

Opérations d'élasticité couvertes

Jelastic permet des opérations élastiques verticales et horizontales.

L'élasticité verticale se situe au niveau du nombre de Cloudlets attribués à chaque service. Initialement, l'utilisateur dimensionne chaque service en renseignant un nombre minimal de Cloudlets. Ce dimensionnement minimal sera réservé et donc attribué en permanence. En revanche, lors de pics de charge, la plateforme permet aux services

d'utiliser plus de cloudlets de façon automatique. L'automatisation de l'élasticité verticale est transparente pour l'utilisateur.

L'élasticité horizontale consiste en la possibilité d'ajouter et retirer dynamiquement des serveurs à l'application. Un cas particulier concerne la possibilité de définir un serveur Tomcat en haute-disponibilité de sorte à ce que chaque serveur est redondé par un autre. Les serveurs Tomcat sont alors organisés en doublons et l'ajout d'un serveur entraine automatiquement l'ajout d'un second.

Architecture applicative

L'architecture applicative est implicite dans Jelastic. Celle-ci provient effectivement de la composition de services et des différentes instances exécutées. Les différentes instances en cours d'exécution sont consultables au travers de l'interface web de la plateforme sans toutefois offrir une réelle vision architecturale.

Gestion du placement

Jelastic repose sur différents fournisseurs de IaaS. Il est possible de spécifier l'emplacement d'exécution de l'application. Pour chaque application, la plateforme place chaque serveur d'un même service sur un nœud physique différent. C'est aussi la plateforme qui place globalement tous les serveurs de sorte à optimiser l'utilisation des ressources physiques. Ce placement complètement automatisé est transparent pour l'utilisateur. La plateforme optimise ces placements en réalisant notamment des migrations à chaud de serveurs. Par ailleurs, toutes les applications sont hébergées dans leurs propres conteneurs de sorte à maintenir une isolation entre les applications.

Positionnement par rapports aux critères

- *Indépendance applicative.* Jelastic gère uniquement des applications web. Ce critère est donc évalué à *Faible.*

- L'*Indépendance architecturale* de la solution est évaluée à *Faible* car Jelastic est spécifique aux applications n-tiers.

- *Indépendance vis-à-vis de l'environnement.* Jelastic repose sur une virtualisation à base de conteneurs légers et exploite différents offres de IaaS réparties dans le monde. Chaque application étant uniquement déployée dans un seul IaaS, l'indépendance à l'environnement est donc *Bonne.*

- *Gestion des applications patrimoniales.* La gestion des applications patrimoniales est évaluée à *Faible* puisqu'elle reste soumise à la bonne adaptation des binaires de l'application aux services à disposition. Si tel n'est pas le cas, il est nécessaire de modifier l'application en profondeur.

- La *Granularité des opérations*est évaluée à *Moyenne*. Jelastic admet à la fois des opérations d'élasticité horizontale et des opérations d'élasticité verticale mais cela concerne des groupes de composants.
- Concernant la *Modélisation à l'exécution*, ce critère obtient une évaluation *Moyenne* puisque Jelastic fournit uniquement une vue de la consommation des services par l'application sans raffinement supplémentaire.
- *Modélisation de l'élasticité* : Jelastic permet de modéliser une application en composant des services. Cette composition est accessible au travers de l'interface de la plateforme. Le reste du comportement élastique étant uniquement implicite, l'évaluation de ce critère est *Moyenne*.
- L'*expression de contraintes* est évaluée à *Bonne* grâce à la possibilité d'exprimer une contrainte de haute-disponibilité sur le frontal d'une application, en plus de la gestion de contraintes décrivant des cardinalités.

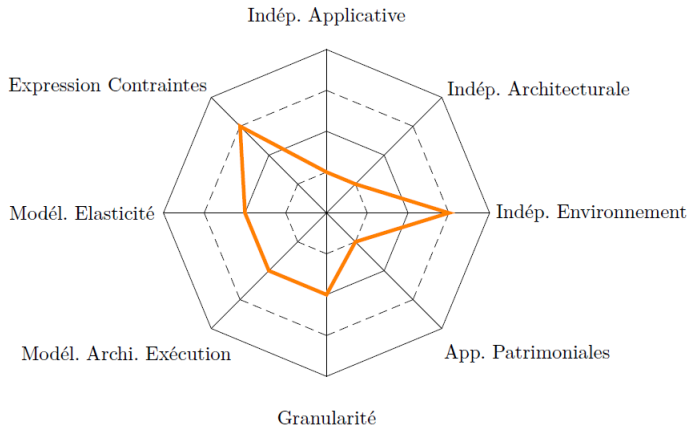


FIGURE 3.3 – Evaluation de Jelastic par rapport aux critères d'étude

3.3.6 Ecosystème Docker

Docker [17] n'est pas une solution permettant de gérer l'élasticité. Il s'agit en fait d'un projet open source qui automatise la gestion de conteneurs légers linux. S'appuyant lui-même sur LXC, Docker permet de créer des conteneurs légers et de les distribuer sous forme d'images de sorte à offrir la possibilité d'avoir toujours les mêmes conditions d'installation et/ou de déploiement de binaires d'applications hébergées par ses conteneurs. Docker est ici présenté en raison de l'écosystème de PaaS se basant dessus. La suite de cette section explore différentes solutions appartenant à l'écosystème de Docker.

Deis [12], Dokku [7] et Flynn [18] sont regroupées dans le même graphe en raison de leurs fortes similarités. Ces solutions sont toutes basées sur Docker et ont une approche qui vise à ressembler à la solution proposée par Heroku. Toutes sont disponibles au téléchargement et peuvent être installées librement.

Deis

Deis est une solution open source créée et maintenue par la société *OpDemand*. Deis permet de déployer des conteneurs légers et de gérer l'élasticité des applications qu'ils hébergent. Deis repose sur Docker et gère les *Buildpacks* de Heroku. Cette solution est à même de provisionner automatiquement des conteneurs sur les infrastructures EC2, Rackspace et Digital Ocean.

Deis supporte de base les applications écrites en Ruby, Python, Node.js, Java, Clojure, Scala, Play, PHP, Perl, Dart et Go. Cependant, Deis supporte plus généralement toutes les applications disposant d'un Buildpack ou de fichiers de configuration Docker. Le déploiement d'une application est réalisée au travers de Git.

Concernant l'élasticité, Deis permet des opérations horizontales d'ajout et de retrait de composants au niveau du tiers métier. La configuration des éventuels nouveaux conteneurs et celle relative à la répartition de charge est gérée automatiquement par Deis. Pour cela, Deis intègre notamment un serveur Nginx qui permet de rediriger les requêtes entrantes de la plateforme vers l'application de destination. C'est au travers de Nginx qu'est réalisé la répartition de charge.

Dokku

Dokku (**Do**c**k**er - Hero**ku**) est un projet open source permettant de déployer un "mini-heroku" comme le décrit la page d'hébergement du projet. Ce projet en développement permet de déployer un environnement d'exécution d'applications web. Dokku offre de base la gestion des applications écrites en Node.js, Ruby et Python. Il est par ailleurs possible d'ajouter la gestion de langages supplémentaires.

Dokku vise le déploiement rapide d'applications non patrimoniales. En effet, le déploiement s'effectue en poussant le code de l'application au travers de Git en direction d'une plateforme Dokku.

La gestion de l'élasticité est réalisée au travers du plugin *dokku-logging-supervisord*. Ce plugin permet de définir des opérations de pseudo-élasticité horizontale. Il s'agit en

réalité d'opérations de mise à l'échelle qui requièrent le redémarrage de l'application pour être exécutées. De plus, tous les composants d'un même type sont en réalité hébergés au sein du même conteneur.

Dokku réalise la répartition de charge et le routage des requêtes vers les applications grâce à Nginx, un serveur HTTP et reverse proxy.

La portabilité des conteneurs légers est clairement à l'avantage de Dokku qui peut être déployé au sein de n'importe quel IaaS offrant des VMs avec un OS Linux (de préférence Ubuntu). Dokku est par exemple directement accessible dans l'offre du fournisseur de IaaS Digital Ocean.

Flynn

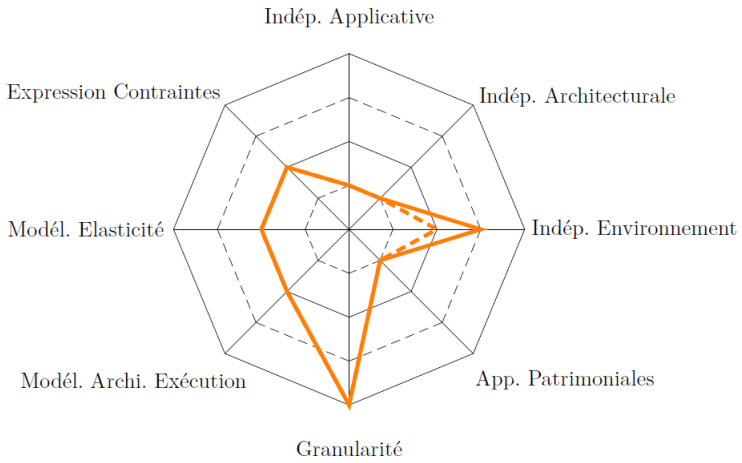
Flynn est une solution open source permettant de déployer une application dans des conteneurs légers grâce à Docker. Flynn s'appuie sur un *scheduler* qui répartit les différentes tâches de création suppression, configuration des conteneurs entre les différents nœuds qui constituent la plateforme. Flynn permet le déploiement d'applications web au travers d'un client en ligne de commandes. Flynn propose une approche originale pour le point d'entrée de la plateforme (tiers présentation des applications) : il s'agit d'un routeur spécifique à Flynn, appelé *Strowger* qui, à l'instar de Nginx pour d'autres solutions, permet le routage HTTP et TCP en direction des applications déployées par la plateforme. Il ajoute une facilité de gestion de la découverte dynamique de services. Strowger rend ainsi possible la composition de services hébergés par la plateforme. Flynn comprend un composant appelé *Slugrunner* qui comme son nom l'indique, permet le déploiement d'applications suivant la même approche que Heroku. Flynn permet de dynamiquement ajouter ou retirer des instances du tiers métier et gère automatiquement leur configuration. Flynn peut être installé sur n'importe quelle plateforme Linux.

Positionnement par rapports aux critères

La discussion des critères pour Deis, Dokku et Flynn est sujette à une évolution rapide car ces trois solutions sont encore peu matures et en phase de développement. Globalement, elles offrent des profils similaires.

- L'*Indépendance applicative* est *Faible* : tout comme pour Heroku, ces solutions ne gèrent que les applications web.
- *Indépendance architecturale Faible* : ces solutions ne gèrent que les applications multi-tiers.
- L'*indépendance vis-à-vis de l'environnement* est *Moyenne* pour Deis en raison de sa spécificité à un ensemble limité de plateformes de cloud ; *Bonne* pour Dokku et Flynn qui tirent pleinement avantage de la non-spécificité de Docker.
- *Gestion des applications patrimoniales* : *Faible*. Toute application doit effectivement être recompilée avant d'être déployée

- La *Granularité des opérations* est évaluée à *Forte* puisque celle-ci est au niveau des composants.
- L'évaluation retenue pour la *Modélisation à l'exécution* est *Moyenne* puisque seuls les services consommés par une application sont décrits.
- En ce qui concerne l'évaluation de la *Modélisation de l'élasticité*,celle-ci est *Moyenne* étant donné que la seule modélisation possible provient de la liste des services composant l'application. Les liaisons et placements sont respectivement implicites et non-exposés.
- L'*Expression de contraintes* est évaluée à *Moyenne*. Il n'est pas possible d'ajouter des contraintes à prendre en compte lors de l'exécution des applications.



Le tracé en ligne continue donne l'évaluation de Dokku et Flynn tandis que le tracé en pointillés illustre celle de Deis.

FIGURE 3.4 – Evaluation des solutions de l'écosystème Docker par rapport aux critères d'étude

3.3.7 Amazon CloudFormation

Amazon CloudFormation [1] est intégré dans Amazon Web Services (AWS), un ensemble de services de cloud computing. Amazon CloudFormation permet de décrire une application et son comportement élastique en utilisant les divers services d'AWS. Chacun de ces services traite des usages identifiés et sont segmentés de la façon suivante :

- Amazon Elastic Compute Cloud (EC2) fournit des machines virtuelles. Cette offre s'appuie sur l'hyperviseur Xen.
- Amazon Elastic Load Balancer (ELB)sert à créer un point d'entrée unique pour un ensemble de composants d'une application répartis dans un ensemble défini d'instances EC2. Ces répartiteurs permettent de gérer les flux HTTP mais ne gèrent notamment pas les flux HTTPS.
- Amazon Auto-Scaling est un service de dimensionnement automatique. Il permet de maîtriser le nombre d'instances EC2 appartenant à un sous-ensemble de l'application. Ce service s'articule avec ELB ainsi qu'Amazon CloudWatch. Il permet de définir des politiques d'élasticité qui dans le cas d'Amazon sont une planification d'opérations de scale-in/out.
- Amazon CloudWatch (ACW) est à la fois un service de prise de décision élastique et un service de surveillance de l'application. Son activation permet l'accès à des sondes permettant de remonter des métriques depuis les instances EC2 et les groupes d'Auto-Scaling. ACW permet d'historiser ces métriques et permet de les consulter soit en direct, soit a posteriori. C'est ce service qui permet le déclenchement de politiques d'élasticité.

Amazon propose également d'autres services de différents niveaux. Ces services peuvent correspondre à des usages très spécifiques :

- Amazon Elastic Block Store (EBS) est une offre de stockage persistant accessible en mode bloc. Il s'agit d'un stockage ne pouvant être accédé que par un seul tenant et uniquement après avoir été monté à la manière d'un disque dur traditionnel. EBS sert de stockage persistant aux instances EC2.
- Amazon Simple Storage Service (S3) est une offre de stockage persistant accessible au travers d'interfaces web ou programmatiques. Contrairement à EBS, il est possible d'accéder aux données au travers d'un protocole HTTP et par plusieurs tenants.
- Amazon RedShift est un service de réplication des données.
- Amazon DynamoDB offre un service de base de données NoSQL sur SSD répliquée en trois dupliquas.
- Amazon Relational Database Service (RDS) est une base de données relationnelle donc des répliquas en lecture peuvent être créés.

- Amazon Simple Queue Service (SQS) est un service de communication par messages entre briques applicatives.

- Amazon Elastic MapReduce (EMR) permet de créer des ensembles de nœds de calculs distribué.

- Amazon Elastic Beanstalk est un service de PaaS permettant de déployer des applications écrites en Node.js, PHP, Java, Python ou .NET. Dans les faits, ce service est issu de la fédération de tous les autres services offerts par Amazon.

- Amazon CloudFront est un service de diffusion de contenu (CDN) à partir de S3 en direction d'un utilisateur

L'offre d'Amazon couvre un large spectre de cas d'usage mais impose cependant certaines restrictions. Il s'agit d'une offre présente aussi bien dans l'industrie que dans les travaux de recherche qui l'utilisent comme modèle technico-économique [20, 34, 82, 153] ou comme plateforme d'évaluation [76, 144]. L'analyse qui suit est focalisée sur CloudFormation qui a l'approche la plus liée aux travaux de thèse exposés dans ce manuscrit.

Modèle d'application

La description des applications et des ressources qu'elles utilisent est réalisé au travers d'Amazon CloudFormation. Amazon CloudFormation s'appuie sur la quasi totalité des services Amazon dont Auto-Scaling EC2, S3 et ELB. Cette description est réalisée au moyen d'un fichier texte au format JSON qui liste des services, leurs interconnexions et leurs configurations. A partir de la description offerte par CloudFormation, les services d'Amazon correspondants sont mis à disposition de l'application lors de son déploiement et de son exécution. Par exemple, Amazon Auto-Scaling conjointement avec ELB permet la gestion d'application multi-tiers au niveau du tiers métier pour les applications web notamment. Les services de bases de données servent alors de tiers persistance. Pour les applications non multi-tiers, il peut exister des services spécifiques à Amazon tels que SQS, RDS, CloudFront ou encore DynamoDB. En dehors de ces services, il est nécessaire de créer soi-même un service à partir de l'offre IaaS EC2.

Opérations d'élasticité couvertes

Les scénarios d'élasticité couverts sont spécifiques à chaque service et sont utilisés en l'état par CloudFormation. La gestion des scénarios d'élasticité est offerte par le service Amazon Auto-Scaling qui permet de définir un nombre minimal et un nombre maximal d'instances EC2 appartenant à un groupe. Amazon Auto-Scaling gère automatiquement le démarrage d'instances supplémentaires lorsque le nombre minimal d'instances est atteint lors de pannes d'instances EC2. Toutes les VMs sont identiques de par leur image virtuelle, leur configuration d'insertion dans l'application et leur profil matériel.

Architecture applicative

L'architecture applicative lors de l'exécution de l'application se résume en la composition de services Amazon. Amazon offre la vision des services consommés mais sans réelle cohérence applicative. En effet, l'interface d'Amazon permet par exemple de savoir quelles sont les instances EC2 en cours d'exécution, à quel ELB elles sont liées mais lors du déploiement de plusieurs applications sur un même compte utilisateur, il n'y a pas de séparation entre applications.

Gestion du placement

La gestion du placement dépend de l'architecture du cloud d'Amazon. Celle-ci comprend sept régions (USA Est, USA Ouest - Oregon, USA Ouest - Californie, USA *GovCloud*, Amérique du Sud, Europe, Asie Singapour, Asie Sydney, Asie Tokyo). Ces régions comprennent elles-mêmes différents zones de disponibilités d'instances EC2. La gestion du placement au sein des services eux-mêmes diffère selon ces mêmes services. Pour RDS, les dupliquas en lecture peuvent être situés dans différentes régions. Pour DynamoDB les répliquas sont placés dans différentes zones de d'une même région uniquement. De même, toutes les instances EC2 d'un groupe Auto-Scaling peuvent être placées dans différentes zones mais à l'intérieur d'une seule région. Enfin, certains services ne sont disponibles que dans certaines zones précises, à l'instar de CloudFront ou de RedShift. Une difficulté d'Amazon est de gérer une application déployée dans plusieurs sites. Sur ce point les services d'Amazon se montrent difficiles à mettre en œuvre. Notamment, ni ELB, ni Auto-Scaling, ni CloudWatch ne gèrent cet aspect. CloudFormation permet d'exprimer des contraintes dans la limite des capacités des services sous-jacents. Il est par exemple possible de décrire explicitement des zones de disponiblité d'exécution pour les services en plus de leur région de déploiement.

Positionnement par rapports aux critères

L'évaluation de CloudFormation est la suivante :

- L'*Indépendance applicative* est *Bonne* puisque CloudFormation propose des services élastiques multiples pouvant être composés entre eux.
- *Indépendance architecturale.* Le constat pour ce deuxième critère est identique à celui du premier. L'évaluation résultante est donc *Bonne.*
- L'*Indépendance vis-à-vis de l'environnement* est *Faible* : CloudFormation ne permet pas de sortir du cloud d'Amazon avec les mêmes fonctionnalités qu'en interne. Il en résulte donc un risque évident d'enfermement propriétaire.
- *Gestion des applications patrimoniales* : l'offre d'Amazon n'est pas parfaitement adaptée aux applications patrimoniales puisqu'en fonction de l'application et de son code propre, il ne sera pas possible d'utiliser les services fournis et il faudra donc en recoder. L'évaluation de ce critère est donc *Bonne.*

- La *Granularité des opérations* est évaluée à *Faible* puisque celle-ci est au niveau de la VM.
- L'évaluation retenue pour la *Modélisation à l'exécution* est *Faible* puisqu'il n'est effectivement pas possible de connaître les services consommés par application : seule une consommation globale par utilisateur est fournie. Il n'y a donc aucune vision architecturale ni même de cloisonnement entre les différentes applications d'un même compte utilisateur.
- En ce qui concerne l'évaluation de la *Modélisation de l'élasticité*,celle-ci est *Moyenne*. CloudFormation ne permet de décrire une application que grâce à une liste de VMs typées et de services de la plateforme. Les placements dépendent des services concernés et leur expression est généralement implicite. Seule est explicite la demande d'un utilisateur de déployer son application en un endroit spécifique ainsi que le profil matériel d'un groupe de VMs.
- L'*Expression de contraintes* est évaluée à *Moyenne*. Il n'y a pas d'expression de contrainte de placement et seules des contraintes sur les cardinalités sont exprimables. Ces contraintes ne sont pas extensibles.

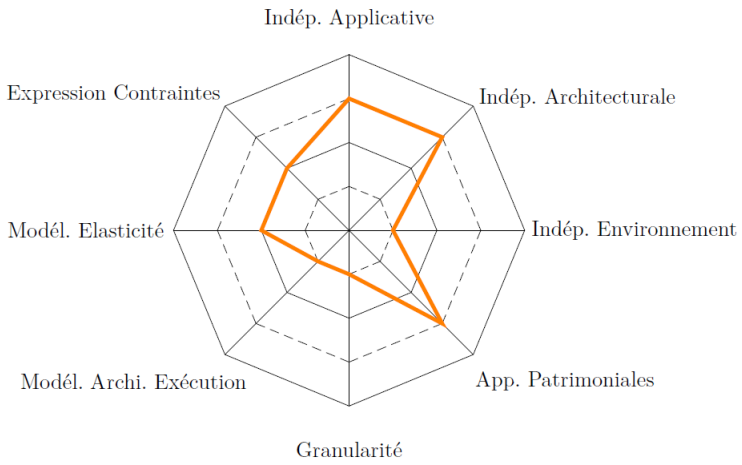


FIGURE 3.5 – Evaluation d'Amazon CloudFormation par rapport aux critères d'étude

3.3.8 Windows Azure

Windows Azure [14] est l'offre de cloud proposée par Microsoft. Cette offre est constituée d'un ensemble de services correspondant tant à la fourniture de VMs qu'à la fourniture d'intergiciels. Ces services se répartissent de la façon suivante :

- Les sites web Windows Azure. Il s'agit d'environnements de déploiement d'applications web Node.js, Java, PHP, ASP.NET, Python et Ruby. De façon globale, toute application acceptée par le serveur Microsoft Internet Information Services (IIS). Ce service vise avant tout le déploiement rapide d'applications en cours de développement. Windows Azure Web Sites s'interface effectivement avec des logiciels de gestion de version ou d'hébergement de code tels que Git, Mercurial, bitbucket ou encore CodePlex.

- Les rôles applicatifs sont des environnements de déploiement permettant un contrôle plus fin à l'utilisateur que les sites web Windows Azure. Ils en existent deux sortes :

  – Les *Web Roles* sont des VMs appelées *instances* dont le système d'exploitation est Windows Server 2012. Ces VMs embarquent un serveur IIS actif. Les *Web Roles* permettent à une application d'exposer une interface web.

  – Les *Worker Roles* sont des VMs dont le but est d'effectuer des traitements pour les applications. Leur usage convient notamment au tiers métier d'une application web multi-tiers. Il ne diffèrent des *Web Roles* que par l'inactivation par défaut de IIS (qui peut malgré tout être réactivé par l'utilisateur).

  Ces rôles applicatifs offrent l'avantage d'être redémarrés aussitôt que la plateforme Azure détecte leur panne. En revanche, il s'agit de VMs sans état : à chaque démarrage, l'image virtuelle est vierge de toute donnée issue d'un démarrage précédent. La persistance doit être assurée par le recours à des services de stockage de la plateforme Azure. Enfin, l'image de ces VMs est maintenue par Microsoft qui ajoute régulièrement des mises à jour.

- Les *Virtual Machines* sont des VMs avec état laissant un contrôle total à l'utilisateur. Elles peuvent aussi bien embarquer un système d'exploitation Microsoft qu'un Linux. Le maintien des images est à la charge de l'utilisateur.

L'ensemble de ces services peut avoir accès à des services de stockage fournis par Windows Azure pour assurer la persistance des données. Les services de stockage comprennent :

- Un service de base de données relationnelle nommé *SQL Database*.

- Un service de base de données non-relationnelle NoSQL. Il s'agit de tables associatives clé-valeur.

- Un service de stockage sous forme de *blobs*. Ce sont des blocs de stockage d'informations binaires qui peuvent permettre de créer des périphériques de stockage embarquant un système de fichier pour les VMs.

Enfin, d'autres services ciblent des besoins plus spécifiques tels que Windows Azure Service Bus, un intergiciel de communication par messages ; HDInsight un service de calcul distribué basé sur Hadoop ; ou encore Windows Azure CDN qui est un service de livraison de contenu.

Modèle d'application

Une application s'exécutant dans le cloud Microsoft Azure est issue de la composition des services offerts par la plateforme.

Microsoft offre un large catalogue de gabarits d'environnements au niveau de son services Web Sites. Ce catalogue contient des gabarits pour des environnements de système de gestion de contenus très populaires tels que WordPress ou Drupal. Ce catalogue contient également des gabarits pour des canevas de prototypage rapide tels que Node.js ou Django.

Opérations d'élasticité couvertes

La gestion de l'élasticité est opérée en trois points distincts. Le premier est intégré à la plateforme (*Autoscale*) tandis qu'un deuxième est un outil additionnel que l'utilisateur doit déployer et configurer lui-même (*WASABi*). Par ailleurs, à l'instar de la base de donnée relationnelle, Windows Azure offre des services élastiques. Toutefois, le contrôle sur ces services est faible puisque géré automatiquement par la plateforme.

Autoscale permet l'élasticité automatique des services Azure. Il offre la possibilité de définir les cardinalités minimales et maximales du nombre de VMs. il permet également de définir des politiques de prises de décision s'appuyant soit sur des métriques de surveillance, soit sur des planifications temporelles. L'équilibrage de charge se fait au niveau des flux réseau.

WASABi est un outil plus ancien qui permet toutefois une plus large gamme d'opérations. Il ne permet cependant de gérer que des rôles applicatifs. De façon globale, il peut être considéré comme un canevas de développement servant à l'élasticité dans Windows Azure. WASABi permet de définir des politiques de décision similaires à celles admises par Autoscale mais permet également à l'utilisateur d'en définir de nouvelles. WASABi permet de définir des priorités à appliquer entre les règles de sorte à pouvoir opérer une conciliation lors de l'exécution. En fonction de la règle à appliquer, des opérations sont planifiées. Ces opérations permettent notamment d'ajouter ou de retirer des VMs mais également de modifier des paramètres applicatifs. Un cas d'usage du dernier point est la mise en place d'un mode dégradé d'une application web lorsque la charge des serveurs est trop importante. Ce scénario est particulièrement adapté pour continuer à offrir un service dans l'attente du démarrage de nouvelles instances.

Architecture applicative

Windows Azure n'offre pas d'architecture applicative à proprement parler, tant durant l'exécution que lors de la spécification de l'application élastique. L'architecture est ef-

fectivement implicite et restreinte aux applications multi-tiers. Une pseudo-vision des liaisons applicatives est proposée au niveau de la spécifications des ports réseau d'entrée et de sortie des rôles applicatifs.

Gestion du placement

Concernant Autoscale, l'utilisateur peut uniquement définir une région d'exécution. L'utilisateur peut également définir les profils des VMs sous-jacentes. Ce profil est fixe, hormis pour les Windows Azure Web Sites qui permettent un redimensionnement dynamique des conteneurs d'exécution à chaud.

Avec WASABi, les mêmes paramètres peuvent être gérés. WASABi ajoute cependant la possibilité d'effectuer des opérations élastiques réparties dans différentes régions. Il se pose alors pour l'utilisateur la problématique de définir et réaliser le déploiement de WASABi au sein de ces régions.

Positionnement par rapports aux critères

- *Indépendance applicative*. Windows Azure permet l'élasticité de plusieurs types d'applications ainsi que de les composer. Ce critère a une évaluation à *Bonne*.

- L'*Indépendance architecturale* de Windows Azure est évaluée à *Bonne* puisque la solution gère plusieurs types d'architectures et permet de les composer.

- L'*Indépendance vis-à-vis de l'environnement* est évaluée à *Faible* car Autoscale et WASABi ne permettent d'accéder qu'au cloud Azure.

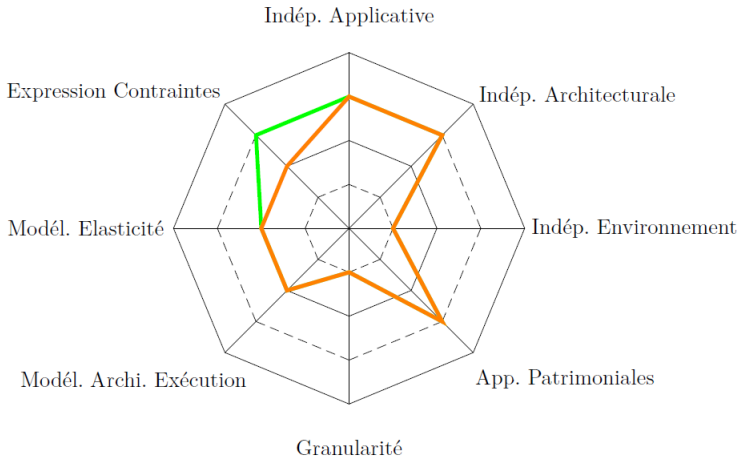
- *Gestion des applications patrimoniales*. La gestion des applications patrimoniales est *Bonne* puisqu'en fonction de l'application, il peut être nécessaire de modifier une partie de l'architecture de l'application de sorte à permettre l'utilisation des services de la plateforme. Il est par exemple nécessaire pour l'application d'être adaptée à l'équilibrage de charge offert par la plateforme (équilibrage matériel sur des ports réseau) ce qui implique le remplacement des éléments de répartition de charge déjà présents dans l'application.

- *Granularité des opérations*. La granularité des opérations se situe au niveau de la VM tant pour Autoscale que pour WASABi. Le critère de granularité est donc évalué à *Faible*.

- Concernant la *Modélisation à l'exécution*, Windows Azure permet de connaitre la consommation en nombre de VMs instanciées pour chaque application. Il n'est fait mention ni de composant, ni de liaison. Ce critère est donc évalué à *Moyen*.

- La *Modélisation de l'élasticité* est évaluée à *Moyenne* en raison du fait que le comportement élastique de l'application est implicitement décrit par la mention de groupes de VMs et de services consommés.

- *Expression de contraintes.* Autoscale et WASABi permettent l'expression de contraintes à base de cardinalités sur le nombre d'instances par type de VMs composant l'application. WASABi se différencie sur ce point d'Autoscale puisqu'il permet d'exprimer des contraintes de placement. Il est possible de répartir des opérations élastiques au sein de différentes régions. Ces contraintes ne sont pas extensibles. L'évaluation de ce critère est *Moyenne* pour Autoscale, et *Bonne* pour WASABi.



Le tracé en Orange donne l'évaluation d'Autoscale. Celle de WASABi est illustrée par le tracé de couleur verte. Il est utile de rappeler que WASABi n'est pas directement accessible à l'utilisateur et que celui-ci doit opérer son déploiement.

FIGURE 3.6 – Evaluation de Windows Azure par rapport aux critères d'étude

3.3.9 Cloud Foundry

Cloud Foundry [4] est une solution open source de PaaS développée par la société VMWare. Cette solution est à la fois accessible en ligne au travers du service cloudfoundry.com et en téléchargement pour une installation sur un cloud privé. Cloud Foundry gère les applications développées en Java, Groovy, Ruby, Node.js et Scala. Cloud Foundry gère effectivement des canevas de développement pour ces langages tels que Node.js, Sinatra, Grails, Ruby on Rails, Spring ou encore Rack. Cloud Foundry vise le déploiement rapide d'une application et s'interface avec l'outil de gestion de version Git. Le déploiement sur Cloud Foundry se résume à pousser le code d'une application depuis le dépôt local à destination de Cloud Foundry avec Git. La plateforme identifie alors automatiquement le type d'application à déployer et offre en conséquence les environnements d'exécution requis. Plus largement, la plateforme offre une interface en ligne de commande *VMC* ainsi qu'un plugin pour Eclipse. Cloud Foundry s'affiche donc clairement comme étant une solution à destination des développeurs.

Au niveau de la virtualisation, Cloud Foundry s'appuie sur *Warden*, un canevas de gestion de conteneurs légers de VMWare. La plateforme peut être installée sur des infrastructures implémentant une interface EC2 ou étant à base d'OpenStack ou de VSphere. Une version de test de présente même sous la forme d'une VM exécutable dans un hyperviseur.

Cloud Foundry supporte différents *services* à disposition des applications dont RabbitMQ, PostgreSQL, Redis, MongoDB ou encore MySQL.

Modèle d'application

Cloud Foundry repose sur l'introspection du code source d'une application et fait correspondre chaque application gérée avec les services qui lui seront nécessaires. Pour cela, la plateforme dispose d'un catalogue de *buildpacks*, des utilitaires de compilation. L'identification du buildpack adapté est réalisée lors du déploiement. A l'aide de ce buildpack, la plateforme compile le code de l'application en un code exécutable en son sein et configure également les conteneurs d'exécution pour qu'ils hébergent les serveurs applicatifs nécessaires. Le catalogue de buildpacks peut être étendu pour supporter de nouveaux canevas ou langages.

La plateforme Cloud Foundry expose des composants appelés *routers* qui routent des requêtes à destination de noms de domaines vers des applications en cours d'exécution. Il en existe actuellement deux implantations aux propriétés différentes. Les *routers* historiques ne gèrent que les protocoles HTTP et HTTPS. Une migration récente vers l'implantation *gorouter* [3] lève cette limitation. Les *routers* font également office de répartiteurs de charge entre les instances des composants applicatifs.

La description d'une application s'effectue dans un fichier de manifeste. Chaque fichier de manifeste permet de décrire :

- les services (au sens Cloud Foundry) requis par l'application. Ces services doivent être pré-existants.

- le domaine associé à l'application.
- la mémoire RAM maximale utilisable par l'application.
- le nombre d'instances de composants applicatifs.

Le fichier de manifeste est détecté automatiquement lors du déploiement de l'application.

Opérations d'élasticité couvertes

Cloud Foundry permet des opérations élastiques sur les applications déployées. Ces opérations visent soit à modifier le nombre d'instances de serveurs applicatifs soit à modifier la taille mémoire maximale allouée à l'application.

Architecture applicative

Cloud Foundry expose des notions de liaisons qui lient les serveurs applicatifs de façon globale aux *services* de la plateforme. Ces liaisons peuvent être créées, modifiées ou détruites dynamiquement au travers de VMC. Il est également possible de consulter quels sont les services à disposition et aussi ceux liés au serveurs applicatifs de l'application.

Gestion du placement

La gestion du placement est totalement transparente pour l'utilisateur. Il n'est effectivement possible que de choisir quelle plateforme Cloud Foundry utiliser pour le déploiement.

Positionnement par rapports aux critères

- L'*Indépendance applicative* de Cloud Foundry est évaluée à *Forte* puisque la solution n'est pas limitée en termes d'applications pouvant être adressées.
- L'évaluation de l'*Indépendance architecturale* de Cloud Foundry est *Faible* puisque la solution ne gère que des applications multi-tiers.
- Grâce à ses possibilités d'installation sur différentes plateformes, Cloud Foundry a une *Indépendance vis-à-vis de l'environnement* qui est *Forte*.
- La *Gestion des applications patrimoniales* de CloudFoundry est *Faible* puisque la plateforme compile nécessairement le code de l'application.
- *Granularité des opérations*. Ce critère est évalué à *Moyen* puisque la finesse des opérations concerne des groupes de composants sans offrir la gestion individuelle des profils matériels de chacun des conteneurs légers.
- Cloud Foundry permet de connaître quels sont les *services* liés à une application et le nombre de serveurs applicatifs. La plateforme n'offre donc pas une modélisation architecturale complète. Le critère portant sur la *Modélisation à l'exécution* est en conséquence évalué à *Moyen*.

- La *Modélisation de l'élasticité* est évaluée à *Moyenne* car Cloud Foundry repose sur l'utilisation de fichiers manifestes ne pemettant qu'une définition des services composant l'application.
- Pour le critère d'*Expression de contraintes*, Cloud Foundry est évaluée à *Moyenne* car la solution ne permet que la mention d'une cardinalité maximale par type de composant ainsi que la RAM pouvant être utilisée. Il n'est en outre pas possible de rajouter des contraintes supplémentaires.

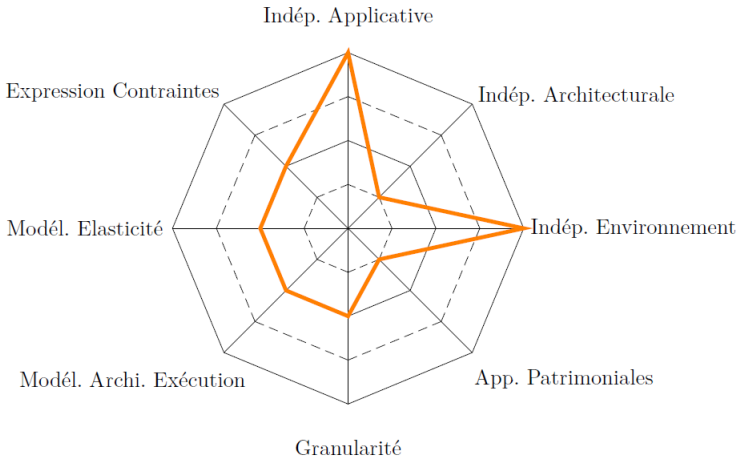


FIGURE 3.7 – Evaluation de Cloud Foundry par rapport aux critères d'étude

3.3.10 soCloud

soCloud [115, 116, 117, 118] est une solution issue des travaux au de l'Université Lille 1. Cette solution vise à adresser plusieurs problématiques du cloud.

La première de ces préoccupations concerne la résistance aux pannes des services fournis dans le cloud. Face aux problèmes rencontrés chez différents fournisseurs, soCloud propose une gestion de la haute-disponibilité basée sur la mise en place automatique de répliquas pour chaque application. Ces répliquas sont répartis dans plusieurs clouds de sorte à palier les problèmes de panne pouvant se produire chez un ou plusieurs fournisseurs.

Une deuxième problématique est la résilience. Celle-ci augmente la première : soCloud propose un mécanisme permettant la résilience en se basant à la fois sur son mécanisme de réplication, mais également par la détection des pannes et le retour vers un état fonctionnel.

Une troisième problématique que soCloud adresse est l'enfermement propriétaire. soCloud opère au sein de différentes offres de cloud de façon uniforme : les applications ne sont ainsi pas liées à un ou plusieurs services spécifiques à une offre. Par ailleurs, cette uniformité permet aux utilisateurs de soCloud de ne pas devoir gérer eux-mêmes différentes APIs et plateformes d'administration.

Enfin, une problématique importante du cloud actuellement, remise en lumière par les récentes actualités relatives à la confidentialité des données est la maîtrise des emplacements d'exécution. Il s'agit pour un utilisateur d'avoir un contrôle total sur la destination géographique lors du déploiement de son application.

SoCloud propose une boucle complète d'élasticité qui s'appuie sur une plateforme multi-cloud. Une originalité de soCloud réside dans sa mise en place transparente de la haute-disponibilité et de la résilience. Afin de mener à bien sa tâche, soCloud utilise les composants principaux suivants :

- Le *Monitoring* est un composant réparti sous la forme d'agents au sein des différentes VMs que compte une application. Il remonte par défaut un ensemble de métriques telles que l'occupation CPU, l'occupation mémoire ou l'utilisation CPU et mémoire par processus. Par ailleurs, ce composant est essentiel afin de repérer les pannes au sein des VMs applicatives.

- Le *Workload Manager* se situe en aval du monitoring. Il a la charge de récupérer et stocker les données du monitoring. Il a également pour rôle d'agréger les données de sorte à pouvoir fournir des indicateurs de dérive. Par exemple, le workload manager peut signaler une occupation CPU trop importante sur les cinq dernières minutes. Ce type d'alerte est ensuite remonté au *Controller*,

- Le *Controller* est le composant de soCloud chargé de la décision. A partir des indications de dérives remontées depuis le workload manager, le controller décide des actions correctrices à mener. Ces actions sont répercutées sur l'application au niveau applicatif et au niveau des clouds par le biais d'actionneurs adaptés.

Le controller offre par ailleurs différents services qui facilitent l'utilisation de soCloud ainsi que l'administration des applications déployées comme un service de géo-localisation qui réalise une répartition géographique des accès. Un autre service est l'éclatement de bases de données relationnelles de sorte à ce qu'elle soit répartie et passe bien à l'échelle. Le controller gère également les problèmes de crédences ou de création de graphiques montrant les métriques du monitoring et les données agréges du workload manager.

- Le *Load Balancer* est le composant qui permet la mise en œuvre pratique de la haute-disponibilité, de la résilience et de l'élasticité. Ce composant gère la répartition de charge et fait office de proxy pour des flux TCP et HTTP. Sa gestion des flux TCP en plus des flux HTTP est clairement à l'avantage de soCloud.

Modèle d'application

Le modèle d'application de soCloud repose sur SCA (Service Component Architecture), le standard du consortium international OASIS relatif aux architectures SOA (Service Oriented Architecture). De façon plus précise, soCloud utilise FraSCAti [135, 136], une implantation des spécifications SCA basée sur le modèle de composants Fractal. FraSCAti offre des capacités de réflexivité à SCA ce qui est un atout important pour son utilisation dans soCloud. FraSCAti adresse la composition de services à destination de composants marqués par une forte hétérogénéité de langages et de canevas. Cette composition est décrite au travers d'un fichier renseignant l'architecture de l'application ainsi que les différentes règles et contraintes d'élasticité.

Le recours à FraSCAti souligne deux limites de la solution. Tout d'abord, la liste des langages et canevas couverts est importante puisque FraSCAti permet à soCloud de supporter les interfaces écrites en WSLD et Java. Les langages et canevas supportés sont Java, Groovy, BPEL, EJB, OSGI, Spring, Xquery, Jython, Jruby, Fscript, Beanshell et Velocity. Cependant, à notre connaissance, il n'est pas possible de gérer des applications écrites par exemple en PHP, Ruby On Rails, ou même Node.js. Une seconde limite est a priori plus gênante : elle concerne la nécessité d'avoir une application ayant une architecture SCA. Il s'agit d'une limite identifiée pour laquelle le recours à des *wrappers* permettrait de conserver les atouts de SCA et FraSCAti au niveau de la gestion des composants et services tout en repoussant cette limite. De fait, cette même limite peut être rédhibitoire à l'égard d'applications ne pouvant pas être modifiées.

Opérations d'élasticité couvertes

soCloud gère une gamme très complète d'opérations d'élasticité qui incluent à la fois les opérations de croissance et décroissance horizontale, mais également les opérations de croissances et décroissances verticales. Il s'agit là d'un point fort de la solution clairement différenciateur par rapport à d'autres solutions existantes. soCloud gère ainsi des opérations de *scale out* sur un type de composant. L'opération inverse consiste en un *scale in* sur ce même type de composant. Il est à noter que la seconde opération est

exactement l'inverse de la première : elle permet de faire revenir l'application dans l'état exact d'avant le *scale out*. Il en est de même pour les opérations d'élasticité verticales.

Architecture applicative

Nous n'avons pas identifié de point d'accès ou d'API offrant une vision de l'architecture de l'application durant son exécution. Cette limite peut, par exemple, être gênante pour un administrateur ou un développeur voulant aller consulter les journaux système présents sur les VMs de l'application à des fins de debug. En revanche, un point fort de soCloud réside dans la modélisation de l'élasticité d'une application. La solution introduit effectivement une description permettant de mentionner des règles d'élasticité et les modalités d'exécution des opérations élastiques. Les règles d'élasticité permettent de spécifier les décisions prises par soCloud à partir de l'expression des actions à entreprendre en cas de dépassement de seuils. Ces règles ont l'avantage d'être extensibles et peu verbeuses. Par ailleurs, le modèle proposé dans soCloud permet à l'utilisateur de spécifier sous la forme d'annotations, les modalités de réalisation des décisions au travers d'un Domain Specific Language (DSL) non-extensible.

Gestion du placement

Les annotations proposés dans soCloud permettent une gestion très fine des placements : il est possible de spécifier le profil matériel des VMs à utiliser, des cardinalités maximales, le nombre de répliquas ou encore des emplacements géographiques précis pour le déploiement des composants de l'application. Là encore, il s'agit d'un point fort pour soCloud.

Positionnement par rapports aux critères

L'évaluation de soCloud montre qu'il s'agit d'une solution très complète par rapport aux critères considérés.

- L'*Indépendance applicative* est *Forte*. soCloud ne souffre d'aucune limite en matière de gestion de types d'applications. Il faut toutefois modérer cette évaluation face à l'absence de gestion de certains langages et canevas populaires. Un point intéressant de soCloud réside dans son load balancer : alors que ce type de composant introduit des limites sur ce critère pour d'autres solutions étudiées dans ce manuscrit, ce n'est pas le cas ici.

- L'*Indépendance architecturale* est évaluée à *Faible*. L'évaluation de ce critère provient de la spécificité de soCloud au type d'architecture SCA et face à la faible utilisation de ce standard. Une couche de wrappers permettrait de faire remonter l'évaluation de ce critère à *Forte*.

- L'*Indépendance vis-à-vis de l'environnement* est *Forte*. Il s'agit évidemment d'un des gros points forts de la solution puisqu'aucune limite n'est notée en matière de

capacité d'accès à quelque cloud que ce soit. Par ailleurs, soCloud gère le déploiement d'une même application au sein de plusieurs clouds en même temps.

- *Gestion des applications patrimoniales* : la gestion des seules applications ayant une architecture SOA est une limite identifiée. Une application ne pouvant être modifiée ne peut être gérée par soCloud, ce critère est évalué à *Faible*. Ici encore, l'introduction de wrappers offrirait potentiellement une évaluation à *Forte*.

- La *Granularité des opérations* est évaluée à *Bonne* puisque soCloud permet des opérations d'élasticité verticale sans toutefois offrir une finesse au niveau du composant.

- L'évaluation retenue pour la *Modélisation à l'exécution* est *Faible* car il ne semble pas possible pour un utilisateur de la solution d'avoir connaissance de l'architecture de son application en cours d'exécution.

- Concernant l'évaluation de la *Modélisation de l'élasticité*,celle-ci est *Forte* grâce à une expression explicite complète.

- L'*Expression de contraintes* est évaluée à *Bonne* en raison de la possibilité d'exprimer des contraintes complexes telles que le placement, le nombre de répliquas ou le profil matériel des VMs. Il faut toutefois noter que les contraintes d'élasticité exprimées sous la forme d'annotations ne sont apparemment pas extensibles. De même les notions de cardinalités sur les liaisons semblent restreintes.

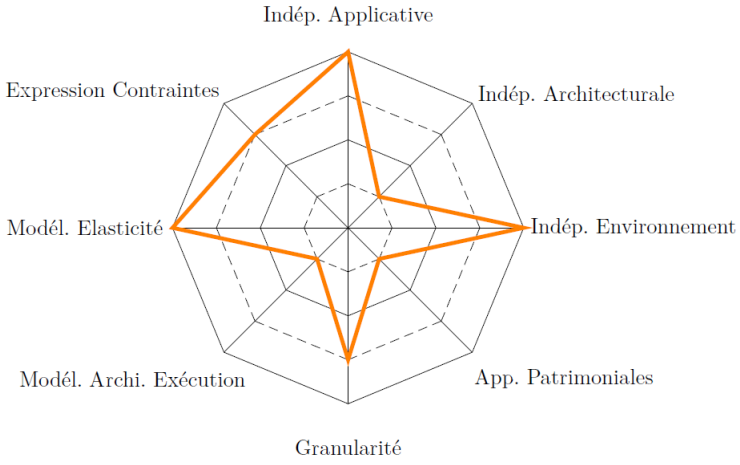


FIGURE 3.8 – Evaluation de soCloud par rapport aux critères d'étude

3.3.11 ElaaS

ElaaS (**El**asticity**a**s**aS**ervice) [90] est une solution qui vise à rendre l'élasticité autonome. Il s'agit d'un canevas issu des travaux de l'Université de Nationale Technique d'Athènes qui s'inscrit comme une contribution du projet Européen 4CaaSt [60].

ElaaS répartit le traitement de l'élasticité en différents modules. Ces modules sont au nombre de cinq :

1. L'*ElaaS Core* est le module en charge de coordonner l'activité globale. Pour cela, l'ElaaS Core initialise les autres modules lors de la phase d'initialisation du canevas en fixant les paramètres adéquats, de même qu'en établissant les canaux de communication inter-modules. Il est crée également tous les éléments de stockage de données requis pour le fonctionnement des modules. Après la phase d'initialisation, l'*ElaaS Core* orchestre les échanges entre les modules.

   C'est le seul module qui ne soit pas extensible (au moyen du chargement de plugins).

2. L'*Application Manager* récupère la description de l'application lors de la phase d'initialisation ainsi que la description des SLAs et les KPIs à surveiller.

3. Le *Monitoring Manager* est en charge de la communication avec les sondes. Celui-ci est configuré pour ne consommer que les KPIs connues de l'*Application Manager*. Il gère un sous-module de stockage de données de sorte à créer et maintenir un historique des relevés des KPIs. Cet historique est primordial pour permettre les prises de décision d'élasticité.

4. Le *Business Logic Manager* est le module de prise de décision d'ElaaS. Il n'inclut pas de logique de prise de décision qui reste à la charge du développeur. Cette approche est justifiée par le fait qu'ElaaS a pour vocation d'être un canevas de prise de décision d'élasticité indépendant de l'approche retenue (i.e. pro-active ou réactive).

5. L'*Action Manager* est un module dont le rôle est triple. Il s'agit en premier lieu d'un passe-plats entre les décisions du *Business Logic Manager* et les composants applicatifs ou les éléments de la plateforme de cloud sous-jacente. L'*Action Manager* est également une brique à même de pouvoir compléter des décisions afin d'élaborer des séquences de reconfigurations à opérer. Par exemple, si la décision de dupliquer un serveur d'application est prise, il ajoutera automatiquement un répartiteur de charge en frontal. Le *Business Logic Manager* est aussi en charge d'établir un graphe d'état cible de l'application (i.e. une architecture cible) appelé *graphe de déploiement*.

Les interactions avec ElaaS sont rendues possibles au moyen de web services. Parmi ces interactions se trouvent celles relatives à l'exposition de chaque instance du canevas ElaaS comme un service élastique auprès d'autres instances : ElaaS a cette particularité de viser l'élasticité offerte comme un service à disposition d'englobants eux-mêmes

élastiques. Par ailleurs, ElaaS se base sur la solution de déploiement *Smartfrog* [64] pour appliquer les *actions* de l'*Action Manager*.

Modèle d'application

ElaaS propose la modélisation des applications en deux modèles de niveaux différents. Le *graphe d'application* est un modèle dit de "haut-niveau" qui est décliné en un second appelé *graphe de déploiement*. Plus précisément, le graphe d'application renseigne l'ensemble des dépendances d'une application (par exemple, un fichier .war est exécuté dans un serveur d'application Glassfish et requiert une base de données MySQL). Le graphe d'application utilise une description à gros grain listant les logiciels requis et les artefacts applicatifs.

Durant le déploiement, le graphe d'application est raffiné pour obtenir le premier graphe de déploiement de l'application. Après le déploiement initial, seul le graphe de déploiement est modifié en fonction des logiques implantées dans les modules *Business Logic Manager* et *Action Manager*.

Opérations d'élasticité couvertes

Les opérations d'élasticité permises dépendent des logiques implantées dans les modules mais aussi des possibilités. ElaaS n'introduit pas de limitation quant aux possibilités d'opérations d'élasticité. Cependant celles-ci restent à la charge de l'administrateur de l'application.

Architecture applicative

ElaaS offre une vision de l'architecture applicative en deux niveaux, au travers des graphes d'application et de déploiement. Le premier graphe de déploiement est issu du graphe d'application, tandis que les suivants proviennent directement de modifications successives appliquées sur les graphes de déploiement courants. Le graphe d'application offre donc une vision à haut-niveau pour le premier déploiement, sans toutefois couvrir des notions de contraintes. Le comportement élastique de l'application est à la fois décrit par le graphe d'application en plus d'être réparti dans les implantations des modules. De son côté, le graphe de déploiement offre la vision de l'architecture durant l'exécution de l'application.

Gestion du placement

De même que pour la couverture des opérations d'élasticité, la gestion du placement est liée à l'implantation des modules d'ElaaS. Il est par exemple possible de gérer du multi-cloud mais cela implique de modifier l'implantation du module appelé *Business Logic Manager*.

Positionnement par rapports aux critères

ElaaS est un canevas d'élasticité à destination d'application qui peut être spécialisé au travers de plugins. Le positionnement par rapport aux critères qui suit a été réalisé en prenant en compte l'état connu de l'implantation.

- L'*Indépendance applicative* est *Forte* car ElaaS ne semble présenter aucune limite en matière de gestion de types d'applications
- L'*Indépendance architecturale* est évaluée à *Faible* car en l'état actuel d'implantation de ses plugins, ElaaS n'adresse que des applications multi-tiers.
- L'évaluation de l'*Indépendance vis-à-vis de l'environnement* est *Bonne* puisque grâce à SmartFrog, ElaaS ne semble pas restreint quant aux clouds accessibles. En revanche, l'utilisation simultanée de plusieurs IaaS pour une même application ne semble pas possible.
- *Gestion des applications patrimoniales* : ElaaS utilisant smartFrog, les applications patrimoniales sont correctement adressées sans contrainte de modification. L'évaluation pour ce critère est donc *Forte*.
- La *Granularité des opérations* est évaluée à *Faible* puisqu'en l'état actuel de son implantation rien n'indique qu'ElaaS raffine la granularité VM de SmartFrog.

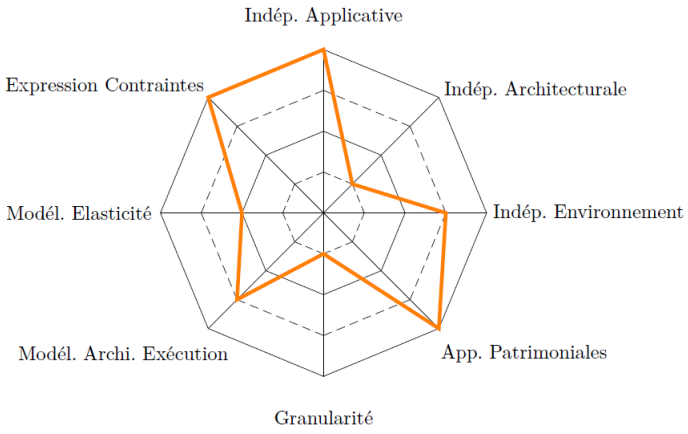


FIGURE 3.9 – Evaluation d'ElaaS par rapport aux critères d'étude

- L'évaluation retenue pour la *Modélisation à l'exécution* est *Bonne*. Le Deployment Graph mentionne effectivement les liaisons entre composants mais toutefois, ce n'est pas le cas des placements.
- Pour le critère de la *Modélisation de l'élasticité*, l'évaluation obtenue est *Moyenne*. Seuls les types de composants de l'application sont mentionnés dans l'Application

Graph. Le reste du comportement élastique est décrit dans le code des plugins et cela n'est pas considéré comme une mention explicite et centralisée en un seul modèle.

- L'*Expression de contraintes* est évaluée à *Forte*. ElaaS permet l'expression de contraintes d'élasticité au travers de l'implantation de plugins pour ses modules. ElaaS permet d'étendre les contraintes gérées. Toutefois, cette évaluation est à nuancer : l'approche par implantation de plugins peut devenir une tâche complexe lorsque le nombre de contraintes à satisfaire augmente et ElaaS ne semble disposer d'aucun mécanisme permettant de gérer les conflits pouvant apparaître.

3.3.12 Application Deployment Toolkit (ADT)

Application Deployment Toolkit (ADT) [83] est une solution issue des travaux menés au sein de l'Université de Paderborn en Allemagne. ADT permet d'automatiser le déploiement et l'élasticité d'applications. ADT constitue une approche tout-en-un composée de cinq modules :

- Le *Topology Module* (TM) assure le maintien de la connaissance des différentes plateformes de cloud à disposition. Pour cela il interroge régulièrement les plateformes connues pour connaître leur état. C'est également lui qui permet la gestion des problématiques éventuelles liées au réseau d'une application. En l'état actuel, le TM permet uniquement de connaître l'état du réseau et pas de le maîtriser.

- Le *Steering Module* (SM) crée une communication bidirectionnelle entre ADT et l'application. Le SM permet d'opérer des (re)configurations au niveau des binaires applicatifs : chaque composant peut ainsi être commandé individuellement. Le SM sert également à remonter des KPI de façon régulière de sorte à rendre possible la prise de décision.

- Le *Resource Module* coordonne le déploiement de ressources. Son rôle consiste tout d'abord à gérer les IaaS en permettant la création, le maintien et la suppression de VMs. Il permet également de connaître l'état courant global de l'application (VMs et composants applicatifs).

- L'*Adaptation Module*. ADT pouvant gérer plusieurs applications concomitamment, chacune est gérée par un *Adaptation Module* (AM). C'est l'AM qui intègre toute la logique de l'élasticité d'une application (placements, décision, optimisations, contraintes). Chaque AM est spécifique à des ensembles d'applications et de son implantation dépend les fonctionnalités d'élasticité possibles. L'AM dialogue à la fois avec le TM, le RM et le SM. Chaque AM est créé et supervisé par le RM.

ADT présente une interface graphique offrant la possibilité à l'utilisateur de fournir un modèle d'application puis de suivre l'évolution de son application.

Modèle d'application

ADT propose un modèle en deux niveaux : un de haut-niveau d'avant déploiement et un autre représentatif de l'état de l'application durant son exécution.

- L'*Application Template* est le modèle de haut-niveau d'ADT. Il repose sur un formalisme proche d'UML constitué d'objets représentant des templates d'éléments de l'application et de liaisons entre eux. Ces liaisons sont orientées en fonction des communications entre les composants applicatifs et sont décorées au niveau de chaque extrémité par un attribut numéral. Cet attribut est une cardinalité maximale pour le template situé à cette extrémité par rapport à chaque instance du template de l'autre extrémité.

- L'*Allocation Graph* est la représentation de l'état courant de l'application ainsi que de son architecture-cible. L'*Allocation Graph* est en partie visible dans l'interface graphique fournie par ADT. Ce graphe comporte des éléments reliés et annotés. Les liaisons représentent les communications entre éléments applicatifs tandis que les annotations mentionnent les divers paramètres de ces éléments.

Le déploiement d'une application s'effectue par le renseignement de la *description de déploiement* de l'application. Celle-ci comprend l'*Application Template*, l'identifiant de l'AM à utiliser et des données dites d'"adaptation" permettant à l'AM d'adapter son fonctionnement à l'application.

Opérations d'élasticité couvertes

Les opérations d'élasticité couvertes sont dépendantes de l'implantation de l'AM mais également des fonctionnalités des IaaS sous-jacents. Typiquement, ADT permet théoriquement les opérations d'élasticité verticale mais un IaaS comme Amazon ne le permet pas. Par ailleurs, ADT permet la gestion à un grain très fin des applications (ex : paramètres des binaires applicatifs), mais toutefois, la gestion de l'élasticité horizontale reste au grain de l'ajout/retrait de VM.

Architecture applicative

L'état courant de l'application est connu au travers de l'*Allocation Graph* contenu dans le RM. Cet état peut être demandé par l'AM de l'application qui peut alors le modifier et retourner au RM l'état-cible. Le RM décompose alors les changements à opérer en actions de (re)configuration au niveau du IaaS et de l'applicatif. Les modifications possibles d'un état courant sont contraintes par l'*Application Template* au travers des cardinalités renseignées. Une restriction de ces modèles porte sur la mise à plat des composants applicatifs et des VMs. En effet, l'*Allocation Graph* et l'*Application Template* ne mentionnent pas des composants applicatifs mais des VMs typées, c'est-à-dire des VMs embarquant un ensemble prédéfini de composants applicatifs.

Gestion du placement

Le placement est adressé par l'implantation des AMs. Ceux-ci intègrent effectivement la logique inhérente à cette problématique hormis le placement des composants applicatifs dans les VMs. Ce dernier aspect est implicite puisque les VMs sont typées. La gestion du placement n'est donc pas intégrée à l'Application Template. En revanche l'Allocation Graph mentionne les nœuds/clouds servant à l'exécution des VMs de l'application.

Positionnement par rapports aux critères

ADT est un canevas d'élasticité qui peut être spécialisé au travers de l'utilisation d'une implantation spécifique de l'AM. Le positionnement par rapport aux critères qui suit a été réalisé en prenant en compte l'état connu d'ADT.

- *Indépendance applicative.* ADT n'est pas spécifique à une application ou à un ensemble défini d'applications : son indépendance applicative est donc *Forte.*
- *Indépendance architecturale.* ADT n'a pas de restriction connue relative à la gestion des architectures applicatives. L'évaluation de ce critère est donc *Forte.*
- L'*Indépendance vis-à-vis de l'environnement* est évaluée à *Bonne* puisque si aucune restriction n'est présente au niveau des IaaS accessibles via ADT, le déploiement d'une application ne peut être réalisé qu'au sein d'un seul cloud.
- *Gestion des applications patrimoniales.* La gestion par ADT des applications patrimoniales ne présente aucune restriction. Ce dernier critère reçoit en conséquence une évaluation à *Forte.*
- *Granularité des opérations.* Seules des VMs typées sont utilisées : la granularité est donc au niveau de la VM. Le critère de granularité est donc évalué à *Faible.*

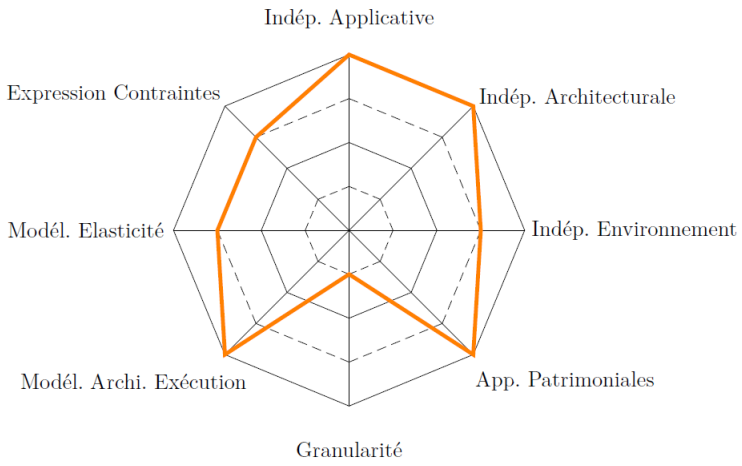


FIGURE 3.10 – Evaluation d'ADT par rapport aux critères d'étude

- Concernant la *Modélisation à l'exécution*, l'état courant de l'application est totalement connu au travers de l'Allocation Graph : sur ce critère ADT est évalué à *Fort.*
- La *Modélisation de l'élasticité* est évaluée à *Bonne* . Une partie du comportement élastique est modélisée par l'Application Template mais cependant, les placements et les configurations sont directement implantés dans l'Adaptation Module propre à chaque application.
- *Expression de contraintes.* L'Application Template permet l'expression de contraintes de connexions entre éléments applicatifs avec la possibilité de décrire des motifs complexes. Les autres contraintes sont implantées dans l'Adaptation Module. Il ne semble pas possible d'étendre ces contraintes. L'évaluation résultante est *Bonne.*

3.4 Synthèse

Cette première partie s'achève après avoir commencé par décrire le contexte des travaux de thèse : il s'agit du cloud computing, un environnement d'exécution marqué par sa jeunesse et son évolution rapide.

Dans un second temps, cette partie a dressé un état de l'art à propos de l'élasticité automatique des applications. Pour cela, douze solutions spécialisées dans la prise de décision ont tout d'abord été décrites afin de déterminer quelles en sont les décisions possibles. Ce point est important car les travaux de thèse se situent en aval de ces solutions.

Ensuite, treize solutions gérant une boucle complète d'automatisation ont été étudiées suivant huit critères. La figure 3.11 montre un récapitulatif des évaluations de ces solutions.

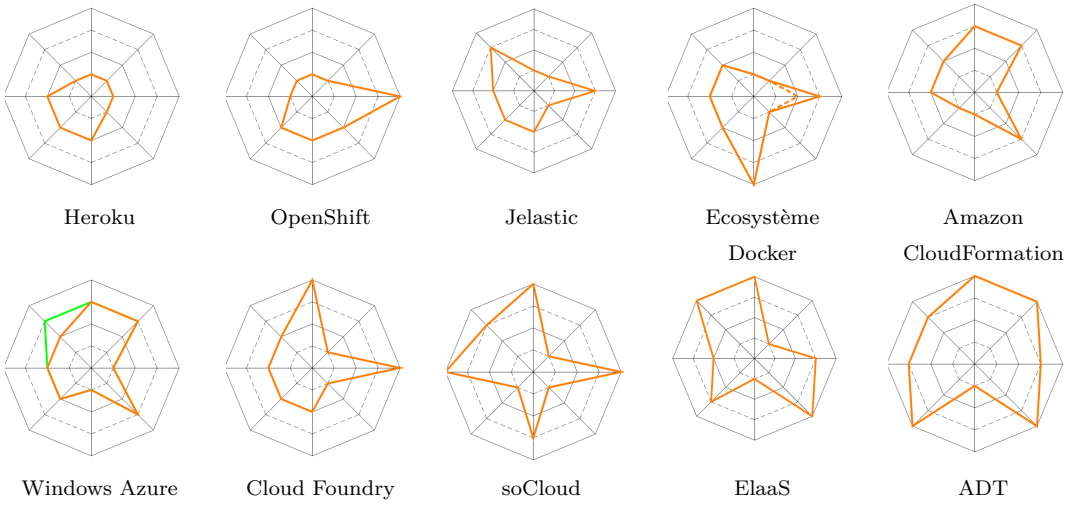
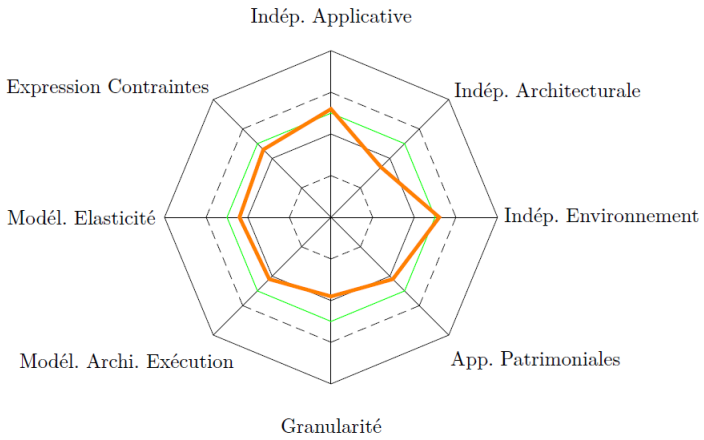


FIGURE 3.11 – Récapitulatif de l'ensemble des évaluations des différentes solutions complètes étudiées

La figure 3.12 montre la moyenne obtenue sur le panel des solutions présentées. Seuls deux critères obtiennent (de peu) une évaluation globale supérieure à la moyenne : l'*Indépendance applicative* et l'*Indépendance vis-à-vis de l'environnement*. Les évaluations globales les plus faibles sont obtenues pour les critères d'*Indépendance Architecturale* et de *Granularité des opérations*.

La faiblesse du critère concernant l'*Indépendance Architecturale* s'explique par le fait que certaines solutions sont très clairement spécialisées pour adresser certains besoins. C'est notamment le cas des solutions Heroku, OpenShift, Jelastic et Cloud Foundry qui sont ostensiblement tournées vers les développeurs d'applications web multi-tiers. Leur succès montre par ailleurs que le besoin qu'elles adressent est un vrai cas d'usage qu'il faut prendre en compte.

A contrario, OpenShift, Cloud Foundry, Jelastic et Heroku sont assez mal adaptées



La ligne orange montre la moyenne obtenue sur l'ensemble des solutions, tandis que la ligne verte sert de repère pour identifier ce qui correspond à l'évaluation moyenne sur chaque critère (i.e. l'équivalent de 10/20)

FIGURE 3.12 – Evaluation moyenne des solutions étudiées

aux besoins d'un administrateur en environnement de production : l'abstraction de ces plateformes ne permet pas de maîtriser l'ensemble de l'environnement d'exécution et des fonctionnalités comme la gestion des placements font défaut. Le passage en production pour des applications développées sur ces plateformes peut par ailleurs devenir problématique en raison du changement d'environnement nécessaire. Un contrôle total doit donc être offert à l'utilisateur afin de correspondre à l'ensemble de ses besoins, qu'il soit développeur ou administrateur en environnement de production. Ce constat est du reste abordé par [24, 59].

Néanmoins, la faiblesse globale du critère concernant la *Granularité des opérations* illustre toute la problématique actuelle que pose un contrôle total sur l'application : beaucoup de solutions ne gèrent que des opérations à la granularité de la VM par mimétisme avec les IaaS alors qu'une finesse supérieure est non seulement possible mais également totalement justifiée au regard des capacités décrites des solutions de prise de décision. La gestion des applications avec des concepts de la couche IaaS a donc clairement un effet négatif sur les possibilités d'administration de ces mêmes applications. Ce constat rejoint celui exposé dans [159] : il existe une adhérence trop forte entre les solutions de gestion des applications et les IaaS, ce qui dans les faits provoque une remontée dans les couches supérieures des limites d'administration de la couche IaaS.

Les travaux de thèse exposés dans ce manuscrit visent à aller dans le sens d'un contrôle total laissé à l'utilisateur. Nous pensons que cela passe par l'émancipation de la réaction dans la boucle d'automatisation de l'élasticité, vis-à-vis des IaaS, des applications et des architectures. Pour l'utilisateur, comme ce contrôle doit être choisi et non subi, nous pensons qu'il est nécessaire de garantir à chaque utilisateur la connaissance de son application, à tout moment de son exécution. Outre cette connaissance, nous pensons également que le moyen doit être laissé à l'utilisateur d'exprimer le comportement élastique de son application, et plus particulièrement en lui permettant d'exprimer ses

exigences au travers de contraintes à respecter pour l'architecture de l'application.

Enfin, une préoccupation également importante concerne la gestion des applications patrimoniales, à l'heure où beaucoup d'utilisateurs du cloud font part de difficultés relatives à la migration d'applications vers le cloud. Nous pensons effectivement que cette problématique doit être adressée, mais évidemment, dans la limite des capacités intrinsèques des applications à adopter l'élasticité.

La suite de ce manuscrit décrit l'approche retenue qui maximise les évaluations pour chaque critère d'étude.

Deuxième partie

Contributions

Chapitre 4

Vue d'ensemble de la solution

Sommaire

4.1 Rappel des objectifs

Les objectifs des travaux de thèse exposés dans ce manuscrit sont au nombre de trois :

1. Définir un modèle architectural et un formalisme devant permettre la description d'applications élastiques dans le cloud. L'approche retenue doit être générique.
2. Définir et prototyper les mécanismes devant permettre l'élasticité. Il s'agit de permettre le calcul d'architectures-cibles et de gérer la répercussion de cette architecture sur l'application.
3. Valider l'approche par des cas d'usage. Cet objectif très concret consiste en l'identification d'exemples d'applications dont il faut gérer l'élasticité. La validation est évidemment primordiale afin d'identifier les besoins et de ne pas adresser des "non-cas d'usage", c'est-à-dire des cas d'usage non représentatifs.

4.2 Automatisation de l'élasticité

Le but principal de l'élasticité automatique est de permettre une adaptation des ressources consommées par l'application en fonction de la charge. L'élasticité permet d'op-

timiser les coûts d'exploitation de l'application grâce à des montées en charge ou inversement à des adaptations à la baisse qui sont rapides. Pour cela, il faut donc adapter **dynamiquement** l'architecture de l'application en fonction de la charge. Seule une adaptation complètement **automatisée**, c'est-à-dire sans intervenant humain, est à même d'utiliser efficacement les avantages du cloud. L'application est ainsi administrée non plus par un être humain mais par un processus **adaptatif** constitué d'une succession d'opérations de surveillance, de calculs et de reconfigurations. Afin que l'architecture soit toujours optimale, ce processus adaptatif doit être **réalisé de façon continuelle** de sorte à suivre les besoins de l'application quasiment en temps "réel".

La gestion automatisée de l'élasticité requiert donc un processus adaptatif automatique qui répète continuellement des successions d'opérations. L'*Autonomic Computing* propose de modéliser un tel processus par la boucle MAPE-K [84] [1]. Le schéma 4.1 illustre l'adaptation de la boucle MAPE-K à la gestion automatisée de l'élasticité.

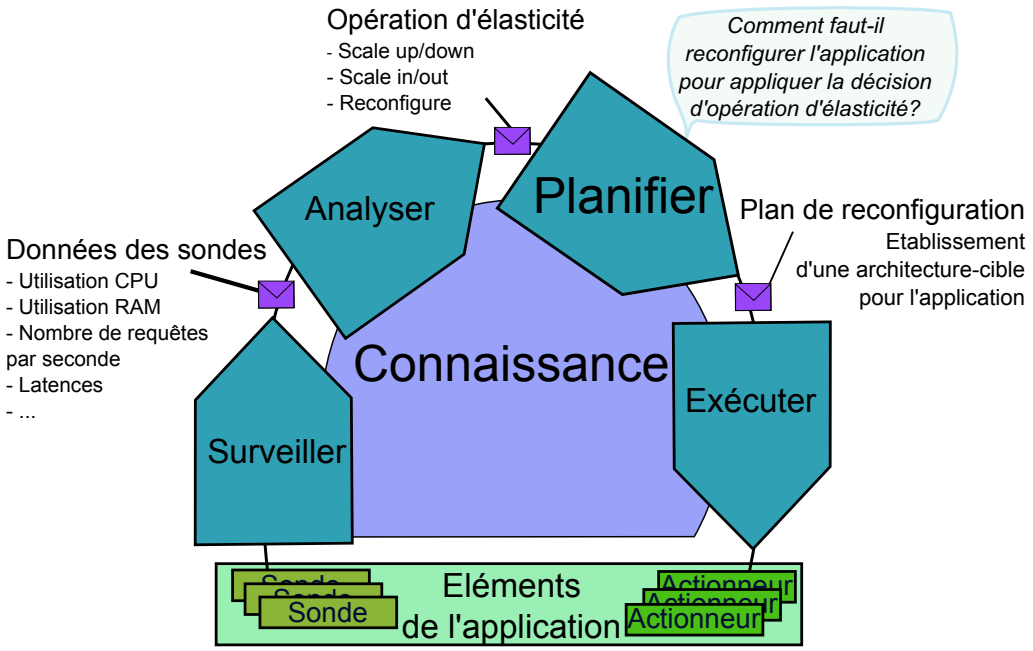


FIGURE 4.1 – La boucle MAPE-K appliquée à la gestion automatisée de l'élasticité

MAPE-K est un cycle dont la finalité est la gestion de ressources de façon automatique. Dans le cas de l'élasticité des applications dans le cloud, ces ressources sont les éléments constitutifs de l'architecture d'une application. Les ressources sont pilotées par des *actionneurs* et observées par des *sondes*. Les *actionneurs* et les *sondes* permettent la mise en place des quatre étapes de la boucle MAPE-K :

1. La *Surveillance* remonte, centralise et agrège les données des sondes. Ces données

[1] Monitoring, Analysis, Planification, Execution - Knowledge

permettent notamment de connaître l'état des différents éléments d'une application. Les données agrégées sont ensuite exposées à l'étape suivante appelée *Analyse*.

2. L'*Analyse* est une étape de prise de décision : en fonction de l'état de l'application et de sa charge, l'*Analyse* décide d'une opération d'élasticité. Cette décision décrit une modification de l'architecture de l'application qui ne suffit pas à elle seule à déterminer une nouvelle architecture. La décision consiste effectivement en une méta-modification de l'application qui implique d'être raffinée afin d'obtenir une nouvelle architecture-cible. C'est le rôle de la *Planification*

3. La *Planification* est l'étape durant laquelle un plan de reconfiguration de l'architecture de l'application va être établi. La *Planification* permet d'obtenir une nouvelle architecture-cible à partir de l'architecture courante en appliquant la décision de l'*Analyse*. Cette étape est au cœur des travaux exposés dans ce manuscrit. Finalement, la nouvelle architecture est fournie à la quatrième et dernière étape de la boucle MAPE-K.

4. L'*Exécution* est l'étape durant laquelle la nouvelle architecture de l'application est effectivement mise en place au niveau des éléments de l'application.

Tout au long de la boucle, les étapes lisent et enrichissent la *Connaissance* du système.

Des approches antérieures ont déjà visé à utiliser la boucle autonomique MAPE-K pour permettre la modification d'architectures applicatives. Ce fut le cas de Rainbow [46, 45]. Cependant, cette solution n'est pas adaptée au cloud. Nous pensons qu'une solution permettant la modification d'architectures applicatives à des fins d'élasticité dans le cloud est primordiale. Plus particulièrement, nous pensons que des avancées peuvent être apportées grâce à une planification innovante.

La suite de ce manuscrit se focalise sur la troisième étape de la boucle MAPE-K , la *Planification*. Comme l'a montré l'état de l'art, il s'agit d'une problématique qui est actuellement très peu adressée. La section suivante détaille le rôle de la planification afin de comprendre en quoi cette étape est primordiale.

4.3 Rôle de la Planification

La planification a pour principal objectif d'appliquer la méta-modification correspondant à la décision de l'analyse sur l'architecture courante de l'application, pour au final établir un plan de reconfiguration. **Le rôle de planification est de contextualiser la décision de l'analyse à l'architecture courante de l'application** tout comme la décision de faire aller tout droit le chat dans le programme *Acoustic Kitty* aurait dû l'être pour traverser la route.

L'établissement d'un tel plan de reconfiguration est un processus complexe puisqu'il faut adresser l'ensemble des problématiques de l'architecture de l'application. Comme le montre la figure 4.2, ces problématiques sont nombreuses. Elles ne se résument pas à de simples arrêts ou démarrages de VM. Afin de comprendre la complexité du rôle de la planification, la suite de cette section s'appuie sur l'exemple illustré par la figure 4.2, qui

correspond à un scénario de *croissance horizontale* sur le tiers métier de l'application web trois tiers Springoo.

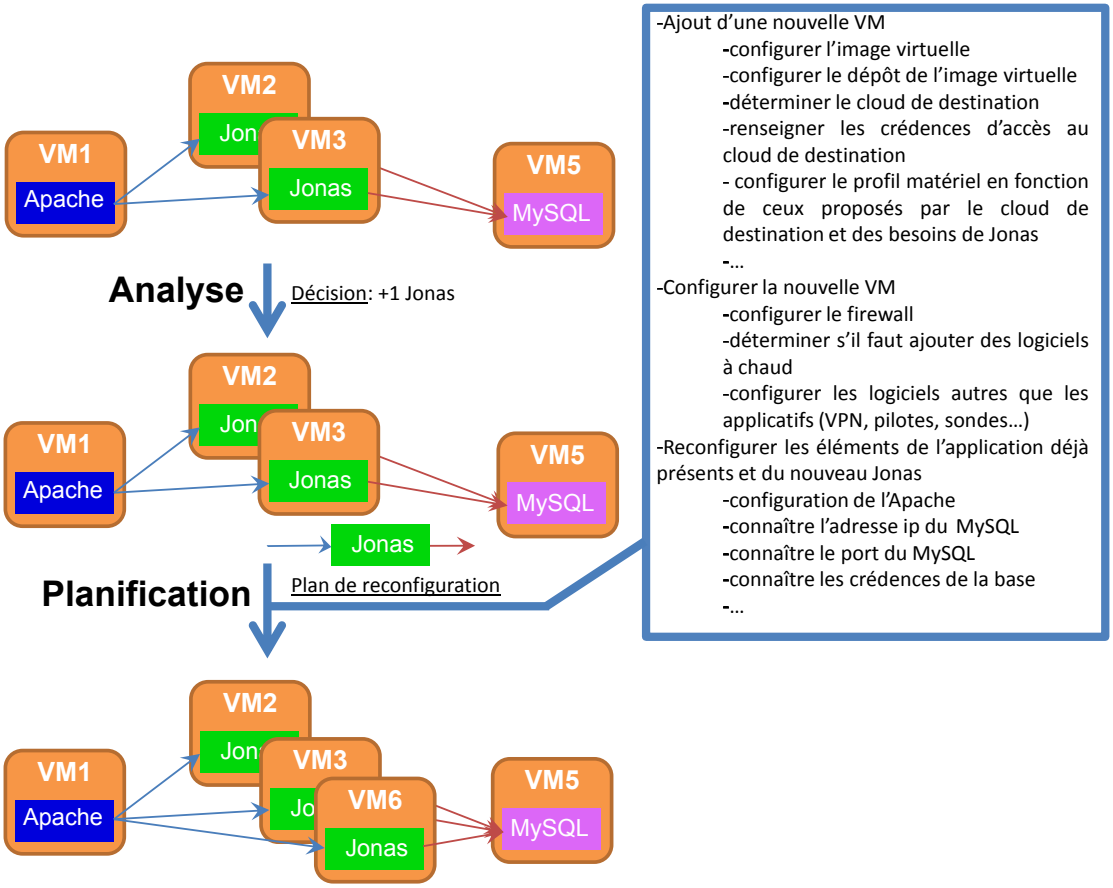


FIGURE 4.2 – Explicitation du rôle de la planification au travers d'une *croissance horizontale* de l'application Springoo

Dans cet exemple, la décision a été prise par l'analyse d'ajouter un serveur Jonas, c'est-à-dire d'ajouter une instance au tiers métier. Cette décision n'est pas suffisante pour que l'architecture de l'application soit correcte. En effet, la nouvelle instance de serveur Jonas n'est non seulement pas placée au sein d'une VM, mais elle reste en outre non-connectée au reste de l'application. **La décision de l'analyse est une méta-modification qui nécessite des calculs supplémentaires pour obtenir une nouvelle architecture-cible complète**. Dans cet exemple, la planification va donc consister à établir un plan de reconfiguration mentionnant l'ajout d'une VM pour y exécuter la nouvelle instance de serveur Jonas ainsi que la façon d'inclure cette nouvelle instance dans l'application. Pour cela, la planification va :

- Spécifier l'ajout d'une nouvelle VM. La planification va établir dans quel cloud la nouvelle VM doit être démarrée. Ce choix n'est pas immédiat puisqu'en fonc-

tion des exigences de l'utilisateur, la planification doit - par exemple - pouvoir gérer des notions de haute-disponibilité ou de tarification. Outre le choix du cloud d'hébergement de la nouvelle VM, la planification doit également en déterminer le profil matériel adéquat. Pour cela, la planification peut être amenée à résoudre, par exemple, des exigences de l'utilisateur portant sur les besoins des logiciels embarqués (ex : un serveur Jonas doit avoir 1Go de ram au minimum), sur le prix maximum que l'utilisateur veut respecter pour chaque VM , voire les deux concomittamment. En plus du choix du cloud d'hébergement, de la détermination du profil matériel, la planification peut être amenée à gérer d'autres aspects tels que le choix de l'image virtuelle à utiliser, la gestion des serveurs de noms DNS, la détermination de la nécessité d'obtenir une adresse IP publique pour la nouvelle VM...

- Configurer la nouvelle instance pour qu'elle intègre l'application. Suivant la solution d'exécution utilisée, cette préoccupation peut nécessiter différentes reconfigurations. Si la solution utilisée est VAMP par exemple, cette configuration passe par la mention des interfaces du composant Fractal représentatif de la nouvelle instance de serveur Jonas. Pour d'autres, il s'agira de renseigner des paramètres possiblement connus uniquement lors de l'exécution de l'application (ex : adresse ip du serveur MySQL, nom du serveur Jonas, port du serveur MySQL, crédences du serveur MySQL,..).

- Configurer le reste de l'application pour intégrer la nouvelle instance de serveur Jonas. Là encore, cette préoccupation est liée à la solution d'exécution utilisée. Il peut aussi bien s'agir de la spécification d'interfaces Fractal, que du renseignement de paramètres possiblement dynamiques (ex : ip de la nouvelle VM, nom de la nouvelle instance de serveur Jonas à renseigner auprès du module mod_jk du serveur Apache,..)

Le rôle de la planification est donc de compléter la décision de l'analyse pour établir un plan de reconfiguration à même d'être interprété par la solution d'exécution. Ce plan dépend donc de la solution d'exécution et doit être adapté en conséquence. Le plan de reconfiguration constitue une description des modifications de l'architecture courante de l'application établie en appliquant la décision de l'analyse. Sa détermination doit également obéir à des exigences de l'utilisateur que celui-ci peut fixer au travers de l'expression de contraintes. L'état de l'art a montré que les solutions actuelles ne permettent pas d'adresser toutes les problématiques de l'établissement d'un tel plan de reconfiguration, or **nous pensons qu'une planification complète et complètement descriptible est nécessaire**. Cependant, les approches actuelles fonctionnent en mode "boîte noire" : **nous pensons qu'en conséquence, une brique séparée permettant d'exprimer l'ensemble des exigences de l'architecture de l'application doit être réalisée**.

Ce manuscrit décrit dans sa suite une proposition de solution à même de lever les verrous des solutions actuelles. La section suivante décrit les caractéristiques requises

pour que cette solution puisse à la fois permettre d'atteindre les objectifs généraux de la thèse mais aussi d'élargir le champ des possibles en matière de gestion de l'élasticité.

4.4 Caractéristiques requises pour la planification

Cette section a pour but d'établir les caractéristiques d'une solution innovante de planification à des fins d'automatisation de l'élasticité d'applications exécutées dans le cloud. A partir des limites identifiées dans l'état de l'art, le tableau 4.1 répertorie les verrous à lever et des caractéristiques que la solution proposée se doit d'avoir. Bien évidemment, pour une application donnée, l'élasticité possible est limitée par les aptitudes de celles-ci.

L'explicitation de ces caractéristiques est la suivante :

- La solution proposée doit être agnostique vis-à-vis des applications, des architectures applicatives et des environnements tout en permettant de coller aux spécificités de chaque cas. La solution doit également gérer les applications patrimoniales en ne nécessitant aucune modification de l'application. Bien évidemment, seules les applications virtualisables et adaptées à l'élasticité sont visées.
- La solution doit permettre de spécifier le comportement élastique d'une application à un niveau de granularité plus fin que la VM. La spécification du comportement élastique doit être à la fois exhaustif mais aussi extensible.
- Une vision fidèle de l'état courant de l'application doit être offert.

A ces caractéristiques s'en ajoutent d'autres relatives à la facilité d'utilisation de la solution ainsi qu'aux performances. Il est nécessaire d'offrir une compréhension aisée de la détermination du plan de reconfiguration. Cette détermination doit s'effectuer en un temps négligeable par rapport aux autres phases de la boucle MAPE-K appliquée à l'élasticité.

L'ensemble des caractéristiques identifiées dans cette section constitue le socle de la réflexion scientifique menée au cours des travaux de thèse. Ce socle a permis d'aboutir à une solution qui lève l'ensemble des verrous identifiés. La section suivante aborde les concepts de l'approche retenue.

4.5 Vulcan, une solution de planification pour une élasticité automatisée

La solution présentée dans ce manuscrit est appelée Vulcan en référence à la vulcanisation, une réaction chimique permettant de rendre un élastomère élastique. Le fonctionnement global de Vulcan est illustré par le schéma 4.3.

La planification réalisée par Vulcan s'appuie sur deux modèles.

1. Le premier des deux modèles est appelé *Modèle par Extension* [77]. Il permet la représentation de l'architecture courante d'une application ainsi que la représentation de l'architecture-cible. Si l'architecture courante est issue de la *Connaissance*

Limite identifiée	Risque possible	Caractéristique requise
Spécificité à une application	La solution est limitée dans sa gestion des applications	Agnosticité vis-à-vis de l'application, tout en permettant de spécialiser l'élasticité à une application
Spécificité à un type d'architecture applicative (ex : applications n-tiers)	La solution est limitée dans sa gestion des architectures applicatives	Agnosticité vis-à-vis de l'architecture applicative
Spécificité à des IaaS ou des solutions d'exécution	Risque d'enfermement propriétaire	Agnosticité vis-à-vis de l'environnement d'exécution
Mauvaise gestion des applications patrimoniales virtualisables	Non-utilisation de la solution par manque d'adaptation	Ne pas nécessiter de modifications de l'application
Granularité limitée à la VM	Limitation quant à la gestion du placement (ex : choix du profil matériel impossible)	Offrir une granularité plus fine que la VM au niveau du composant applicatif et des paramètres de configurations
Non-exposition à l'utilisateur de l'architecture de l'application	Non-connaissance de l'état de l'application par l'utilisateur	Offrir une représentation de l'architecture de l'application durant son exécution
Comportement élastique non descriptible(ou partiellement)	La solution est limitée dans les scénarios possibles d'élasticité	L'élasticité doit pouvoir être décrite totalement
Absence d'expression de contraintes	Le comportement élastique est compliqué à spécifier pour l'utilisateur et donc potentiellement incompréhensible	Permettre l'expression de contraintes relatives à l'élasticité
Expression du comportement élastique suivant différentes approches (ex : contraintes et programmation)	Difficulté d'apprentissage de la solution par l'utilisateur	Recours à une approche unique pour la spécification du comportement élastique
Les contraintes ne peuvent être étendues	La solution ne peut gérer de nouvelles contraintes	Le formalisme du modèle d'élasticité doit permettre d'étendre les contraintes gérées par la solution

TABLE 4.1 – Caractéristiques requises pour une solution innovante de planification

de la boucle MAPE-K, l'architecture-cible provient du moteur de calculs de Vulcan. Vulcan réalise ses calculs directement sur le modèle par extension. La réalisation de ces calculs dépend d'un algorithme de planification innovant. Une des particularités de cet algorithme est de réaliser les calculs en tenant compte des différentes contraintes de l'application. Ces contraintes sont en fait connues grâce au second

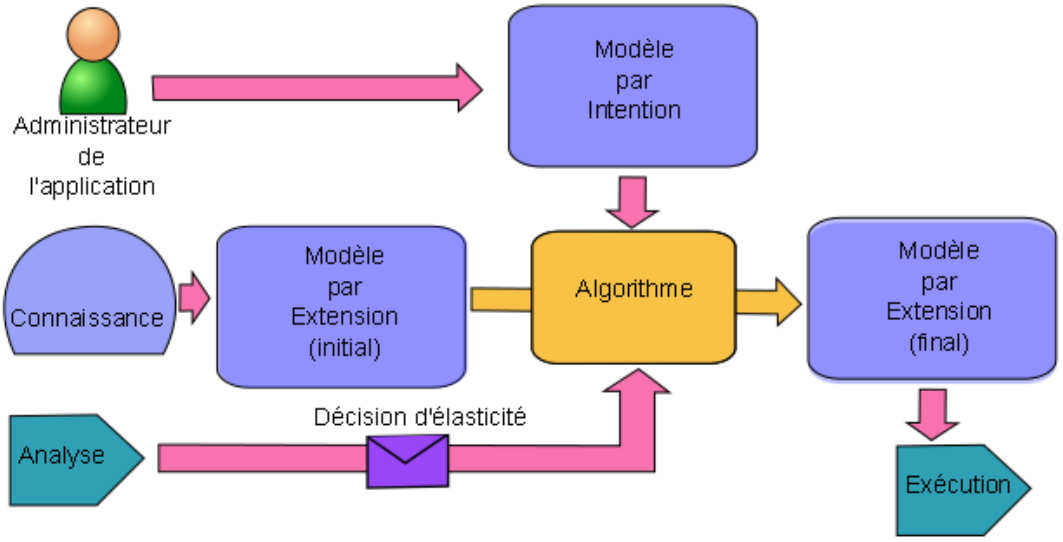


FIGURE 4.3 – Fonctionnement global de Vulcan, un gestionnaire de planification de l'élasticité

modèle sur lequel s'appuie Vulcan : le *Modèle par Intention*.

2. Le *Modèle par Intention* [77] permet à l'algorithme de connaître comment modifier l'architecture de l'application. Il est fourni par l'administrateur de l'application à rendre élastique. Il s'agit d'un gabarit pour l'ensemble des modèles par extension possibles. Pour cela, le modèle par intention centralise l'ensemble des contraintes d'une application et permet d'adresser l'ensemble des calculs nécessaires à l'établissement d'un plan de reconfiguration. Le modèle par intention de Vulcan fait usage d'un formalisme novateur permettant de décrire des contraintes suivant un mode d'expression unique. L'utilisation de ce formalisme conjointement avec celle de l'algorithme de Vulcan offre l'avantage de faciliter la réutilisation des contraintes exprimées pour une application par d'autres applications. Cela permet un apprentissage aisé de la solution.

La suite de ce document approfondit les concepts énoncés dans cette section tout en faisant le lien avec les caractéristiques requises identifiées dans la section précédente.

Chapitre 5

Modèle par extension d'applications élastiques

Sommaire

L'émergence du cloud computing a vu des travaux proposer des ontologies afin d'adresser la diversité des offres de cloud et l'absence de standards de description. Ces travaux visent principalement à homogénéiser des offres de cloud ayant des descriptions très hétérogènes. Une ontologie se définit par l'ensemble structuré des termes et concepts caractéristiques d'un domaine. Les ontologies permettent de décrire des éléments en les liant par des relations avec une sémantique précisée par l'ontologie [169]. Certaines approches visent à offrir une connaissance homogène la plus complète possible, accessible par l'utilisateur pour des services offerts par les fournisseurs de cloud [147, 111]. D'autres ont pour objectif d'offrir des services spécifiques comme du stockage répartis et fiable dans le cloud [22, 85] ou du déploiement automatisé sur plusieurs IaaS en prenant en compte les niveaux de services demandés par l'utilisateur [110].

Ces travaux montrent un enjeu actuel de recherche qui est d'offrir une description homogène de services hétérogènes. Il s'agit plus précisément de permettre une connaissance simple des plateformes sous-jacentes mais aussi de l'application comme le montre [169].

Le modèle par extension va dans ce sens et offre une connaissance de l'état courant de l'application. Il permet en outre à Vulcan de planifier les reconfigurations à opérer sur l'application lorsqu'une décision d'élasticité est prise par l'analyse. Ce chapitre a pour

objectif de décrire de façon détaillée ce qu'est le modèle par extension.

5.1 Description du modèle par extension

Le modèle par extension est une représentation de l'état de l'application. Il ne s'agit aucunement d'un modèle de programmation comme OSGI [48], ou Fractal [39, 129] par exemple. Le modèle par extension est une abstraction des ressources de l'application servant à leur administration. De ce fait, **Vulcan n'impose pas de changement au niveau du code applicatif**. En revanche, le modèle par extension permet une réelle connaissance de l'application durant la mise en œuvre de l'élasticité.

S'il n'est pas un modèle de programmation comme Fractal, le modèle par extension en utilise toutefois les concepts architecturaux. Ces concepts permettent de décrire l'intégralité des préoccupations à adresser pour permettre l'élasticité d'une application dans le cloud. Les préoccupations à adresser se répartissent en deux niveaux d'administration de l'élasticité [24, 59, 157] : le niveau infrastructure et le niveau applicatif.

- Le niveau infrastructure de l'administration de l'élasticité des applications dans le cloud concerne principalement les VMs, les IaaS, le réseau, leurs configurations et leurs liens. Une illustration d'un tel lien est l'hébergement d'une VM dans un IaaS : de l'existence, la disponibilité et la capacité d'accueil du IaaS dépend l'exécution de la VM.

- Le niveau applicatif concerne les logiciels qui par leur exécution assurent la mise en œuvre des fonctionnalités de l'application. Ce niveau concerne également les configurations de ces logiciels. D'autre part, le niveau applicatif inclut également les liens entre les briques logicielles. Les briques logicielles peuvent effectivement dépendre d'autres briques logicielles pour leur exécution. Cette notion de dépendance reflète la nécessité d'une brique logicielle de communiquer avec la brique logicielle dont elle dépend, afin de pouvoir elle-même être fonctionnelle. C'est par exemple le cas dans une application web comme Springoo où le serveur frontal Apache dépend d'un serveur Jonas qui lui même dépend d'un serveur de base de données MySQL.

La gestion de l'élasticité d'une application requiert donc d'administrer ces deux niveaux. Pour cela, il est nécessaire d'en permettre la représentation dans le modèle par extension. En outre, ces deux niveaux de préoccupations doivent être gérés de façon concomitante de sorte à également adresser les liens existants entre les deux niveaux. A titre d'exemple, les briques logicielles d'une application s'exécutent au sein de VMs qui elles-mêmes sont hébergées dans des clouds. Une réponse à cette problématique est en partie fournie par les travaux autour des solutions Jade, TUNe et TUNeEngine évoquées dans la suite de ce manuscrit en sous-sous-section 6.2.3. Ces travaux ont effectivement permis de démontrer la pertinence du recours aux architectures à composants pour rendre homogène la gestion automatisée et simplifiée d'éléments hétérogènes [68].

Afin de permettre la modélisation de l'ensemble des préoccupations décrites ci-avant, le modèle par extension utilise des concepts issus des architectures à base de composants et plus particulièrement du modèle Fractal. Le choix de retenir les architectures à base de composant pour le modèle par extension se justifie également par l'expressivité de cette approche ainsi que par sa capacité à permettre des calculs et des raisonnements. Si le premier point a été démontré par des solutions comme Fractal ou OSGi, le second l'a été par exemple par SystemCAS, une solution de simulation de circuits et de programmes [91, 122].

Le schéma 5.1 illustre les concepts retenus pour le modèle par extension. Une application est constituée de *composants primitifs* qui sont exécutés dans des *composants composites*. Un *composant primitif* ne peut contenir d'autres composants, tandis qu'un *composant composite* peut contenir des *composants primitifs* et/ou des *composants composites*. La suite du manuscrit désigne les *composants primitifs* par le terme de *composants*, et les *composants composites* par celui de *conteneurs*. L'exécution d'un composant au sein d'un conteneur est décrite par le *placement* de ce composant dans son conteneur d'exécution. Par exemple, dans la figure 5.1, le composant C.1 est *placé* dans le conteneur VM.1. La notion de placement désigne également l'exécution d'un conteneur au sein d'un autre : le conteneur VM.1 est ainsi *placé* dans le conteneur Cloud.2. Cet exemple met en évidence une notion additionnelle qui est *la hiérarchie de conteneurs* : les conteneurs de plus bas niveau sont placés dans des conteneurs de plus haut niveau.

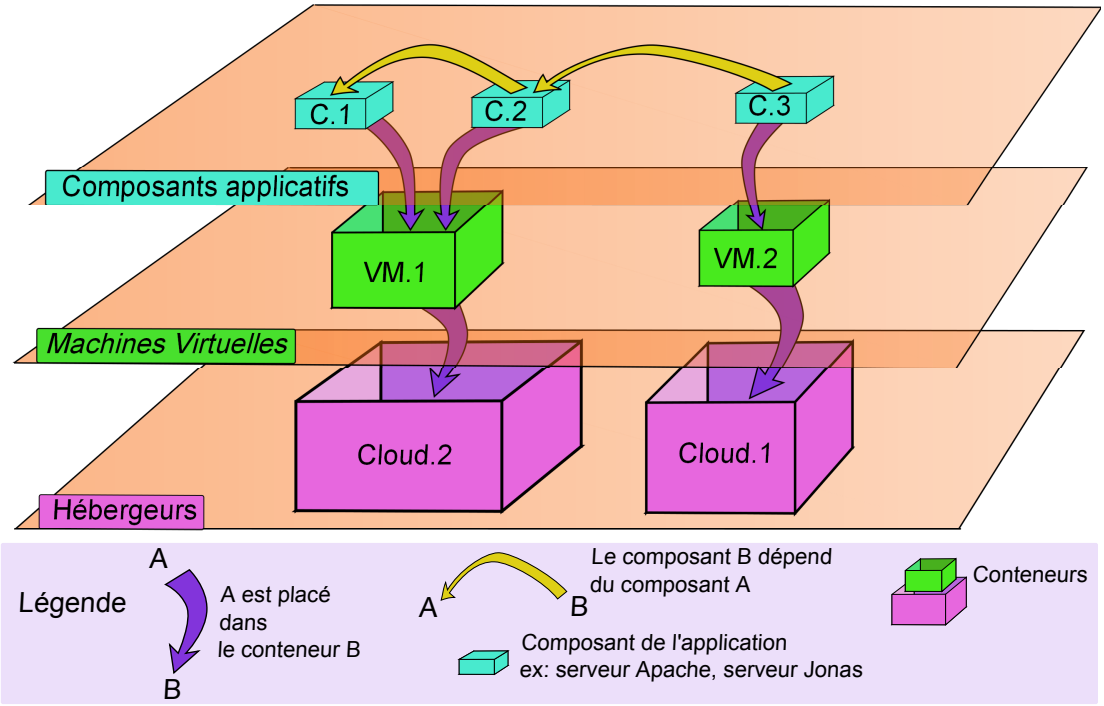


FIGURE 5.1 – Vue globale des concepts manipulés dans le modèle par extension

Outre la notion de placement, le modèle par extension utilise celle de *liaison*. Une liaison relie deux composants et traduit une dépendance de l'un par rapport à l'autre. Plus précisément, il s'agit d'une relation orientée qui indique la dépendance du composant d'origine envers le composant de destination. Dans le cas de la figure 5.1, il existe une *liaison* depuis le composant C.3 vers le composant C.2 : cela traduit une dépendance de C.3 envers C.2.

La configuration de chaque composant et de chaque conteneur est portée par chaque élément grâce à des paramètres de configuration. Les paramètres permettent de mentionner l'ensemble des attributs de chacun des éléments. Par exemple, les paramètres d'un conteneur de type VM mentionnent - notamment - le profil matériel de la machine représentée.

L'ensemble des notions du modèle par extension est issu du méta-modèle illustré par la figure 5.2 et la section suivante donne un exemple de modèle par extension pour une application très largement utilisée. De plus, des exemples issus de l'implantation de Vulcan sont donnés dans le chapitre 8.

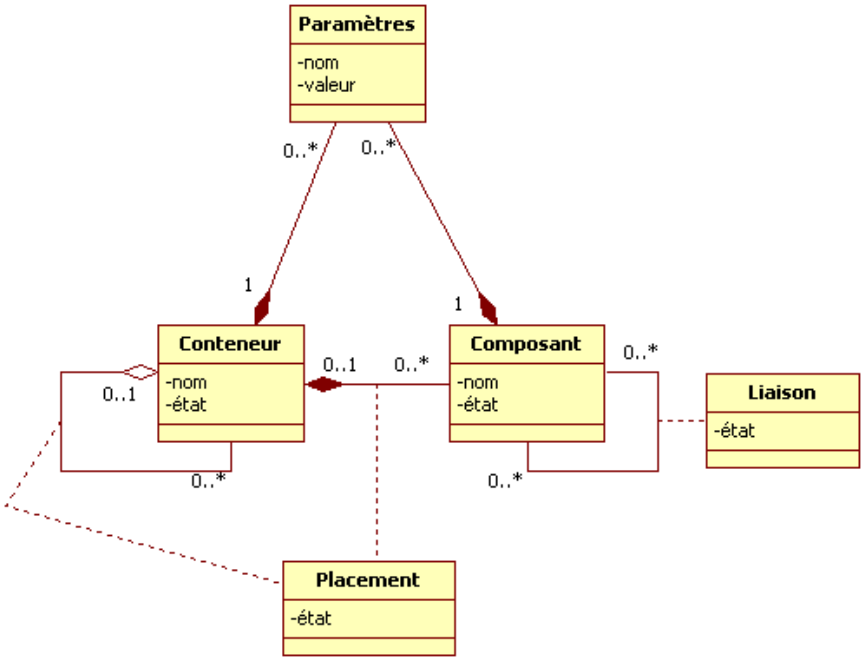


FIGURE 5.2 – Méta-modèle du modèle par extension

5.2 Modèle par extension d'une application de distribution de contenu

La figure 5.4 est une représentation graphique d'un modèle par extension pour l'application représentée par la figure 5.3. Il s'agit d'un service de distribution de contenu (CDN) [173]. L'utilisation du cloud pour les services CDN suscite l'intérêt de nombreux travaux de recherche [37, 44, 99] et plus particulièrement en ce qui concerne l'élasticité [37, 97, 163].

Un service CDN sert à acheminer du contenu statique (images, fichiers) vers des utilisateurs finaux géographiquement éloignés. La particularité d'un CDN est de faire transiter les données depuis le serveur central appelé *serveur.1* sur la figure 5.3 vers les clients finaux au moyen de caches intermédiaires locaux appelés *nœuds*. Un service CDN est particulièrement bien adapté pour réaliser du streaming vidéo et soutenir une demande élevée.

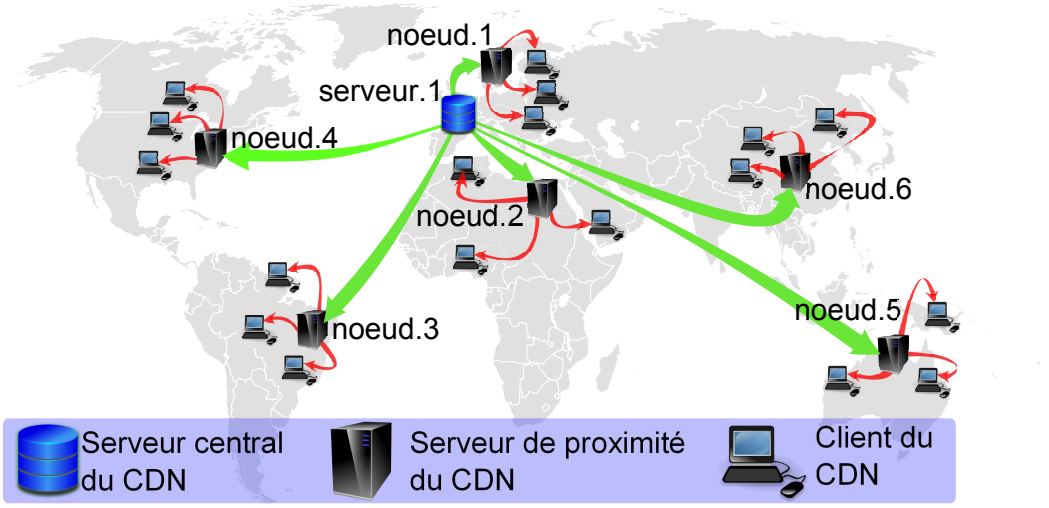


FIGURE 5.3 – Exemple d'application de distribution de contenu (CDN)

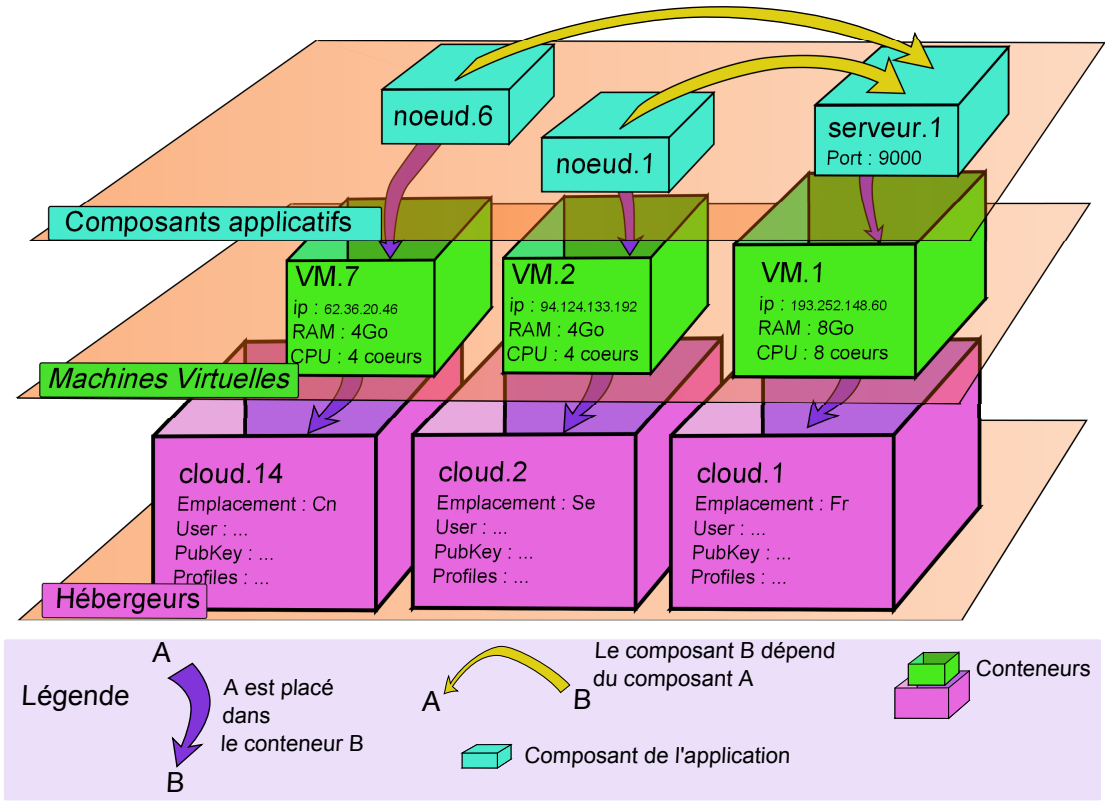


FIGURE 5.4 – Représentation graphique d'un modèle par extension d'une application CDN

Le recours à un service CDN permet :

1. Une réduction de la charge du serveur central.

2. Une amélioration de l'expérience utilisateur en réduisant les latences et/ou en maximisant la bande passante.

Pour cela, les nœuds stockent les données du serveur central et les envoient vers les utilisateurs qui leur sont connectés. Afin d'optimiser les performances, les clients finaux doivent être routés vers le meilleur nœud, qui correspond à celui qui minimise la distance le séparant du client final. Toutefois, la charge des différents nœuds doit être prise en compte pour qu'il n'apparaisse pas de nœuds surchargés aussi appelés *points chauds* [132].

Dans la figure 5.4, seuls certains nœuds ont été représentés par souci de lisibilité. L'intérêt de ce schéma est de montrer un exemple des différentes notions décrites théoriquement ci-avant dans ce chapitre avec un cas d'usage différent de Springoo. Ce sont ces notions que la planification doit manipuler afin d'obtenir une liste exhaustive des reconfigurations à opérer sur une application durant son élasticité. Cette manipulation

implique de maîtriser des opérations élémentaires qui par leur application vont permettre de modifier le modèle par extension d'une application. La section suivante décrit ces opérations.

5.3 Opérations élémentaires de modification d'un modèle par extension

Le méta-modèle du modèle par extension représenté par la figure 5.2 est extrêmement simple. On ne prétend pas que ce méta-modèle est exhaustif mais il s'est montré suffisant pour l'ensemble des scénarios considérés. D'autres possibilités ont été explorées : elles visaient à apporter des raffinements au modèle par extension, mais elles ont été abandonnées. Ces abandons sont la conséquence d'un constat pragmatique simple pouvant être énoncé ainsi : **l'augmentation du nombre de concepts dans un modèle augmente la complexité pour le modifier d'une façon pas toujours justifiée au regard des précisions offertes**. De façon concrète ce constat a résulté en **un modèle par extension simple mais non simpliste** : il s'agit d'un choix délibéré et calculé pour pouvoir à la fois décrire les applications élastiques avec une large couverture tout en ayant un ensemble contenu d'opérations élémentaires de modification. Le méta-modèle du modèle par extension résulte donc d'un choix volontaire et propre à l'approche de n'avoir qu'un faible nombre de concepts à manipuler. Il en résulte un ensemble d'opérations élémentaires de modifications clairement identifié et avec une petite cardinalité. Les opérations élémentaires de modifications sont effectivement les suivantes :

- Ajout/retrait d'un composant/conteneur.
- Lier/délier des composants.
- Placer un composant/conteneur dans un conteneur, et son opération inverse.
- Définir la valeur d'un paramètre d'un composant/conteneur.

Toutefois, même si le nombre d'opérations élémentaires de modification du modèle par extension est réduit, il n'en demeure pas moins qu'elles doivent être calculées pour permettre la planification. La contribution exposée dans ce manuscrit utilise un gabarit qui permet de gérer l'ensemble de ces reconfigurations. Il s'agit du *Modèle par Intention* que le chapitre suivant décrit.

Chapitre 6

Modèle par intention d'applications élastiques

Sommaire

Si le modèle par extension permet une représentation statique de l'application, la gestion de la planification de l'élasticité automatisée implique de le manipuler de façon dynamique. En effet, une application élastique n'a plus un unique modèle par extension mais une séquence potentiellement infinie. Un enjeu actuel de la recherche est d'établir des modèles complets de façon automatique tout en ayant un moyen d'exprimer des contraintes. Ces contraintes concernent aussi bien le niveau applicatif que le niveau infrastructure exposés dans le chapitre précédent. L'obtention de modèles complets suscite de nombreux travaux de recherche mais qui n'adressent pas toutes les préoccupations de l'élasticité. [47, 138, 172] se focalisent sur le niveau infrastructure de l'administration de l'élasticité. [74, 25, 86] adressent quant à eux le niveau applicatif. Enfin, la solution [128] adresse le niveau infrastructure, le niveau applicatif ainsi que leurs liens mais n'est pas adaptée à la gestion de l'élasticité puisque seuls des modèles statiques sont calculés. D'autre part, des travaux démontrent les limites des solutions actuelles de gestion de l'élasticité en créant des solutions à même de retrouver de l'information perdue qui concerne pourtant les deux niveaux d'administration de l'élasticité. [106] adresse le recouvrement d'informations sur les images virtuelles d'Amazon EC2 afin de connaître

les différents logiciels embarqués. Il s'agit d'informations du niveau applicatif dont la connaissance n'est pas exposée par absence de modèle adapté.

L'automatisation de l'élasticité implique donc de non seulement adresser les deux niveaux d'administration de l'élasticité mais aussi de limiter la perte d'information nécessaire à la prise de décision. Il est aussi nécessaire de pouvoir exprimer des contraintes comme le montrent les travaux de recherche actuels [74, 25, 86, 128]. Les sections qui suivent décrivent l'approche retenue pour le modèle par intention.

6.1 Gestion de la modification du modèle par extension courant

Le modèle par intention sert de gabarit pour la modification du modèle par extension courant. Une telle modification requiert d'adresser l'ensemble des concepts exprimés dans le modèle par extension. Pour cela, le modèle par intention permet d'adresser quatre types de contraintes qui suffisent à couvrir l'ensemble des aspects de modification d'un modèle par extension. Ces types sont :

- **Héritage** : une application est constituée de composants et de conteneurs. Chaque composant hérite une partie de ses caractéristiques d'un *type de composant* tout comme un objet hérite d'une classe en programmation orientée objet. Dans tout modèle par extension, aucun composant ne peut exister s'il n'hérite pas d'un type de composant. De la même façon, un conteneur doit hériter d'un type de conteneur.

- **Liaison** : tout composant peut avoir des liaisons depuis ou vers d'autre(s) composant(s). La création de ces liaisons doit satisfaire des contraintes afin de garantir les exigences exprimées. Par exemple, dans la figure 5.4, chaque composant représentatif d'un nœud doit avoir une liaison unique vers le composant nommé *serveur :1*.

- **Placement** : tout composant doit être placé au sein d'une hiérarchie complète de conteneurs. En reprenant l'exemple du schéma 5.4, tout composant doit être placé dans sa propre VM. De plus, chaque VM doit elle-même être placée dans un conteneur *cloud*.

- **Configuration** : tous les composants et les conteneurs doivent avoir une configuration complète. Dans le cas de la figure 5.4, chaque composant *nœud* doit exposer son service au travers du port 80.

Ces quatre types de contraintes sont au cœur de la contribution du modèle par intention. Le tableau 6.1 montre une liste non-exhaustive d'exemples de contraintes pouvant être exprimées et le type qui leur correspond. Comme le montre ce tableau, les contraintes pouvant être exprimées ne sont pas limitées par les types cités ci-avant. En effet, les quatre types servent à identifier quelles sont les entités et propriétés du modèle par extension concernées.

Les contraintes peuvent mêler plusieurs types. Il est par exemple possible de vouloir exprimer des contraintes de liaison en fonction du placement et réciproquement.

Id	Type de contrainte	Objectif	Exemple de contrainte
1	Héritage	Cardinalité	Garantir qu'un type de composant n'a toujours qu'une seule instance (singleton)
2	Héritage	Cardinalité	Imposer un nombre maximal de VMs pour une application
3	Liaison	Qualité de service	Lier uniquement des composants dont la distance n'excède pas une certaine limite
4	Liaison	Cardinalités	Lier des composants qui ne sont pas déjà liés par plus de n liaisons
5	Placement	Prix	Utiliser uniquement des clouds dont le coût horaire est inférieur à un prix donné
6	Placement	Localisation	Utiliser uniquement des clouds localisés en Europe
7	Placement	Disponibilité	Maintenir des répliquas géographiquement distribués
8	Configuration	Dimensionnement	Avoir des profils matériels de VMs qui soient adaptés aux besoins des composants hébergés
9	Configuration	Répartition de charge	Gérer une répartition dynamique de charge entre plusieurs composants en gérant l'affectation de coefficients de répartition

TABLE 6.1 – Exemples de contraintes pouvant être exprimées dans le modèle par intention

Par définition, les contraintes ne résultant que d'un seul type sont dites *contraintes du premier ordre*. Toujours par définition, les contraintes résultant de plusieurs types sont appelées *contraintes du second ordre*. Dans le tableau 6.1, les contraintes présentées sont de premier ordre à l'exception des contraintes 3, 7, 8 et 9.

D'autres exemples sont illustrés par la figure 6.1. Celle-ci montre un modèle par extension pour une application de surveillance similaire à celle évoquée dans le chapitre 9. Cette application est constituée de sondes qui au sein des VMs, remontent des informations telles que la charge CPU, l'occupation mémoire ou les entrées/sorties. Les sondes remontent leurs informations vers un agrégateur local. Cet agrégateur doit surveiller les VMs dont il a la charge ainsi qu'agréger les données remontées pour répartir la charge relative à leur traitement. Une fois agrégées, ces données sont ensuite remontées vers le serveur central *central.1* qui historise ces données et les expose. Le composant *WebUI.1* fournit une interface web qui expose les données du serveur central à des utilisateurs. Cette application présente des contraintes du premier ordre mais aussi des contraintes du second ordre. La figure 6.1 montre deux exemples de contraintes du second ordre. La première concerne l'établissement des liaisons applicatives : celles-ci doivent se faire

en fonction des placements puisque chaque sonde doit remonter ses informations auprès de l'agrégateur en charge de la zone dans laquelle la VM de la sonde est exécutée. La seconde contrainte de second ordre illustrée par la figure 6.1 concerne la co-localisation des composants *central.1* et *WebUI.1* au sein de la même VM : une instance de composant WebUI liée au serveur central doit être co-localisée avec ce même serveur central. Le cadre grisé de la figure 6.1 montre un scénario illicite au regard de ces contraintes. Si pour la première contrainte de second ordre, les liaisons sont construites en fonction des placement, la seconde contrainte donnée en exemple montre un cas de réciprocité.

Un exemple de contrainte du second ordre est également visible dans la figure 5.4 où pour chaque conteneur de type VM qui contient un composant de type *nœud*, son profil matériel doit être au minimum de 2Go de RAM et deux CPUs. Une telle contrainte est à la fois basique et courante mais elle combine pourtant des contraintes de type placements et héritage avec une contrainte de type configuration.

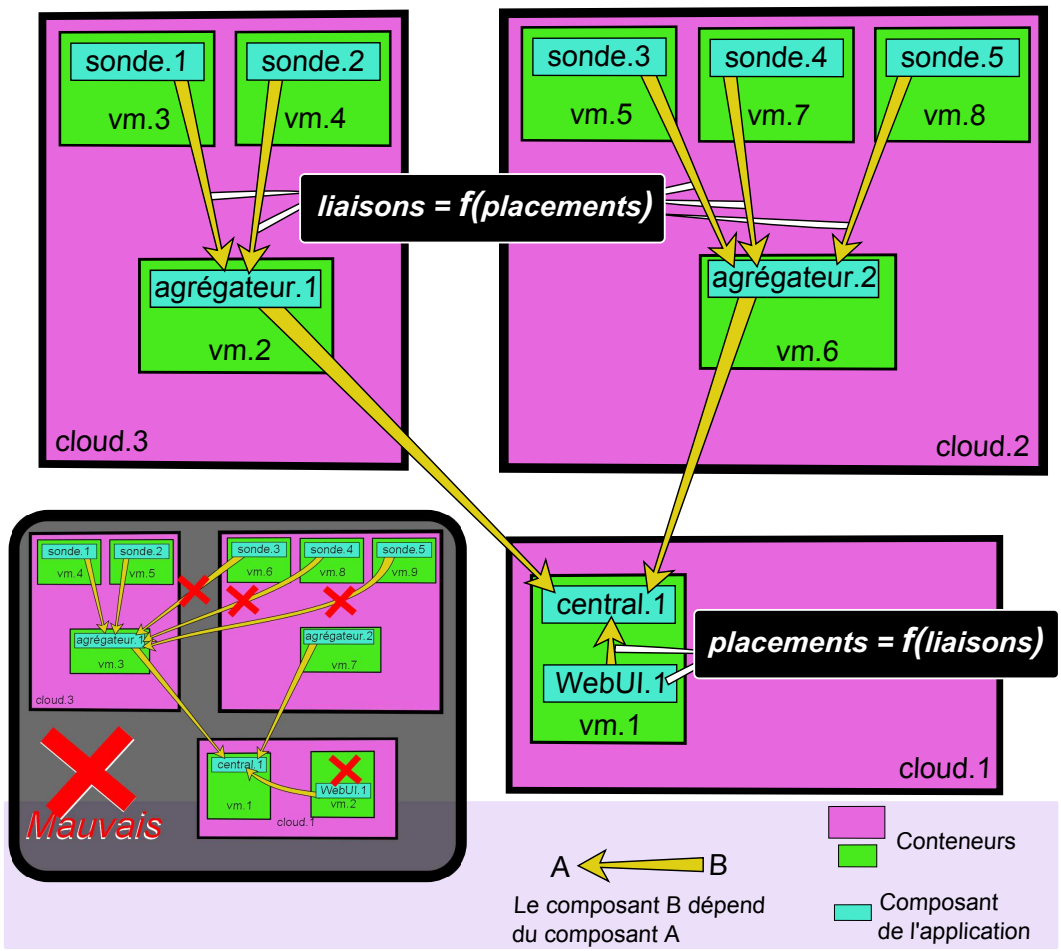


FIGURE 6.1 – Modèle par extension représentatif de l'état d'une application de surveillance

Un enjeu de la planification réside dans l'expression des quatre types de contraintes mais aussi dans la possibilité d'exprimer leur combinaison. Le modèle par intention permet d'exprimer aussi bien des contraintes du premier ordre que des contraintes du second ordre. Pour cela, il fait usage d'un formalisme innovant dont la section suivante donne une description.

6.2 Formalisme du modèle par intention

Le formalisme proposé est à la base des fonctionnalités et des caractéristiques du modèle par intention. Cette section commence par décrire dans une première sous-section les propriétés fondamentales qui ont été identifiées pour le formalisme du modèle par intention. Cette sous-section peut paraître très théorique mais elle permet de jeter la lumière sur les choix réalisés durant la conception de Vulcan. Ensuite, une deuxième sous-section décrit le formalisme proposé en donnant des exemples des contraintes exprimables dans le modèle par intention. Enfin, une troisième sous-section évoque d'autres solutions possibles et/ou étudiées.

6.2.1 Propriétés du formalisme

Durant la conception de la solution, plusieurs propriétés fondamentales ont été retenues pour le formalisme du modèle par intention.

- Une première propriété fondamentale concerne **la simplicité d'utilisation**. Pour cela, le formalisme du modèle par intention fait usage d'une approche uniforme pour trois des quatre types de contraintes à adresser avec une simplification pour le quatrième type (ce point est détaillé dans la suite du document). Toujours dans cette optique de simplicité, le formalisme utilise une expression de contraintes qui permet d'établir un plan de reconfiguration grâce à l'introspection du modèle par extension courant.

- Une deuxième propriété retenue est **l'agnosticité** vis-à-vis de l'ensemble des préoccupations d'une application élastique. Le formalisme du modèle par intention permet une totale agnosticité tant au niveau applicatif qu'au niveau infrastructure. Par exemple, la notion même de hiérarchie de conteneurs n'est pas déterminée et un composant peut aussi bien être placé au sein d'une VM qu'au sein d'un conteneur léger.

- Une troisième propriété identifiée concerne **l'évolutivité** du modèle par intention. Pour cela, le formalisme doit permettre au modèle par intention de ne pas être restreint à un ensemble de contraintes. En plus du modèle par extension qui offre une description non-figée des configurations des composants et des conteneurs, le formalisme du modèle par intention permet une extension des contraintes administrables.

- La quatrième et dernière propriété retenue est **l'adaptation** du modèle par intention à l'application. Il s'agit de permettre à un administrateur d'application de décrire un modèle par intention qui colle au plus près des spécificités et des besoins d'une application. Cette propriété peut paraître contradictoire avec la deuxième propriété de cette liste, mais elle est pourtant offerte par le modèle par intention grâce à son formalisme.

Outre ces propriétés fondamentales, le formalisme du modèle par intention doit également en garantir une fonctionnalité fondamentale. Il s'agit de permettre une introspection du modèle par extension courant depuis le modèle par intention. En effet, le choix a été fait de lier d'un point de vue fonctionnel les deux modèles de sorte à avoir une approche cohérente. Le modèle par extension expose donc la connaissance de l'architecture courante de l'application aux contraintes du modèle par intention de sorte à ce qu'elles puissent retourner des séquences de modifications permettant de les satisfaire. En ce sens, l'introspection du modèle par extension est une fonctionnalité primordiale que doit garantir le formalisme du modèle par intention. La sous-section qui suit décrit le formalisme proposé.

6.2.2 Description du formalisme

Le formalisme du modèle par intention repose sur une expression des contraintes à base d'un langage de requêtes ensemblistes à l'instar de SQL [93]. Les langages de requêtes ensemblistes présentent deux caractéristiques particulièrement intéressantes pour le modèle par intention puisqu'ils permettent non seulement de facilement **extraire des vues depuis une base de connaissance** mais aussi d'en **décrire les modifications**. Ces deux aspects sont en totale adéquation avec les nécessités fonctionnelles du modèle par intention vis-à-vis du modèle par extension. Il s'agit de deux caractéristiques qui ont également été largement traitées et démontrées par le passé pour les bases de données qu'elles soient relationnelles [42] ou orientées objets [31]. Le formalisme du modèle par intention à base d'un langage de requêtes ensemblistes constitue fonctionnellement la pierre angulaire du modèle par intention dans la mesure où il permet l'introspection du modèle par extension.

L'utilisation d'un langage à base de requêtes ensemblistes offre une expressivité qui simplifie le traitement massif des données, ce qui est particulièrement intéressant pour les systèmes visant à modifier des bases de connaissances comme le souligne [66]. Il est ainsi possible de réaliser un traitement spécifique sur de multiples éléments et par une expression minimale. Cette approche permet par exemple d'effectuer au travers d'une unique requête, des opérations sur les composants placés dans des VMs dont le coût horaire est supérieur à une limite. La force du mode d'expression par requêtes ensemblistes réside principalement dans le fait que les contraintes peuvent être décrites à partir d'une connaissance partielle du modèle. Dans le cas de l'exemple ci-avant, les composants recherchés peuvent être identifiés et filtrés par la seule mention d'un nom de paramètre et de la valeur attendue : $application.iaas.vms.coût < limite$. Il n'est pas nécessaire de connaître le chemin d'accès complet comme par exemple dans le monde java où l'ex-

pression est de la forme *IaaS.getVMs().get(VM).getCout() < limite* si l'application est exécutée dans un seul IaaS. Dans le cas contraire, il faudrait d'abord parcourir tous les IaaS et stocker dans une variable temporaire les éléments retournés par l'expression précédente. Cet exemple très simple, voire naïf, démontre bien l'utilité du mode d'expression par requêtes ensemblistes. Ceci est encore plus vérifiable lorsqu'il est requis d'effectuer la prise en compte croisée de plusieurs paramètres comme mentionné dans la section 6.1.

Un langage de requêtes ensemblistes permet donc une expression concise et compréhensible des contraintes mais il offre également d'autres avantages :

- Une composition de contraintes comme le souligne [166] ce qui est tout particulièrement intéressant puisque cette propriété permet une approche de type "divide and conquer" afin de "casser" la complexité. Ce point est le cœur de l'algorithme de planification expliqué dans le chapitre suivant.

- Un ordonnancement d'éléments. Cette caractéristique est intéressante lorsqu'il s'agit d'opérer des modifications sur le modèle par extension courant en mentionnant un ordre d'accession aux éléments. Par exemple, en considérant la contrainte n°4 du tableau 6.1 , cette caractéristique permet de réaliser une liaison avec le composant le moins déjà lié.

- Une bonne connaissance auprès des utilisateurs potentiels de Vulcan. SQL et XQuery sont en effet largement répandus.

Le modèle par intention constitue - grâce à son formalisme - un moyen programmatique de spécification des contraintes à respecter lors de l'élasticité pour une application. Le pseudo-code 1 montre un exemple pour une application CDN d'une contrainte qui impose que toute VM contenant un composant de type *nœud* ait un profil matériel avec 2 CPUs et 2Go de ram.

Code 1 Exemple d'une requête écrite en pseudo-code permettant la configuration d'un profil matériel en fonction des placements

```
1 : for each Container c in Extensional_Model/[all VM containers]
2 : where c contains a component inheritingfrom "node"
3 : return Configure(c, RAM=2GB, CPU=2)
4 :   ▷ Seuls les conteneurs de type VM qui contiennent un composant de type nœud. Le résultat Configure(c, RAM=2GB, CPU=2) est par la suite interprété par le moteur de planification.
```

Le résultat retourné par la requête en pseudo-code est une opération de configuration qui est ensuite interprétée par le moteur de planification de Vulcan. D'autres exemples sont donnés dans le chapitre 8. La figure 6.2 présente le méta-modèle du modèle par intention.

La sous-section suivante compare le choix retenu pour le formalisme du modèle par intention avec d'autres propositions.

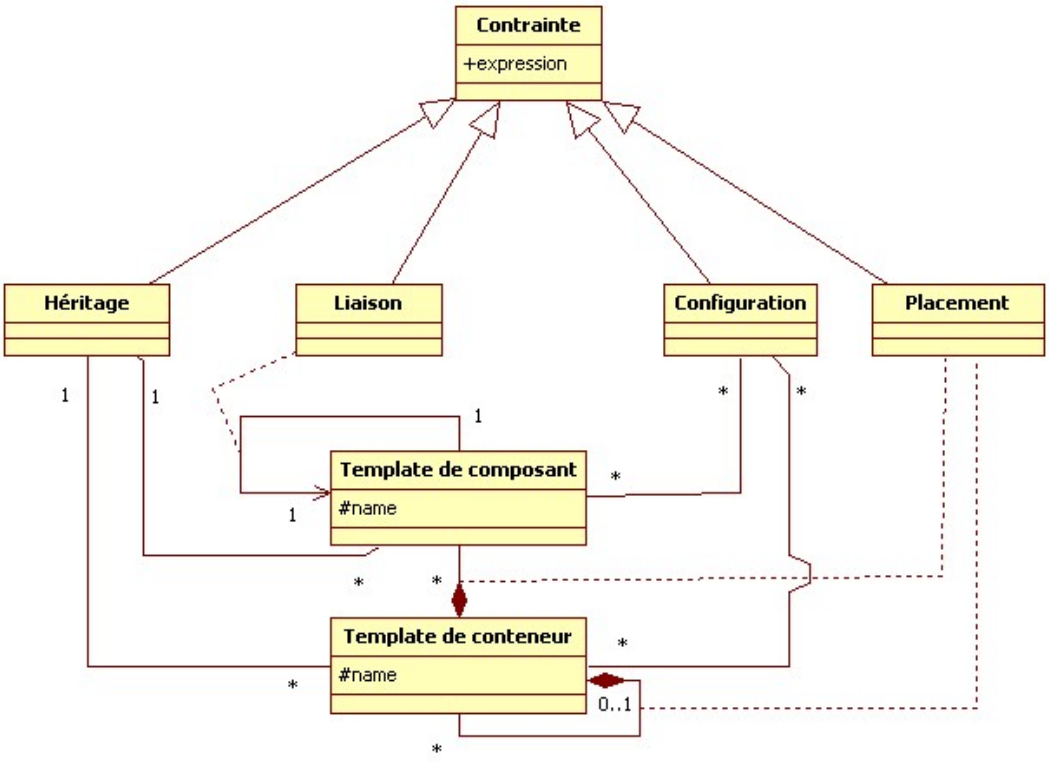


FIGURE 6.2 – Méta-modèle du modèle par extension

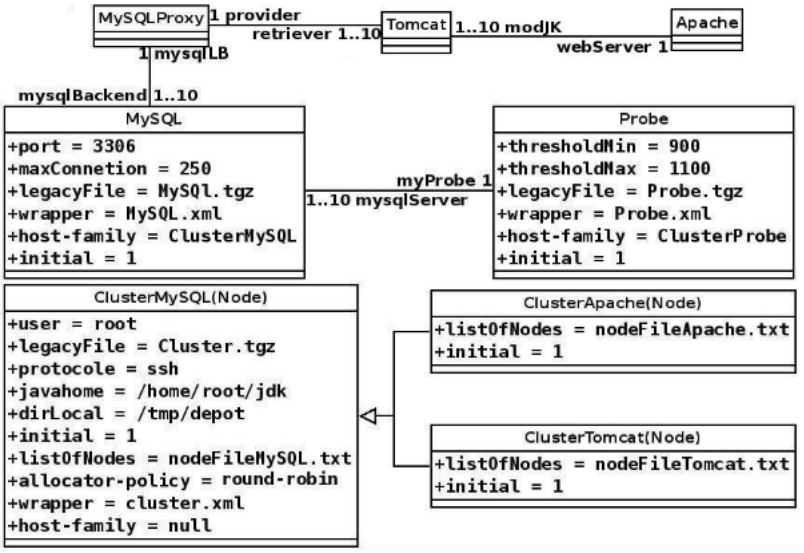
6.2.3 Positionnement par rapport à d'autres approches

Le recours à un langage à base de requêtes ensemblistes présente plusieurs avantages en comparaison d'autres approches. Cette sous-section les présente et explicite le choix pour un langage à base de requêtes ensemblistes.

Cardinalités

Le formalisme du modèle par intention n'est pas spécifique à un un type de contraintes comme l'est une approche par cardinalités. Les cardinalités sont par exemple utilisées par TUNe [38], un système autonome d'administration développé par l'équipe Astre au sein de l'Institut de Recherche en Informatique de Toulouse. Si TUNe n'adresse pas l'élasticité, cette solution offre une modélisation très intéressante permettant de lever les difficultés liées à la description d'une architecture de déploiement pour une application répartie et large échelle. En effet, TUNe fait suite aux travaux menés dans Jade [68, 36], une solution d'administration autonome d'applications réparties. Les travaux autour de Jade ont effectivement montré un verrou relatif à la description d'une architecture complète qui s'avère être non seulement une tâche fastidieuse lorsque le nombre d'éléments

à décrire augmente, mais également rebutant quant à l'adoption de la solution, en plus d'augmenter le risque d'erreurs. TUNe adresse cette problématique grâce à deux modèles complémentaires : la représentation de système, appelée *Representation System*(*SR*) est une architecture complète de l'application qui est le résultat de la projection du *modèle par intension* de TUNe. Le modèle par intension permet une représentation de haut-niveau de l'architecture de l'application. Son formalisme, à base de diagrammes UML, permet une description simplifiée d'architectures grâce à des *patterns* architecturaux. Ces patterns sont stipulés au moyen des liaisons entre des types de composants, lesquelles sont décorées par des cardinalités comme illustré par la figure 6.3. Cette approche à base de cardinalités permet de mentionner le nombre minimal et maximal d'éléments pouvant être liés entre eux.



Cette image extraite de [150] montre un exemple de modèle par intension pour une application J2EE

FIGURE 6.3 – Exemple de Modèle par Intension

Par ailleurs, les travaux portant sur le TUNeEngine [151, 149, 152] apportent des notions de dynamicité aux architectures applicatives contrairement à Jade et TUNe. Il devient ainsi possible de réaliser des opérations d'élasticité mais limitée dans la mesure où, par exemple, les contraintes 3, 5, 6, 7, 8 et 9 du tableau 6.1 ne sont pas exprimables.

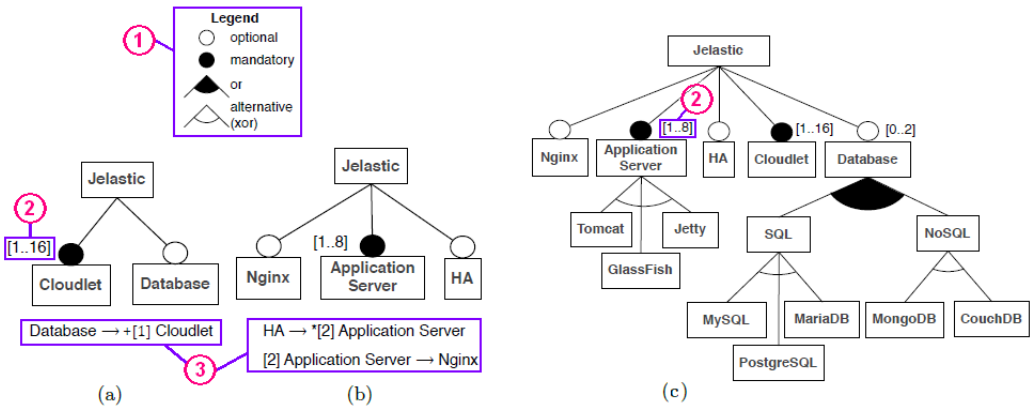
Il est à noter que les travaux autour de Jade, TUNe et TUNeEngine ont mis en lumière des bonnes pratiques de conception qui se retrouvent dans la solution proposée.

Enfin, l'approche par cardinalités si elle a été étudiée durant ces travaux de thèse en faisant l'objet d'un prototype, celle-ci fut abandonnée en raison de ses limitations. Lesdites limitations concernent aussi bien les contraintes non-adressées du tableau 6.1 que la nécessité de rajouter de la sémantique sur les cardinalités lors de la manipulation du modèle par intension. Par exemple, [156] propose plusieurs algorithmes permettant de mettre en œuvre des politiques de résolution du modèle par intension, afin de savoir comment interpréter les cardinalités. Par exemple, faut-il rechercher une connectivité

maximum ou au contraire minimum ? Par souci de recherche d'un formalisme homogène, les cardinalités n'ont donc pas été retenues pour nos travaux.

Feature models

Une approche plus ouverte est proposée par [124, 125, 126, 127, 128]. Cette approche est à base de *feature models* très utilisés dans les *Software Product Lines* qui visent à la production de logiciels adaptés à des domaines mais à partir d'une base de code commune. Les feature models constituent un moyen de modélisation de la variabilité. Les auteurs proposent d'étendre les feature models au moyen de cardinalités et de contraintes décrites par un Domain Specific Language (DSL). Les auteurs soulignent effectivement l'obligation de pouvoir exprimer des contraintes supplémentaires à celles exprimables par le formalisme des feature models et à celles des contraintes par cardinalités [125, 126]. Ils ont pour cela recours à un formalisme additionnel reposant sur un langage spécifique à ce domaine (DSL).



Cette image extraite de [126] montre trois feature models. Le cadre violet n°1 correspond au formalisme de base des feature models. Les auteurs y ajoutent les cardinalités comme dans les cadres n°2 et des contraintes sous la forme d'un *Domain Specific Language* visible dans les cadres n°3

FIGURE 6.4 – Exemple de Feature Model

Dans [119], les auteurs ont recours aux feature models pour calculer l'évolution de l'architecture d'une application. Dans ce but, ils utilisent les feature models conjointement à de la résolution de contraintes. Cette résolution est effectuée grâce à *un moteur de résolution de contraintes*.

De façon globale, l'approche par feature models n'offre pas autant de flexibilité que le formalisme du modèle par intention puisqu'elle nécessite la maîtrise des différents formalismes la composant : cela ne correspond donc pas à la recherche d'une approche homogène. Par ailleurs, [50] confirme le manque d'adaptation des feature models pour le formalisme du modèle par intention en les plaçant au niveau des préoccupations de la

thèse qui sont adressées par le modèle par extension.

Langage de navigation

Par le passé, des tentatives antérieures ont été menées sur les architectures à base de composants afin d'en permettre l'accès et la manipulation comme par exemple Fscript et Fpath pour Fractal [8]. Fscript est un DSL permettant d'effectuer des opérations de manipulations et de modifications sur des composants Fractal. Pour cela, Fscript s'appuie sur Fpath, un DSL d'accès aux composants Fractal inspiré de XPath[1]. Cette approche a été étudiée comme candidate pour la solution proposée, mais face à sa faible adoption et à son expressivité en retrait elle n'a pas été retenue.

Bilan des autres approches

Au cours des travaux de thèse, différentes approches ont été expérimentées. La première reposait sur une gestion des cardinalités avec le recours à des concepts additionnels. Cette approche n'a pas été retenue puisqu'elle ne permettait pas d'adresser l'ensemble des contraintes identifiées dans différents cas d'usage. Par ailleurs, cette approche nécessitait d'effectuer des choix algorithmiques résultant en une opacité pour l'utilisateur. De plus, les langages à base de requêtes ensemblistes permettent non seulement de gérer des cardinalités mais également d'aller plus loin. Cette approche a finalement été abandonnée.

Une approche à base de graphes a aussi été expérimentée pour la gestion des placements. Il s'agissait à partir d'une notion de distances croissantes, d'exprimer la distance minimale et maximale entre deux éléments. Cette approche avait comme intérêt de permettre d'adresser certaines problématiques du placement. Des algorithmes de résolution ont été créés afin de permettre la validation du modèle par intention proposé mais également sa résolution. Au final, l'approche souffrait des mêmes limites que l'approche à base de cardinalités : une opacité forte lors de la résolution, une absence d'extensibilité et un manque d'expressivité.

Finalement, si d'autres approches ont été étudiées, voire expérimentées dans le cadre des travaux présentés dans ce manuscrit, la solution proposée pour le modèle par intention s'appuie sur un langage à base de requêtes ensemblistes. Les autres solutions ont en effet montré des limites rédhibitoires.

6.3 Synthèse

Le modèle par intention permet d'introspecter le modèle par extension courant et d'en décrire les modifications. Ces modifications résultent des contraintes exprimées par l'utilisateur de Vulcan. Le recours à un langage à base de requêtes ensemblistes offre la possibilité de décrire des traitements massifs d'une façon simple, expressive et extensible. Un tel formalisme est à la fois globalement agnostique mais permet également de

[1] http ://www.w3.org/TR/xpath20/

coller aux spécificités de chaque application. Ces points sont détaillés dans la dernière partie de ce document. Malgré toutes ses qualités, le formalisme proposé ne permet pas à lui seul de conjuguer la simplicité d'utilisation avec la possibilité de décrire des contraintes complexes. Le chapitre suivant présente un algorithme qui, utilisé concomitamment avec le modèle par intention, permet d'offrir une planification qui repousse les limites actuelles de l'élasticité automatisée des applications dans le cloud.

Chapitre 7

Algorithme de planification

Sommaire

L'approche proposée repose sur une modélisation à deux niveaux : le modèle par intention et le modèle par extension. Le premier est le gabarit du second tandis que le second sert de base de connaissance au premier. Le modèle par intention lui-même repose sur un formalisme innovant à base d'un langage de requêtes ensemblistes. Ces contributions permettent d'exprimer des contraintes d'une façon simple, expressive et extensible. Cependant, la notion même de contraintes renvoie à une complexité pour l'utilisateur qui n'est pas anodine. Les solutions actuelles de gestion de l'élasticité réduisent cette complexité en diminuant l'ensemble des paramètres à préciser. Cependant, comme l'a montré ce document, ce type d'approche n'est pas souhaitable puisqu'il en résulte des restrictions importantes dans la gestion de l'élasticité. Ce chapitre présente la contribution des travaux de thèse qui permet d'adresser la gestion des contraintes du modèle par intention en réduisant la complexité d'utilisation. Pour cela, une approche innovante est exposée. Celle-ci repose sur un algorithme basé sur *un raisonnement par révision.*

7.1 Raisonnement par révision

Le raisonnement par révision est une notion issue de l'intelligence artificielle. Les qualités du raisonnement par révision sont largement décrites dans la littérature scientifique

[43, 52, 121, 133]. Il s'agit d'un raisonnement qui est dit *non-monotone*. Là où un raisonnement monotone ne permet que la considération ordonnée d'une succession d'éléments qui peuvent être des règles ou des contraintes, un raisonnement non-monotone permet de sortir de cette succession monotone en faisant des retours. Ces retours permettent de reconsidérer certaines règles et de ce fait, un raisonnement non-monotone permet de résoudre les conflits pouvant apparaître entre des règles ou contraintes comme l'explique [43] avec les trois lois de la robotique :

1. Un robot ne peut pas blesser un être humain ni, par son inaction, permettre à un être humain d'être blessé.

2. Un robot doit obéir aux ordres donnés par un être humain, *sauf si cela implique de contrevenir à la première loi.*

3. Un robot doit protéger son existence *tant que cette protection n'entre en conflit ni avec la première loi, ni la deuxième.*

Une approche monotone ne permet pas de résoudre les conflits inhérents à la résolution de ces lois sans une considération groupée et en seule passe de toutes ces lois. En revanche, une approche non-monotone peut résoudre ces conflits par un accès séquentiel grâce aux retours possibles dans la considération des différentes lois/règles/contraintes.

Outre la possibilité d'adresser de possibles conflits, le raisonnement par révision permet à l'utilisateur de Vulcan d'améliorer sa compréhension des scénarios d'élasticité. En effet, un constat simple réside dans le fait que plus un mécanisme est complexe plus la compréhension de ce mécanisme sera compliquée pour un être humain. Celui-ci peut effectivement éprouver des difficultés à comprendre les liens logiques liant le mécanisme aux résultats obtenus. Dans le cas d'une solution comme Vulcan, il en découlerait notamment :

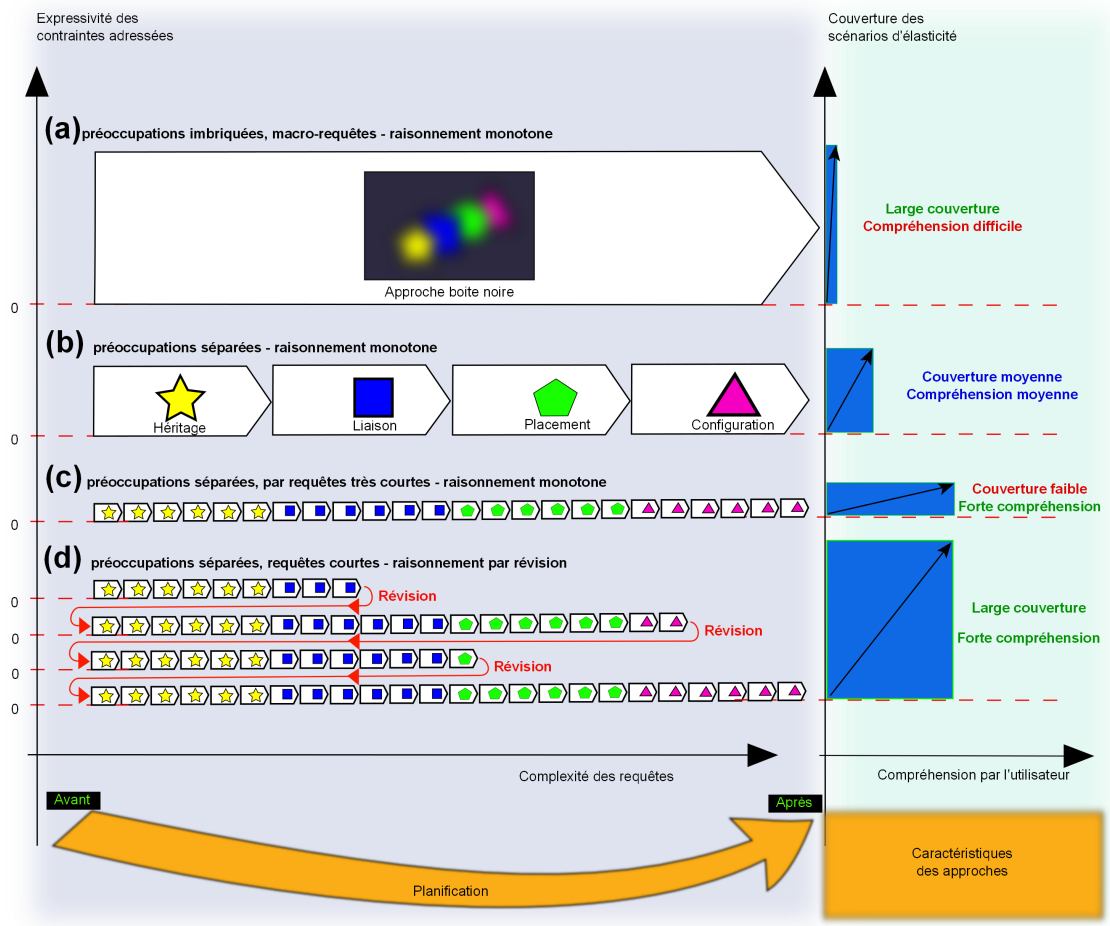
- une difficulté d'apprentissage.

- un manque de compréhension conduisant à des bogues.

- des difficultés lors du débogage et lors de l'exécution de l'application.

La complexité étant clairement un obstacle, la simplicité est un objectif recherché pour Vulcan. Il s'agit donc d'offrir un mécanisme qui soit accessible : l'approche à base de raisonnement par révision va justement en ce sens. Comme illustré par la figure 7.1, une telle approche permet effectivement de **réduire la complexité de chaque contrainte tout en maintenant non seulement une bonne compréhension mais également une bonne couverture des scénarios d'élasticité**.

De façon plus précise, la réduction de la complexité est obtenue en diminuant le nombre de préoccupations adressées par chaque contrainte. Pour cela, **les préoccupations globales de l'application sont réparties en sous-problèmes individuellement simples**. Ces sous-problèmes sont issus d'un type unique de contraintes et



La figure présente deux parties : à gauche quatre approches sont décrites (a, b,c et d). Pour chaque approche, la partie de droite correspondante en discute les (dés)avantages suivant deux critères : la compréhension offerte à l'utlisateur de Vulcan et la couverture des scénarios d'élasticité par l'approche.

source : `http://www.tomsguide.fr/article/ios-maps-iphone,5-389.html`

FIGURE 7.1 – Intérêt du raisonnement par révision : comparaison de différentes approches

n'adressent que des modifications simples du modèle par extension. L'apport du raisonnement par révision se situe principalement dans sa capacité à composer ces sous-problèmes entre eux de sorte à maintenir une très bonne couverture des scénarios d'élasticité, y compris sur les cas les plus complexes.

Une analogie de l'approche proposée est l'établissement d'un parcours permettant d'aller d'un pont A à un point B distants de plusieurs kilomètres en milieu urbain [1]. En dehors de situations "réflexes", un humain ne va ni créer un itinéraire complet en une

[1] Cette analogie n'est pas caractérisée par une grande rigueur scientifique mais elle a l'avantage de permettre une meilleure compréhension.

seule phase de réflexion, ni même le créer mètre par mètre. En revanche, une approche réaliste consiste à construire un itinéraire de façon itérative en vérifiant que le point en cours est atteignable et qu'il le rapproche du point final. L'approche proposée vise à respecter ce processus cognitif : il est non seulement intuitif mais également naturellement compréhensible par un être humain [43]. Il s'agit là de deux caractéristiques absolument primordiales.

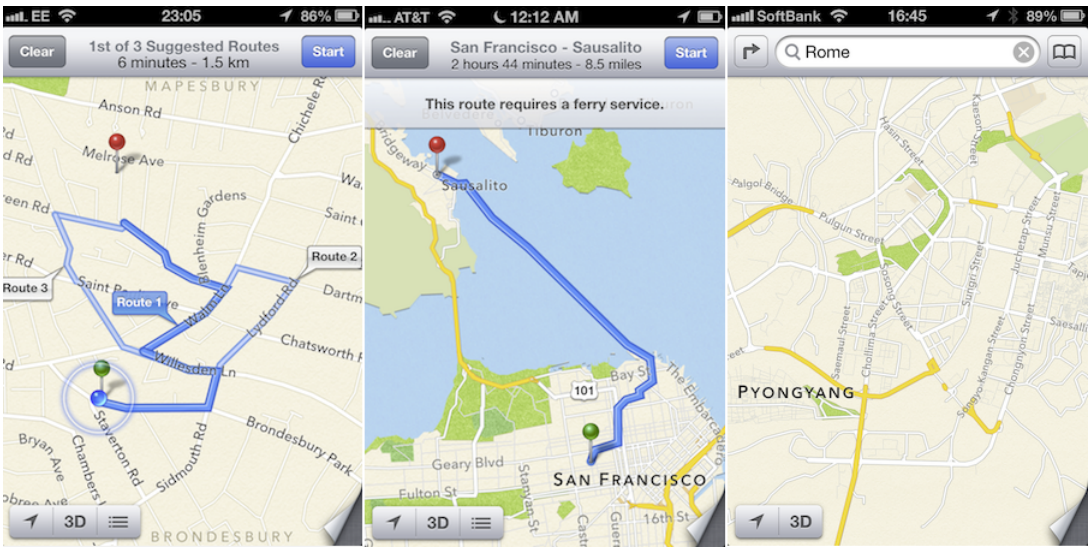
A titre d'illustration, la partie a) de la figure 7.1 montre une approche *boite noire* qui mêle l'ensemble des types de contraintes. Si ce type d'approche est satisfaisant du point de vue de la couverture des scénarios pouvant être exprimés, ce n'est pas le cas concernant la compréhension en raison d'une expression nécessairement très complexe. Pour reprendre l'analogie précédente, la partie a) revient à créer en une seule passe un itinéraire en tenant compte - par exemple - des sens interdits, de l'optimisation de la distance, de l'heure de circulation (pour les bouchons), du parking gratuit à l'arrivée, des travaux en cours de chemin... Au final, l'utilisateur crée des éléments rapidement très complexes au point qu'il lui est difficile de comprendre le lien entre le résultat qu'il obtient (i.e le scénario d'élasticité) et ce qu'il a décrit.

Les parties b) et c) illustrent quant à elles le fait qu'en diminuant la complexité de chaque contrainte, la compréhension augmente alors que les possibilités de description de scénarios d'élasticité se restreignent. Pour ces parties l'analogie peut être une création d'itinéraire avec une construction élémentaire : cela reviendrait par exemple à créer un itinéraire en ne faisant que naviguer tout droit, à droite, à gauche sans pouvoir prendre en compte d'autres éléments tels que ceux mentionnés pour la partie a). Les optimisations ne sont pas toujours possibles (ex : Paris - Grenoble via Marseille) mais en revanche, le processus est simple à comprendre. Néanmoins, il est possible que l'itinéraire ne soit pas calculable si les considérations deviennent trop simplistes (ex : gestion d'une voie sans issue sans opération de demi-tour). La compréhension qu'a l'utilisateur est réellement améliorée mais cela se fait au détriment des fonctionnalités, c'est-à-dire des scénarios couverts.

Enfin, la partie d) illustre l'approche retenue à base de raisonnement par révision qui permet à la fois une forte compréhension grâce à des contraintes très simples mais aussi une forte couverture des scénarios d'élasticité grâce à la composition permise par ce type de raisonnement. L'approche à base de raisonnement par révision permet ainsi d'allier les avantages d'une expression complexe à ceux d'une expression simplifiée. L'utilisateur peut décrire des scénarios complexes par plusieurs contraintes simples et garde une compréhension aisée du rapport de causalité liant chacune des contraintes décrites au comportement élastique obtenu. Pour reprendre l'analogie de l'itinéraire, il devient possible de supprimer des parties de cet itinéraire si celui-ci ne satisfait pas certaines exigences : la voie sans issue n'est ici plus un problème même avec une navigation simple, puisque la résolution de l'itinéraire permet de supprimer la partie l'y envoyant.

Comme l'a montré [133] dans le cas des applications web, le raisonnement par révision est particulièrement bien adapté pour la création de mécanismes de détection et de récupération d'états problématiques. Ceci est primordial, car tout système de contraintes

peut créer des erreurs comme illustré [1] par la figure 7.2. **Le raisonnement par révision permet que l'erreur ne soit non plus une fatalité mais au contraire, un mode de fonctionnement normal qui soit compréhensible et maitrisé par l'utilisateur**.



Bugs avec l'application de cartographie Apple Maps. De gauche à droite :
- Trois itinéraires sont proposés sans qu'aucun ne parvienne à l'arrivée.
- Une route originale empruntant les voies maritimes plutôt que le pont. Heureusement, l'utilisateur est prévenu qu'il devra avoir recours à un service de ferry !
- Une recherche de Rome, la capitale de l'Italie retourne Pyongyang, capitale de Corée du Nord, un résultat surprenant.

source : `http://www.tomsguide.fr/article/ios-maps-iphone,5-389.html`

FIGURE 7.2 – Tout système peut générer des erreurs qu'il convient de détecter et corriger

Cette section a présenté le raisonnement par révision une approche issue de l'intelligence artificielle. Le recours au raisonnement par révision permet d'offrir une plus grande compréhension à l'utilisateur de Vulcan tout en permettant une bonne couverture des scénarios d'élasticité. Cette approche innovante constitue le cœur de la contribution de ces travaux de thèse qui porte sur un algorithme de planification. La section suivante détaille cet algorithme.

7.2 Détails de l'algorithme de planification

L'algorithme de planification proposé se base sur un raisonnement par révision. Il permet à l'utilisateur de Vulcan de décrire des contraintes complexes à l'échelle de l'application au moyen d'un ensemble de contraintes pouvant être extrêmement simples. Les contraintes de l'utilisateur permettent de calculer des modifications à opérer sur le modèle par extension à partir de son introspection. Les sous-sections qui suivent donnent

[1] De façon humoristique.

l'algorithme de Vulcan et expliquent le contexte de sa mise en place ainsi que les moyens retenus pour le faire.

7.2.1 Énoncé de l'algorithme

Cette sous-section fournit un énoncé de l'algorithme de planification Vulcan. La figure 7.3 en est une représentation graphique. L'algorithme de planification de Vulcan est constitué d'une séquence principale et de déroutements depuis cette séquence.

- **La séquence principale** comprend quatre phases. Chacune d'elle correspond à l'un des quatre types de contraintes décrits en section 6.1. Dans la figure 7.3, les phases de la séquence principale sont repérées par des boites oranges.
- **Les déroutements** permettent d'opérer des révisions lorsqu'une étape de la séquence principale s'achève par un état "problématique", comme une voie sans issue dans le tracé d'un itinéraire. La figure 7.3 permet de bien visualiser la notion de déroutement pour le traitement des révisions. La boite verte correspond à la gestion des révisions.

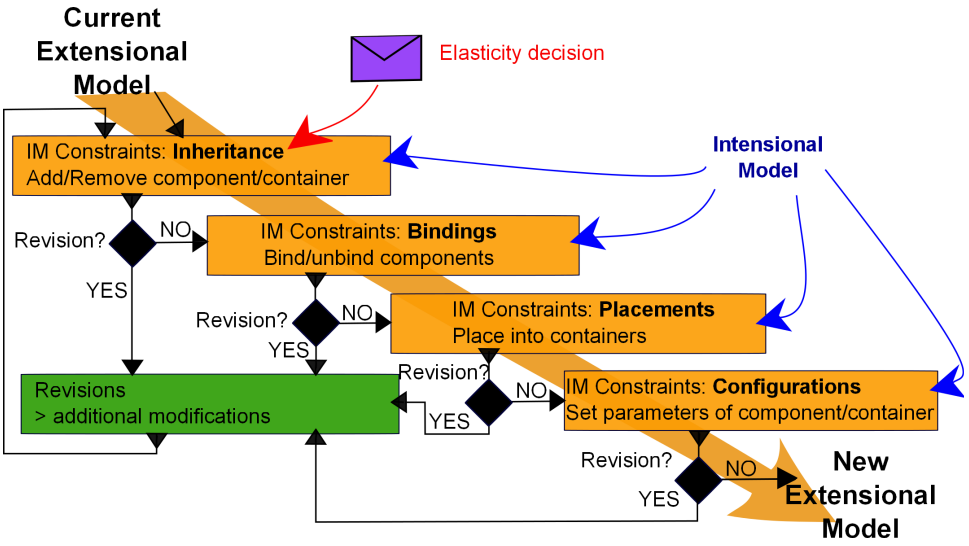


FIGURE 7.3 – Représentation graphique de l'algorithme de planification

Durant son déroulement, l'algorithme va passer par chacune des phases de la séquence principale. Au cours de chacune d'elles, les contraintes du modèle par intention qui correspondent au type de la phase en cours vont être examinées. Si une seule contrainte n'est pas satisfaite, un déroutement vers une phase de révision est opéré. La réalisation d'une révision consiste à corriger une erreur à partir de la description du contexte dans lequel celle-ci est apparue. Pour reprendre l'analogie de la section précédente, une phase décrirait une erreur par exemple de la façon suivante : *"cette voie est sans issue : impossible de continuer"* et la révision consisterait soit à supprimer la partie de l'itinéraire

menant dans cette voie, soit à opérer un demi-tour. La phase de révision est la partie de l'algorithme qui permet d'adresser la gestion des erreurs. Des exemples de révisions sont donnés la partie suivante de ce manuscrit.

L'algorithme 2 présente le détail de l'algorithme de planification proposé. De la ligne 1 à ligne 17, l'algorithme commence par appliquer la décision d'élasticité sur une copie du modèle par extension courant (*em'*) puis parcourt les différentes phases en appliquant les modifications calculées. Au final, *em'* correspond au modèle par extension cible de l'application qui dénote son futur état. De la ligne 5 à la ligne 17, tous les types de contraintes sont successivement parcourus puis appliqués sur le modèle par extension cible. Le premier type de contrainte appliqué est l'*Héritage* lorsque des composants ou des conteneurs sont ajoutés comme c'est le cas au sein des lignes 5 et 30. Tous les autres types de contraintes (*Liaison*, *Placement* et *Configuration*) sont gérés par la fonction nommée *manageConstraints* (lignes 20 à 35). L'appel à cette fonction est effectué dans les lignes 7, 9 et 11, permettant de réaliser la séquence principale. Durant chaque appel à cette fonction, les contraintes d'un type sont récupérées dans le modèle par intention (ligne 21), puis exécutées une-à-une sur le modèle par extension *em'*(ligne 24) tant qu'une révision n'est pas requise. Si aucune révision n'est requise (ligne 25), les modifications déterminées à l'aide de la requête sont appliquées sur *em'* (ligne 26) puis la prochaine contrainte est considérée. En revanche, si une révision est nécessaire (ligne 27), l'algorithme est dérouté vers la phase de révision. Cette étape est dénotée par l'appel à la fonction *handleRevisions* (lignes 38 à 46). Ce déroutement consiste principalement en l'identification entre le code qui identifie la révision requise dans sa description et l'ensemble des révisions connues. Par exemple, si une description de révision signifiant *"il manque une VM vide"* est requise, la correspondance avec la ligne 41 sera faite résultant dans une modification de *em'* qui ajoutera un nouveau conteneur de type VM (ligne 42). A la ligne 30, toutes les modifications additionnelles provenant des révisions sont appliquées sur *em'* avant de recommencer la séquence principale.

L'algorithme de Vulcan est donc constitué d'une séquence principale dont chacune des quatre phases qui la constituent peut résulter en un déroutement lorsqu'un état non satisfaisant - du point de vue des contraintes contenues dans le modèle par intention - est atteint . C'est ce découpage en phases, chacune propre à un type de contrainte, qui permet une approche simplifiée pour l'utilisateur puisque celui-ci n'a qu'à énoncer des contraintes spécifiques à un type unique. L'algorithme compose ensuite toutes ces contraintes entre elles de façon à décrire des scénarios élaborés d'élasticité. D'autre part, c'est aussi ce découpage en phases qui garantit la connaissance qu'a l'utilisateur du déroulement de la planification : celui-ci peut alors suivre les successions des modifications en sachant les phases traversées et les révisions requises. Cela lui offre la connaissance du lien de causalité liant ses contraintes au nouveau modèle par extension obtenu. La séparation des types de contraintes constitue donc le fondement même de l'algorithme proposé. Toutefois, les avantages offerts par cet algorithme deviennent totalement caduques si l'utilisateur mélange les types de contraintes : la complexité des contraintes augmente ce qui réduit automatiquement la maîtrise de la solution par l'utilisateur. Il convient alors de mettre en place des "garde-fous" de sorte à empêcher des hybridations

Algorithme 2 Détails de l'algorithme de planification

ENTRÉE : Une décision d'élasticité provenant de l'Analyse

ALGORITHME :

```
1 : M ← input.getCorrespondingBaseModifications() ▷ Premièrement : obtenir la modification du modèle par extension correspondant à l'entrée
2 : em' ← em.copy() ▷ Copie du modèle par extension courant
3 : endFlag ← false ▷ Définition d'un drapeau signifiant la fin du déroulement de l'algorithme
4 : ▷ Les contraintes du type héritage sont traitées par la ligne suivante
5 : em' ← em'.doAllModifications(M) ▷ Application de la modification de base sur la copie du modèle par extension courant
6 : while not(endFlag) do ▷ Démarrage de la séquence principale
7 :     revisionFlag ← manageConstraints(em',Bindings,im)
8 :     if not(revisionFlag) then
9 :         revisionFlag ← manageConstraints(em',Placements,im)
10 :        if not(revisionFlag) then
11 :            revisionFlag ← manageConstraints(em',Configurations,im)
12 :            if not(revisionFlag) then
13 :                endFlag ← true
14 :            end if
15 :        end if
16 :    end if
17 : end while
18 :
19 :
20 : function manageConstraints(em',t, im)
21 :    typedConstraints ← im.getConstraintsOfType(t) ▷ Récupération des contraintes dans le modèle par intention par type
22 :    revisionFlag ← false
23 :    for c in typedConstraints do
24 :        (M, r) ← c.executeOn(em') ▷ Obtention des modifications dans M, des révisions dans r
25 :        if r is null then ▷ Aucune révision : application des modifications
26 :            em' ← em'.doAllowedModificationsForType(M, t)
27 :        else ▷ /!\ Une révision est requise
28 :            revisionFlag ← true
29 :            handleRevisions(em',c,M,r)
30 :            em'.doAllModifications(M) ▷ Application des modifications nécessitées par les révisions
31 :            break ▷ Retour au début de la séquence principale de l'algorithme
32 :        end if
33 :    end for
34 :    return revisionFlag
35 : end function
36 :
37 : ▷ A partir des déclarations de révisions, trouver un ensemble de modifications correctives
38 : function handleRevisions(em',c,M,r)
39 :    if r.getCode().equals("Lacks one component of type") then
40 :        M ← {new Modification(add, r.getConcernedType()}
41 :    else if r.getCode().equals("Lacks one container of type") then
42 :        M ← {new Modification(add, r.getConcernedType()}
43 :    else if others then
44 :        ...
45 :    end if
46 : end function
47 :
```

SORTIE : Transformer le nouveau modèle par extension dénoté par EM' en un formalisme de sortie compréhensible par l'Exécution

de types de contraintes. A ce titre, le lecteur peut noter que l'algorithme 2 fait mention aux lignes 26 d'une fonction nommée *doAllowedModificationsForType* qui applique des

modifications sur *em'*. Celle-ci ne permet que certaines modifications en fonction du type de contraintes considéré. En fait, cette fonction s'appuie sur une propriété du modèle général d'élasticité offert par Vulcan qui est abordée dans la sous-section suivante.

7.2.2 Type de contraintes et modifications

Une des contributions exposées dans ce manuscrit concerne le modèle d'applications élastiques dans le cloud. Pour rappel, celui-ci repose sur la dualité du modèle par extension qui décrit les éléments d'une application de façon exhaustive, et du modèle par intention qui décrit les contraintes régissant la modification des modèles par extension durant l'élasticité. De la taille contenue de l'espace de l'ensemble de ces modifications élémentaires résulte un nombre de types de contraintes qui est faible mais aussi une propriété visible dans le tableau 7.1.

Pour rappel, les opérations élémentaires de modifications sont les suivantes :

- Ajout/retrait d'un composant/conteneur.
- Lier/délier des composants.
- Placer un composant/conteneur dans un conteneur et son opération inverse.
- Définir la valeur d'un paramètre d'un composant/conteneur.

De la taille contenue de l'espace de l'ensemble de ces modifications élémentaires résulte un nombre de types de contraintes qui est faible mais aussi une propriété visible dans le tableau 7.1.

Type de contrainte	Modifications possibles
Héritage	ajouter/retirer un composant/conteneur
Liaisons	lier/délier des composants
Placements	placer un composant/conteneur au sein d'un conteneur
Configurations	fixer la valeur d'un paramètre d'un composant/conteneur

TABLE 7.1 – Modifications possibles par type de contraintes du modèle par intention

Comme le montre le tableau 7.1, pour chaque type de contraintes les modifications élémentaires possibles forment des ensembles disjoints et complémentaires puisque leurs intersections deux-à-deux sont vides tandis que leur union comprend l'ensemble des modifications possibles. Il s'agit d'une propriété primordiale du modèle de Vulcan car elle permet à Vulcan de garantir le respect du typage des contraintes et de ce fait permet de garantir le respect par l'utilisateur de la séparation des phases de l'algorithme. En effet, l'identification du tableau 7.1 permet de mettre en place une vérification de la non-hybridation des contraintes décrites par l'utilisateur, puisque pour chaque type de contrainte, un ensemble fini d'opérations élémentaires de modification du modèle par extension est attendu, sans recouvrement possible avec les autres types. Il s'agit donc d'une capacité primordiale de Vulcan qui permet de garantir le respect des principes de la solution.

7.3 Synthèse

Les chapitres et sections précédentes ont présenté les concepts essentiels de Vulcan ainsi que les différents objectifs recherchés. D'autres notions moins essentielles pour la compréhension de l'approche offrent toutefois à Vulcan des caractéristiques intéressantes.

Parmi les objectifs mentionnés ci-avant, se trouve la facilitation de la compréhension et de la gestion des erreurs. En ce sens, un concept important qui a fait l'objet de préoccupations tout au long des travaux menés est désigné sous l'appellation *"cerveau-B"* dans [109]. En effet, Vulcan est un système visant à permettre une intelligence dans la gestion de l'élasticité. C'est notamment pour cela qu'il vise à s'intégrer dans la boucle MAPE-K en aval d'une prise de décision et en amont de la modification des briques réelles de l'application. Dans [109], Minsky décrit un système intelligent en le donnant comme comprenant un *cerveau-B* qui ne surveille pas le *"monde extérieur"* (i.e l'état de l'application), mais l'état du *cerveau-A*. Le cerveau-B est en effet un agent qui a pour objectif de surveiller et d'examiner le cerveau-A. Le cerveau-A de son côté est en charge de l'observation et de la gestion du *"monde extérieur"* ce qui correspond ici à la gestion de l'élasticité de l'application. C'est le cerveau-B qui peut, par exemple, décider d'arrêter le cerveau-A si celui-ci rentre dans une boucle infinie. Le cerveau-B a donc un rôle prépondérant pourtant négligé par nombre d'approches. D'une façon générale, Vulcan a été conçu de façon à respecter cette dualité cerveau-A - cerveau-B :

- Le cerveau-A correspond à l'ensemble formé par le modèle par intention, le modèle par extension et la considération des contraintes du modèle par intention vis-à-vis d'une décision d'élasticité appliquée sur le modèle par extension courant.

- Le cerveau-B est en partie fourni dans Vulcan au travers du suivi des révisions et de la progression dans les phases de la séquence principale de l'algorithme. L'algorithme de planification de Vulcan est la pierre angulaire de ce second aspect d'autant plus qu'il facilite la compréhension du fonctionnement du cerveau-A en permettant de suivre de façon précise les répercussions des contraintes du modèle par intention sur les évolutions du modèle par extension. Vulcan permet également de suivre les erreurs lors de l'établissement de la planification comme cela peut se produire, si par exemple, aucune révision n'est définie pour une description donnée d'erreur : l'utilisateur et/ou les autres éléments de la boucle MAPE-K sont ainsi notifiés du dysfonctionnement car aucune solution n'a été trouvée.

Les concepts présentés par Minsky sont également présents dans l'implantation de Vulcan et seront présentés dans la partie suivante de ce manuscrit. Le cerveau-B peut faire l'objet d'améliorations notamment pour la détection anticipée de problèmes : ceci sera également abordé dans la suite de ce document.

Ce chapitre a présenté l'algorithme de planification de Vulcan. Il s'agit d'une approche innovante qui a recours au raisonnement par révision, une notion issue des travaux portant sur l'intelligence artificielle. Cet algorithme est à la fois à la base de la simplicité d'utilisation de Vulcan, mais est aussi garant de ses capacités de couverture des scénarios possibles d'élasticité. Cet algorithme s'appuie sur les spécificités du méta-modèle du

modèle par extension afin de garantir les caractéristiques recherchées de Vulcan. Vulcan est donc une approche composite issue de briques dont les qualités s'auto-entretiennent : le modèle par extension permet la mise en place de l'algorithme de Vulcan qui lui même permet la description par intention, laquelle offre une maîtrise simple de la modification du modèle par extension et fournit des contraintes à l'algorithme, qui lui-même garantit la simplicité de la maîtrise de la modification du modèle par extension. L'algorithme de planification n'est donc qu'une partie de l'édifice que constitue Vulcan. Pour rappel, celui-ci se compose des apports suivants :

- **Le modèle par Extension** : un modèle représentant l'architecture d'une application lors de son exécution. Ce modèle peut être modifié faisant de lui le creuset des calculs nécessaires à la planification dans la boucle autonomique MAPE-K.
- **Le modèle par Intention** : un modèle de description des modifications licites du précédent modèle. Celui-ci se base sur un formalisme innovant qui garantit une forte expressivité, une forte couverture des scénarios possibles et une forte extensibilité.
- **Un algorithme de planification** novateur tant par son mode de raisonnement issu de l'intelligence artificielle, que par ses capacités de calcul et sa simplicité d'utilisation. Cet algorithme permet une gestion aisée et compréhensible des erreurs.
- Une approche homogène à la confluence de travaux de modélisation et de travaux d'intelligence artificielle [109]. Cette approche offre une planification simple et qui étend les capacités de gestion des scénarios d'élasticité.

Grâce à ces concepts et contributions, l'approche adresse des cas d'usage à la fois complexes et réels. La partie suivante permet d'apprécier ce dernier point et plus généralement de valider l'approche de Vulcan.

Troisième partie

Implantation et Expérimentations

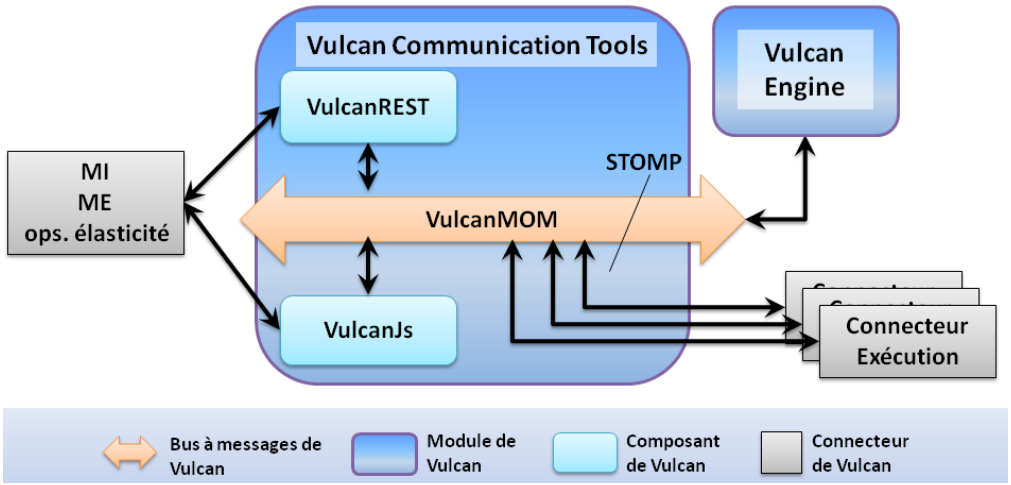
Chapitre 8

Implantation

Sommaire

Le prototype implanté durant les travaux de thèse se nomme *Vulcan*. Il est composé de code écrit en Java [1] et Node.js [154]. Ce prototype comprend deux modules : le *Vulcan Engine* et le *Vulcan Communication Tools* comme illustrés par le schéma 8.1. Le *Vulcan Engine* est le module constitué des éléments de calcul de la planification tandis que les éléments au sein du *Vulcan Communication Tools* sont des éléments permettant d'interagir avec le *Vulcan Engine*. Ce second module comprend notamment une interface web graphique appelée *VulcanJs*, une interface web RESTful [57] (*VulcanREST*) et un bus à messages (MOM) reposant sur ActiveMQ. Ce bus à messages (le VulcanMOM) est totalement asynchrone et permet de faire communiquer les agents de Vulcan qui sont eux aussi asynchrones. Il repose sur STOMP, un protocole d'échange de messages au format texte ce qui permet un excellent découplage du code des éléments au sein des modules : VulcanJs, codé en Node.js peut ainsi aisément dialoguer avec des éléments codés en Java tels que ceux au sein du Vulcan Engine. VulcanREST offre à Vulcan une API permettant de s'interfacer avec une brique de décision et une brique d'exécution. Le choix de REST est justifié par le fait que ce mode de communication permet un découplage complet avec

les autres briques, tout en permettant un mode d'accès qui passe bien les pare-feu. De son côté, VulcanJs offre une interface graphique qui offre un bon suivi à un utilisateur.



Architecure de Vulcan au niveau de ses modules. "MI" désigne le Modèle par Intention, "ME" désigne le Modèle par Extension, "ops. élasticité" désigne les opérations d'élasticité.

FIGURE 8.1 – Architecture de Vulcan

Cette notion de découplage et ce bus à messages offrent à Vulcan une grande flexibilité quant à sa configuration ce qui est à la base d'une caractéristique de Vulcan : son auto-élasticité. Avant d'aborder ce point par la suite, ce chapitre va commencer par décrire l'architecture utilisée pour la validation et à une granularité plus fine pour le *Vulcan Engine*.

8.1 Le *Vulcan Engine*

Le Vulcan Engine se situe au cœur des préoccupations de la thèse puisque c'est lui qui a été - et qui est encore - la principale source d'innovations en matière de recherche autour de l'approche proposée. En effet, ce moteur doit permettre à la fois l'implantation de l'algorithme par révision mais aussi l'utilisation des implantations des autres contributions de la thèse, c'est-à-dire les modèles par extension et par intention. Ce moteur a été pensé pour être modulaire et performant avec une implantation simple et compréhensible de sorte à faciliter son utilisation et son perfectionnement futurs par d'autres personnes que l'auteur actuel de son code.

Le Vulcan Engine se compose ainsi de cinq gestionnaires, chacun servant à la prise en charge d'une des phases de l'algorithme exposé précédemment :

1. Le *Vulcan Planification Manager* est en charge d'orchestrer les différentes phases de l'algorithme. C'est aussi lui qui gère les contraintes d'héritage et assure que toute entité du modèle par extension aura un nom unique.

[1]Le code Java représente un volume d'environ 40.000 lignes de code

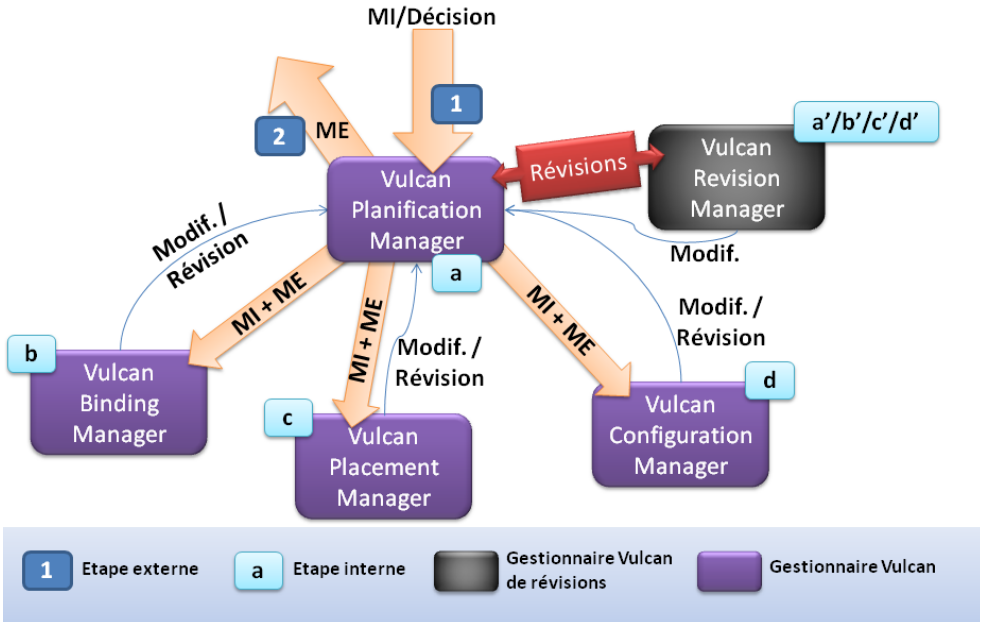
2. Le *Vulcan Binding Manager* gère l'ensemble des contraintes de liaisons entres les composants d'une application élastique.

3. Le *Vulcan Placement Manager* permet de gérer les contraintes de placement des composants et conteneurs au sein de conteneurs.

4. Le *Vulcan Configuration Manager* assure la prise en charge des contraintes de configuration des composants et des conteneurs.

5. Le *Vulcan Revision Manager* est l'entité qui permet d'offrir un mécanisme de révisions configurables.

Tous ces gestionnaires ont été implantés en Java. Ils communiquent au travers de messages sous la forme de chaines de caractères : il n'y a donc pas d'échange d'objets. Le *Vulcan Planification Manager* constitue l'unique point d'entrée du Vulcan Engine. L'architecture du Vulcan Engine est illustrée par le schéma 8.2 qui illustre également le déroulement de l'algorithme de planification de Vulcan.



Détail des échanges effectués entre les gestionnaires du Vulcan Engine. MI désigne le Modèle par Intention, ME désigne le Modèle par Extension.

FIGURE 8.2 – Architecture du Vulcan Engine

Outre la prise en charge de leur phase de calculs, les gestionnaires du Vulcan Engine assurent également la vérification du type des opérations décidées. Pour rappel, à chaque phase de l'algorithme correspond un ensemble d'opérations de modifications du modèle par extension courant. Par exemple, le placement d'un composant ne peut être réalisé que durant la phase de Placement de l'algorithme et en dehors de celle-ci, une telle

opération doit être interdite. En ce sens, les gestionnaires du Vulcan Engine filtrent les modifications calculées en ne gardant que celles permises par leur phase d'algorithme. Les gestionnaires permettent également l'expression de demandes de révisions de sorte à respecter l'algorithme principal du Vulcan Engine.

Le choix de filtrer les opérations autorisées au sein des gestionnaires - avec un éventuel avertissement - a été préféré à celui d'un départ en erreur mettant brutalement fin aux calculs. Ce choix s'inscrit dans la philosophie de Vulcan de toujours faciliter le traitement ce qui ne peut provenir que d'une simple erreur d'écriture des contraintes du modèle par intention. Ce choix vise à éviter des moments de désarroi pour un administrateur utilisant Vulcan en limitant autant que possible les sorties en erreur à l'exécution.

Cette section vient de présenter le *Vulcan Engine*, le moteur de calculs pour la planification implanté pour la validation de Vulcan. Ce moteur est constitué de modules qui font usage des deux modèles issus des contributions de la thèse : le modèle par intention et le modèle par extension. Chacun de ces modèles a fait l'objet d'une implantation et les sections suivantes les décrivent.

8.2 Implantation du Modèle par Extension

Le modèle par extension permet la représentation de la connaissance de l'état de l'application. Il repose sur un méta-modèle qui bien que simple n'a pas présenté de restriction de cas d'usages. De par le fait qu'il représente l'état courant de l'application et qu'il sert de base à l'introspection, le modèle par extension se doit d'avoir une implantation en rapport avec son rôle de base de connaissance. Par ailleurs, le modèle par extension doit également être à même de proposer une gestion simple et rapide des modifications. Ces deux constats sont à la fois simples et évidents mais ils sont par nature opposés. La sous-section suivante décrit plus précisément cette problématique ainsi que l'approche utilisée pour l'implantation du modèle par extension qui permet de tenir compte de ces constats.

8.2.1 Dualité objet-représentation

Une gestion simple des modifications implique d'avoir des méthodes facilement accessibles comme le permet la programmation orientée objet. De plus, la programmation orientée objet offre globalement de bonnes performances grâce à une compilation en code machine. Cependant, le paradigme de l'objet se prête relativement mal à une représentation d'état pouvant être facilement introspectée et compréhensible pour un humain. En ce sens, le travail autour de JMX, un ensemble d'outils permettant de surveiller des applications Java, montre la complexité d'une telle compréhension. A contrario, des formalismes comme XML, YAML ou JSON facilitent la compréhension pour utilisateur humain, grâce à des données pouvant être facilement lisibles même si celles-ci peuvent être un peu verbeuses comme avec XML. Par ailleurs, XML se prête très bien à l'utilisation de requêtes ensemblistes durant l'introspection. L'inconvénient de ces formalismes est leur absence de méthodes de manipulation.

Face à ces constats et face aux nécessités du modèle par extension, un travail préparatoire autour de l'implantation du modèle par extension a donc consisté à allier la simplicité de manipulation des objets à la facilité de compréhension des formalismes comme XML, YAML et JSON. **L'implantation du modèle par extension repose donc sur la dualité objet - formalisme de représentation**. La forme objet du modèle par extension permet d'effectuer des modifications de façon rapide, simple et performante, tandis que sa représentation offre une introspection compréhensible et facilitant l'usage du formalisme du modèle par intention. Pour un modèle par extension donné, ces deux formes sont le miroir l'une de l'autre. Si le langage java a été retenu pour la partie objet, le XML a quant à lui été choisi pour la représentation. Le listing 8.1 donne un exemple d'une telle représentation pour l'application Springoo. Dans cet exemple, tous les composants sont déployés dans des VMs séparées, elles mêmes placées au sein d'un même IaaS (ici un OpenStack interne à Orange).

Le choix du formalisme XML a été principalement motivé par le fait que celui-ci se prête très bien à l'utilisation de requêtes ensemblistes constitutives du formalisme du modèle par intention. Ces requêtes servent effectivement à décrire les contraintes applicatives qui, une fois appliquées sur la forme de représentation XML du modèle par extension courant, renvoient des modifications de ce même modèle ou des demandes de révisions. Si les demandes de révisions sont primordiales pour le cheminement global dans l'algorithme du Vulcan Engine, les modifications du modèle par extension le sont également puisqu'elles permettent une évolution de l'architecture de l'application afin que celle-ci vérifie les contraintes fixées. Toutefois la gestion de ces modifications n'est pas immédiate puisque le Vulcan Engine doit également assurer tout au long de la planification la cohérence du modèle par extension. La sous-section suivante détaille ce point précis.

Listing 8.1 – Modèle par Extension de l'application Springoo sous forme de représentation XML

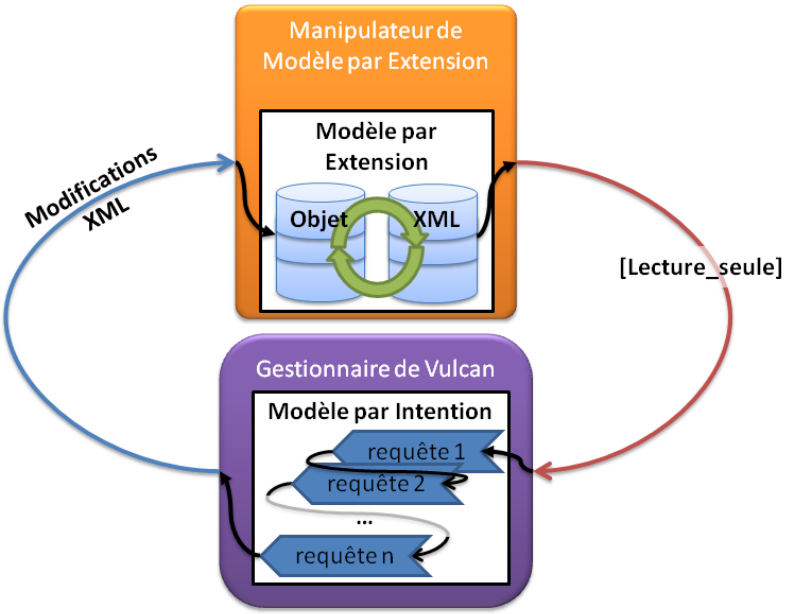
```
<extensionalModel>
<!— liste des instances de composants—>
        <!— instance du serveur Apache qui a une liaison vers Jonas:1
         son conteneur est vm:1—>
<instance type="Apache" name="1">
<container type="vm" name="1"/>
        <binding endType="Jonas" name="1"/>
</instance>
<instance type="Jonas" name="1">
        <container type="vm" name="2"/>
        <binding endType="MySQL" name="1"/>
</instance>
<instance type="MySQL" name="1">
        <container type="vm" name="3"/>
</instance>

<!— liste des VMs avec leurs profils de IaaS—>
        <!— conteneur vm:1 son profil materiel est m1.small
         son conteneur est iaas:1—>
<container type="vm" name="1">
        <container type="iaas" name="1"/>
        <property name"profile">
                <value>m1.small</value>
        </property>
</container>
        <container type="iaas" name="1"/>
        <property name"profile">
                <value>m1.small</value>
        </property>
</container>
<container type="vm" name="3">
        <container type="iaas" name="1"/>
        <property name"profile">
                <value>m1.small</value>
        </property>
</container>

<!— IaaS de destination —>
        <!— conteneur iaas:1
         il propose ici un seul profil materiel pour les vm : "m1.small"—>
<container type="iaas" name="1">
        <property name"id">
                <value>Orange_OpenStack</value>
        </property>
        <property name"profiles">
                <value>
<profile name="m1.small" ram="1024" cpu="1" arch="x86" disksize="10GB"/>
                </value>
        </property>
</container>
</extensionalModel>
```

8.2.2 Maintien de la cohérence du Modèle par Extension lors des calculs de la Planification

Le traitement des modifications a donné lieu à un travail important autour du maintien de la cohérence du modèle par extension durant la planification. Il faut effectivement éviter des manipulations hasardeuses conduisant à un état-cible incohérent et inapplicable. Par exemple, avant que le placement d'un composant au sein d'un conteneur ne soit effectif, le moteur doit d'abord vérifier que ledit conteneur existe et qu'il s'agit bel et bien d'un conteneur. De même, pour les liaisons entre composants, deux composants ne peuvent évidemment être liés que si chacun d'eux existe. Ces exemples de vérifications sont basiques mais ils illustrent toutefois bien les problématiques du maintien de cohérence. La mise en place de cette vérification de modèle s'appuie sur un choix d'implantation qui empêche une modification directe du modèle par extension courant. De façon plus précise, le Vulcan Engine applique lui-même des séquences de modifications sur la forme objet du modèle par extension courant tandis que sa représentation en XML n'offre qu'un accès en lecture. C'est durant cette phase que la vérification de cohérence est effectuée. Le schéma 8.3 illustre le fonctionnement générique de l'implantation du modèle par extension.



Ce schéma montre les échanges réalisés dans le Vulcan Engine. Les flèches indiquent les flux de données. Les requêtes du modèle par intention accèdent en lecture seule à la représentation XML du modèle par extension.

FIGURE 8.3 – Détails de la manipulation de l'implantation du Modèle par Extension

Empêcher toute modification du modèle par extension est certes une contrainte faite à l'utilisateur, mais elle permet la mise en place de la vérification de cohérence. D'autre

part, cette approche offre plusieurs autres avantages :

- L'implantation du manipulateur de modèle par extension repose sur BaseX, une base de données NoSQL [69, 96, 158] orientée document. Dans cette base, les opérations de lecture sont très rapides alors que les opérations en écriture sont bien plus lentes [96]. La partie objet de l'implantation du modèle par extension permet en revanche de garantir de bonnes performances en écriture. Le choix d'interdire les modifications directes de la représentation XML du modèle par extension permet ainsi de meilleures performances.

- Le fait de rassembler les modifications sur la forme objet du modèle par extension permet un suivi à faible coût de l'ensemble des modifications opérées sur le modèle par extension. Ceci s'inscrit dans la continuité des principes de Minsky en permettant la mise en place d'un cerveau-A exposant son état et son évolution pour qu'un cerveau-B puisse le surveiller.

- La partie objet de l'implantation du modèle par extension bénéficie d'un ensemble très complet d'outils permettant de calculer des répercussions de modifications, des différentiels, des unions et des copies de modèles. Empêcher les modifications directes du modèle par extension courant au niveau de sa représentation permet donc à l'utilisateur de bénéficier de l'intégralité de ces outils. Il ne lui reste alors qu'à écrire des contraintes dans le modèle par intention et à énoncer les modifications requises sans avoir à se soucier de leur bonne répercussion.

Face à ces avantages, le choix d'empêcher la modification directe de la représentation du modèle par extension s'est donc imposée. Les requêtes ensemblistes du modèle par intention permettent ainsi de décrire des séquences de modifications que le Vulcan Engine prend en charge et applique sur la partie objet. Cela permet notamment de maintenir la cohérence du modèle par extension lors de la planification. Lors de cette prise en charge, le choix a été fait de filtrer les modifications valides et de rejeter les autres plutôt que de sortir en erreur.

Ainsi, **les requêtes ensemblistes accèdent en lecture seulement à la représentation XML du modèle par extension et retournent des séquences de modifications et/ou des demandes de révisions**. Le Vulcan Engine applique ces modifications sur la forme objet du modèle par extension mais uniquement après vérification. La sous-section qui suit donne une liste des modifications possibles du modèle par extension.

8.2.3 Modifications du Modèle par Extension et révisions

L'implantation du modèle par extension présente une dualité objet - représentation. Sa manipulation en écriture revient au Vulcan Engine qui applique donc les modifications retournées par les requêtes ensemblistes du modèle par intention. Le Vulcan Engine traite également les demandes de révisions. Les demandes de révisions et les modifications sont renvoyées directement sous une forme XML et le listing 8.2 en montre deux exemples

pour des modifications inverses l'une de l'autre. La première va de la ligne 2 à la ligne 5 et signifie l'ajout d'un conteneur de type "vm", c'est-à-dire un ajout de machine virtuelle. De façon inverse, la seconde modification (lignes 8-11) indique le retrait de cette même machine virtuelle.

Listing 8.2 – Exemple de modifications du Modèle par Extension

```
1 <!— modification : ajouter une VM —>
2 <modification type="add">
3         <container type="vm">
4         </container>
5 </modification>
6
7
8 <!— modification inverse : retrait de cette VM —>
9 <modification type="remove">
10        <container type="vm" name="1">
11        </container>
12 </modification>
```

Les demandes de révision sont également retournées sous forme XML tout comme le montre le listing 8.3. Cet exemple montre une demande de révision retournée lors de la phase de placement dont la contrainte de placer chaque composant dans une VM séparée ne peut être satisfaite en raison d'un manque de conteneur de type vm pour le composant *Jonas :1*. Cette demande de révision permettra, après traitement par le Vulcan Revision Manager, de rajouter les conteneurs nécessaires. Outre la modification de l'architecture de l'application, cette demande de révision permettra également à un cerveau-B de suivre les traitements dans le Vulcan Engine notamment grâce au champ *msg* qui renseigne la description du problème rencontré lors des calculs.

Listing 8.3 – Exemple de demande de révision

```
1 <!— revision : il manque une VM pour placer le composant Jonas:1—>
2 <revision ownertype="Jonas" ownername="1" targettype="vm" code="0X1001"
3         msg="One container of type vm is missing" />
```

C'est ce type de retour constitué de descriptions XML de modifications et/ou de demandes de révisions qui est renvoyé par les requêtes ensemblistes du modèle par intention. Ces retours vont servir - durant la planification - à faire évoluer le modèle par extension afin qu'il satisfasse l'ensemble des contraintes de l'application.

L'implantation du modèle par extension a nécessité un travail important et primordial de conception et de réalisation qui est complètement utilisé au sein des différents gestionnaires du Vulcan Engine. Ces gestionnaires implantent tous une phase de l'algorithme de planification proposé dans ce manuscrit. Pour cela, chacun d'eux applique le type de contraintes du modèle par intention qui est propre à sa phase. La section suivante décrit l'implantation du modèle par intention et plus précisément de celle de son formalisme.

8.3 Implantation du Modèle par Intention

Le modèle par intention est au cœur des travaux de thèse. Il repose effectivement sur un formalisme innovant permettant de décrire de façon simple des contraintes sur des architectures applicatives. Le formalisme du modèle par intention permet en effet d'introspecter le modèle par extension courant sous sa forme XML et d'en décrire les modifications et révisions nécessaires pour parvenir à une architecture-cible respectant l'ensemble des contraintes exprimées. La prochaine sous-section décrit le langage retenu pour la description des contraintes du modèle par intention.

8.3.1 Langage des contraintes

L'implantation du modèle par intention nécessite la description de contraintes. Ces contraintes sont décrites en *XQuery*, un langage de requêtes ensemblistes permettant de lire, créer et modifier des arborescences XML. Ces requêtes accèdent donc en lecture à la représentation XML du modèle par extension courant et retournent des arborescences XML décrivant les modifications requises. XQuery est un standard faisant partie des recommandations du World Wide Web Consortium (W3C) depuis 2007. Les requêtes écrites dans ce langage sont composées d'expressions dites *FLOWR*, en rapport avec les mots-clés de XQuery qui les composent. Ces mots-clés sont similaires à ceux des expressions en langage SQL : select from, where, order by.

- *For* permet de signifier le parcours sur des ensembles pouvant être désignés par des expressions de filtrage.
- *Let* sert à la définition de variable temporaire afin de - par exemple - stocker un résultat intermédiaire.
- *Order* signifie qu'un tri sera appliqué sur le résultat renvoyé.
- *Where* permet de ne garder que des éléments remplissant une condition.
- *Return* exprime le résultat retourné par la requête. Ce résultat peut être mis en forme de sorte à pouvoir notamment recréer une arborescence XML.

Le langage XQuery utilise les expressions XPath qui permettent une navigation conditionnelle au sein de l'arborescence XML tout comme FScript fait l'usage d'expressions FPath. A titre d'illustration, le listing 8.4 montre un exemple de requête XQuery utilisant une navigation XPath. Appliquée sur la représentation XML du modèle par extension 8.1, cette requête renvoie toutes les instances du composant Jonas. Pour cela, elle parcourt l'ensemble des instances de composants dont le type est égal à *Jonas* (ligne 2) et les retourne tels qu'ils sont stockés dans le modèle par extension (ligne 3).

Listing 8.4 – Requête XQuery renvoyant tous les Jonas pour le modèle par extension 8.1

```
1         <!-- Requete -->
2         for $i in $ExtensionalModel/instance[@type="Jonas"]
3         return $i
4
5         <!-- Resultat -->
6         <instance type="Jonas" name="1">
7                 <container type="vm" name="2"/>
8                 <binding endType="MySQL" name="1"/>
9         </instance>
```

Pour aller plus loin, les listings 8.6 et 8.7 montrent comment des contraintes peuvent être vérifiées à la fois par l'introspection du modèle par extension transitoire (i.e. une architecture incomplète au cours des calculs de la planification) et par le calcul de séquences de modifications. Le modèle par extension transitoire est donné par le listing 8.5. Dans celui-ci aucun composant n'est placé. Le listing 8.6 montre une requête XQuery permettant d'en calculer les modifications afin de satisfaire une contrainte requérant que tous les composants de l'application soient placés dans une seule et même VM. Le listing 8.7 donne quant à lui une requête relative à une contrainte requérant que chaque composant soit placé dans sa propre VM. Ces listings font appel à des fonctions externes - elles aussi simples - mais non écrites ici par souci de lisibilité et de facilité de compréhension pour le lecteur.

Listing 8.5 – Modèle par Extension transitoire de l'application Springoo

```
1 <extensionalModel>
2         <!-- liste des instances de composants-->
3         <instance type="Apache" name="1"/>
4         <instance type="Jonas" name="1"/>
5         <instance type="MySQL" name="1"/>
6
7         <!-- liste des VMs-->
8         <container type="vm" name="1"/>
9         <container type="vm" name="2"/>
10        <container type="vm" name="3"/>
11 <extensionalModel/>
```

Dans la requête du listing 8.6, les composants non placés sont stockés dans la variable temporaire *$unplacedComponents* à la ligne 2. Ensuite la requête parcourt chaque composant non-placé (ligne 3) et retourne une modification du modèle par extension courant (ligne 4). Le résultat retourné par l'exécution de cette seule requête est exposé de la ligne 7 à la ligne 15. Il sera appliqué par le Vulcan Engine.

Listing 8.6 – Requête XQuery calculant des modifications de placement

```
1  <!-- Requete -->
2  let $unplacedComponents := getUnplacedComponent()
3  for $component in $unplacedComponents
4  return <modification type="place"
       target="vm:1">{$component}</modification>
5
6  <!-- Resultat -->
7  <modification type="place" target="vm:1">
8          <instance type="Apache" name="1"/>
9  </modification>
10 <modification type="place" target="vm:1">
11         <instance type="Jonas" name="1"/>
12 </modification>
13 <modification type="place" target="vm:1">
14         <instance type="MySQL" name="1"/>
15 </modification>
```

Dans le listing 8.7, la requête est similaire à celle du listing 8.6 hormis pour le choix de la vm de placement. Celui-ci était effectivement codé en dur, là où dans cette présente requête, il est calculé en direct. Les conteneurs vides sont ainsi stockés dans la variable temporaire *$emptyContainers* (ligne 3). Le parcours des composants non-placés (ligne 4) retourne des modifications de placement (ligne 7) de chaque composant dans un conteneur de type vm qui lui est propre (ligne 5). Ce conteneur est désigné par son identifiant qui est extrait à la ligne 6 et spécifié dans le champ *target* de la modification. Le résultat obtenu est écrit entre les lignes 10 et 18.

Listing 8.7 – Requête XQuery permettant de calculer toutes les modifications nécessaires au placement de chaque composant non déjà placé dans un conteneur vide de type vm

```
1  <!-- Requete -->
2  let $unplacedComponents := getUnplacedComponent()
3  let $emptyContainers := getEmptyContainers()
4  for $component in $unplacedComponents
5          let $container :=
               $emptyContainers[index-of($unplacedComponents,
               $component)]
6          let $id := getId($container)
7  return <modification type="place"
       target="{$id}">{$component}</modification>
8
9  <!-- Resultat -->
10 <modification type="place" target="vm:1">
11         <instance type="Apache" name="1"/>
12 </modification>
13 <modification type="place" target="vm:2">
14         <instance type="Jonas" name="1"/>
15 </modification>
16 <modification type="place" target="vm:3">
17         <instance type="MySQL" name="1"/>
18 </modification>
```

Dans le prototype implanté pour la validation des contributions de la thèse, des bibliothèques de contraintes ont été créées avec une documentation expliquant leur signification. Ces bibliothèques sont extensibles ce qui permet à Vulcan de ne pas être restreint quant aux contraintes adressées. Il s'agit d'un des objectifs de Vulcan qui vise à être **une solution de gestion de l'élasticité qui soit extensible**. Les deux contraintes exposées dans les listings 8.6 et 8.7 sont ainsi accessibles par des fonctions présentes dans les bibliothèques du prototype. Elles se retrouveront d'ailleurs dans les modèles par intention utilisés pour la validation de l'approche. Le listing 8.8 donne ces fonctions : la fonction de la ligne 2 correspond à un équivalent de la requête exposée dans le listing 8.6 tandis que l'équivalent de la requête du listing 8.7 est présente ligne 5.

Listing 8.8 – Fonctions XQuery présentes dans les bibliothèques du prototype

```
1  <!-- Placer tous les composants dans une seule VM -->
2  placement:placeAllOne()
3
4  <!-- Placer chaque composant dans sa propre VM -->
5  placement:placeOneOne()
```

Le listing 8.8 montre également - en partie - une qualité de Vulcan : sa facilité d'utilisation. L'utilisateur peut ainsi utiliser les requêtes déjà écrites et présentes dans les bibliothèques du prototype ce qui permet d'en faciliter la prise en main. Les bibliothèques de Vulcan incluent également des *traitants de révisions*, c'est-à-dire du code permettant

de faire correspondre à une demande de révision, une nouvelle séquence de modifications dans le but de corriger un état transitoire problématique du modèle par extension. Dans l'exemple de demande de révision donné par le listing 8.3, le traitant retourne une modification requérant l'ajout d'une nouvelle VM. Le Vulcan Revision Manager procède en réalité à l'exécution du bon traitant à partir de l'établissement d'une correspondance avec le code de la demande (ici 0X1001). Tous les traitants sont également écrits en XQuery afin de pouvoir introspecter le modèle par extension problématique et la demande de révision. Les requêtes des traitants sont également présentes dans une des bibliothèques de Vulcan qui est elle aussi extensible.

Cette sous-section a décrit le langage de requêtes ensemblistes utilisé pour l'implantation des contraintes mais également des révisions. Elle a également donné des exemples basiques d'expression de contraintes architecturales et montré une partie de la simplicité d'apprentissage de Vulcan. Les contraintes du modèle par intention sont écrites en langage XQuery. Le prototype offre des bibliothèques de requêtes qui permettent non seulement une prise en main simple et rapide pour un utilisateur débutant, mais également une grande flexibilité et de bonnes capacités d'extension pour l'utilisateur averti. Toutefois, l'expression des contraintes ne constitue qu'une partie du formalisme du modèle par intention. La sous-section suivante aborde donc le formalisme global du modèle par intention.

8.3.2 Formalisme complet du Modèle par Intention : XQuery dans XML

Le formalisme complet du modèle par intention constitue une des parties visibles pour l'utilisateur de Vulcan avec le modèle par extension et l'API de notification d'opération d'élasticité. De façon à rester cohérent au niveau de l'approche proposée, le formalisme du modèle par intention est en XML. Il intègre l'ensemble des requêtes XQuery qui spécifient les contraintes de l'application.

Pour rappel, il existe quatre types de contraintes : héritage, liaison, placement, configuration. Ces quatre types de contraintes sont décrits dans leurs sections propres comme le montre le listing 8.9. Ce listing correspond à un modèle par intention pour l'application Springoo.

De la ligne 3 à la ligne 10, les contraintes d'héritage sont fixées. Les contraintes d'héritage ne sont pas décrites en XQuery car cela n'est pas apparu comme étant nécessaire. Les champs *init* permettent de mentionner le nombre initial exact d'instances de composants et de conteneurs. Un champ *max* peut également être renseigné afin de limiter ce nombre.

Ensuite, les contraintes de liaisons sont décrites de la ligne 13 à la ligne 20. La première de ces deux requêtes (ligne 15) permet de lier chaque composant Apache à au moins un composant Jonas et réciproquement de lier chaque Jonas à exactement un Apache. La seconde (ligne 18) permet quant à elle de lier chaque composant Jonas à exactement un MySQL et de lier chaque MySQL à au moins un Jonas.

Entre les lignes 23 et 30, sont décrites les contraintes de placement. La requête à la ligne 25 est une des contraintes données en exemple dans la partie précédente : chaque

composant doit être placé dans sa propre VM. La seconde requête présente à la ligne 28 permet de n'avoir que des VMs utiles, c'est-à-dire de ne conserver que des VMs hébergeant des composants.

Enfin, les contraintes de configurations sont écrites entre les lignes 33 et 37. Il s'agit d'une unique requête permettant de placer un marqueur sur toute VM embarquant un composant Apache afin qu'une adresse IP publique lui soit assignée par la brique d'exécution.

Listing 8.9 – Modèle par Intention pour l'application Springoo

```
1  <intensional-model>
2          <!-- Heritage -->
3          <components>
4                  <component name="Apache" init="1"/>
5                  <component name="Jonas" init="1"/>
6                  <component name="MySQL" init="1"/>
7          </components>
8          <containers>
9                  <container name="vm" init="0"/>
10         </containers>
11
12         <!-- Liaisons -->
13         <bindings>
14                 <binding-query>
15                         local:bindExactlyOneAll("Apache","Jonas")
16                 </binding-query>
17                 <binding-query>
18                         local:bindAllExactlyOne("Jonas","MySQL")
19                 </binding-query>
20         </bindings>
21
22         <!-- Placements -->
23         <placements>
24                 <placement-query>
25                         <!-- un composant par vm-->
26                         local:placeOneOne()
27                 </placement-query>
28                 <placement-query>
29                         <!-- pas de vm sans composant en son sein -->
30                         local:purgeEmptyContainers("vm")
31                 </placement-query>
32         </placements>
33
34         <!-- Configurations -->
35     <configurations>
36                 <configuration-query>
37                         local:setPublicIpOn(("Apache"), "vm")
38                 </configuration-query>
39     </configurations>
40 </intensional-model>
```

Il s'agit là d'un exemple relativement simple de modèle par intention qui permet néanmoins de donner une idée de la faible complexité de la tâche. Dans les chapitres 9 et 10, des scénarios autrement plus complexes mais avec une expression restant toujours simple seront montrés.

Le formalisme complet du modèle par intention permet au Vulcan Engine d'assurer le déroulement de l'algorithme de planification. Ainsi chaque gestionnaire reçoit les requêtes dont le type correspond à la phase de l'algorithme qu'il implante. Le Vulcan Binding Manager reçoit par exemple les requêtes de type *Liaisons* identifiées par les balises *<bindings>...</bindings>* dans le modèle par intention. En fonction de l'application, un type de contrainte peut évidemment être vide.

Cette section a présenté l'implantation du modèle par intention. Son formalisme permet de décrire des contraintes qui, à partir de l'introspection du modèle par extension courant, permettent de décrire des séquences de modifications ou des demandes de révisions. Ces séquences de modifications et ces demandes de révisions sont ensuite traitées par le Vulcan Engine de sorte à parvenir à un nouveau modèle par extension respectueux des contraintes souhaitées. Ces contraintes sont écrites en XQuery, un langage de requêtes ensemblistes standard, et sont renseignées dans l'arborescence XML du modèle par intention. La section suivante aborde une caractéristique intéressante de Vulcan : son auto-élasticité. Elle dresse également un bilan de ce chapitre.

8.4 Auto-élasticité et synthèse

Vulcan est une application composée de modules. Ces modules communiquent par un bus à messages, le VulcanMOM. L'auto-élasticité de Vulcan a été imaginée dès sa conception afin de pallier des problèmes de fiabilité et de performances. Cette architecture permet par exemple d'avoir un serveur frontal VulcanJs avec une interface VulcanREST sur un autre serveur en tête d'un ou plusieurs Vulcan Engines eux-mêmes placés sur d'autres serveurs pouvant avoir un ou plusieurs gestionnaires de chaque type. L'intérêt est d'avoir une solution de gestion de la planification de l'élasticité qui soit évolutive, non seulement au niveau des contraintes pouvant être exprimées, mais également au niveau de la façon de les gérer. Il s'agit là de perspectives orientées recherche qui seront exposées dans la conclusion de ce manuscrit. Toutefois, l'implantation actuelle de Vulcan propose une première version d'auto-élasticité servant à des fins de preuve de concepts.

L'auto-élasticité de Vulcan s'appuie sur un *bootstrap* qui permet à un administrateur de renseigner un modèle par intention pour Vulcan. Ce modèle est ensuite traité par le bootstrap qui procède ensuite au déploiement proprement dit de Vulcan, c'est-à-dire celui de l'application Vulcan en tant que telle, celle qui permet la planification d'une autre application. De façon plus précise, le Vulcan Bootstrap n'est ni plus ni moins qu'une application Vulcan déployée au sein d'une même machine pouvant avoir une interface VulcanREST mais pas d'interface VulcanJS de façon automatisée. Le Vulcan Bootstrap

fait usage du Vulcan Deployer, un canevas de déploiement qui a notamment permis le déploiement et l'élasticité de Springoo sur un cluster d'hyperviseurs Virtualbox. A tout moment, il est par exemple possible de rajouter un nouveau Vulcan Engine pour accueillir un nouvel utilisateur. L'auto-élasticité de Vulcan est une propriété remarquable dans le sens où elle offre une capacité d'évolution intéressante à des fins de recherche. Le Vulcan Bootstrap constitue ainsi une partie du cerveau-B décrit par Minsky et laisse entrevoir le potentiel théorique de Vulcan.

Conclusion

Ce chapitre a expliqué l'implantation des concepts théoriques de Vulcan dans un prototype caractérisé par une architecture modulaire. Ce prototype implante une première version d'auto-élasticité. Vulcan fait usage d'un modèle par extension se présentant sous une double forme, chacune étant le miroir de l'autre. Sa forme objet permet des modifications simples et rapides pour le Vulcan Engine. Sa forme-représentation est écrite en XML et permet un accès simplifié à la connaissance de l'architecture courante de l'application. La forme-représentation du modèle par extension courant n'est accessible qu'en lecture aux contraintes exprimées dans le modèle par intention. Seul le Vulcan Engine peut modifier le modèle par extension. Il s'agit d'un choix qui obéit en tout premier lieu à une nécessité de maintien de cohérence du modèle par extension. Ce choix obéit également à d'autres considérations telles que la simplicité d'usage ou les performances. Le Vulcan Engine utilise la base de données NoSQL Basex qui lui permet d'exécuter les contraintes de l'application. Ces contraintes sont exprimées en XQuery et sont regroupées dans le fichier XML décrivant un modèle par intention. Lors de l'écriture d'un modèle par intention, un utilisateur de Vulcan a accès à des bibliothèques de requêtes XQuery qui lui permettent une expression simple des contraintes de son application. Ces bibliothèques sont extensibles de sorte à permettre l'élargissement des contraintes possibles. Les révisions, ce mécanisme de résolution d'états problématiques, sont également des requêtes XQuery et bénéficient aussi d'une bibliothèque extensible. Vulcan s'inscrit donc comme une canevas de planification d'élasticité automatisée. Ce canevas opère un contrôle de cohérence architecturale. Vulcan est auto-élastique, simple d'utilisation et extensible. Le chapitre suivant propose une évaluation du prototype réalisé.

Chapitre 9

Évaluation qualitative

Sommaire

Un objectif des travaux de thèse est de valider l'approche par des cas d'usages permettant de montrer l'apport de la solution proposée. Ce chapitre présente donc des cas d'usages significatifs et va montrer comment Vulcan se différencie par rapport à d'autres solutions de gestion de l'élasticité. Pour cela, une première section présente les applications recensées et explique leur intérêt pour l'élasticité. Cela a pour but d'identifier des scénarios d'élasticité représentatifs et pourtant loin d'être tous adressés par les solutions actuelles de gestion de l'élasticité des applications dans le cloud.

9.1 Applications retenues pour la validation

Cette section présente la liste des applications retenues afin de valider l'approche proposée. Chacune de ces applications diffère des autres par ses besoins en termes de scénarios d'élasticité. Cette différence s'explique aussi bien par les dissimilitudes d'architectures

que par des logiques métiers hétéroclites. La première de ces applications est Springoo, un exemple déjà exposé au cours des explications de ce document.

9.1.1 Springoo

Springoo est une application web trois tiers JavaEE. Elle se compose de serveurs Apache, Jonas et MySQL. Cette application présente les caractéristiques suivantes :

- Springoo est une application représentative de 80% du Système d'information d'Orange.
- Par ses concepts elle est similaire à d'autres applications dans des domaines variés comme par exemple l'*Auto-Configuration Server (ACS)* de la société Motorola, une application permettant l'authentification et la configuration de matériels tels que des Livebox, des actionneurs de vannes, des capteurs de température ou d'autres périphériques. Un autre exemple est CoopNet, une application permettant de créer des ponts téléphoniques, de partager des documents et le bureau des utilisateurs en cours de conférence. Cette application est proposée par Orange Business Services.
- Springoo fait également partie des cas d'usages du projet FSN OpenCloudWare dans lequel s'inscrit cette thèse.
- Le champ des similitudes de Springoo peut même être étendu aux applications PHP reposant sur la stack LAMP en raison de leur architecture très similaire : seuls les serveurs du tiers métier changent.
- L'élasticité des applications multi-tiers est largement adressée par les solutions actuelles.

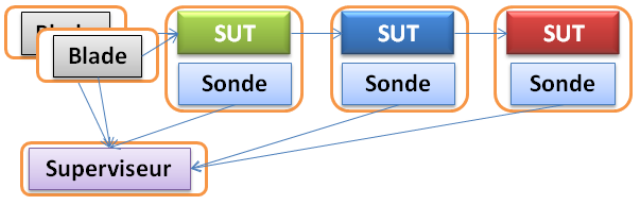
Toutes ces caractéristiques démontrent l'intérêt de Springoo : il s'agit d'une application très représentative et dont l'élasticité est déjà adressée par les solutions actuelles de gestion de l'élasticité. Springoo présente donc l'avantage d'être non seulement un cas d'usage pertinent mais également différenciateur dans la mesure où au final beaucoup de scénarios d'élasticité ne sont aujourd'hui pas adressés. C'est cette application qui a été mise en œuvre lors de l'intégration avec VAMP, une solution de déploiement d'applications dans le cloud qui implante l'Exécution dans la boucle MAPE-K.

9.1.2 Clif

Clif [51] est un canevas d'injection de charge permettant de stresser une application et d'en observer le comportement induit. Clif est une application Open Source écrite en Java. Elle fait partie intégrante des contributions du projet OpenCloudWare. Cette application est par ailleurs très utilisée pour tester des applications en cours de développement ainsi que pour effectuer des calibrages. Clif diffère totalement de Springoo par son architecture qui comprend les éléments suivants :

- Le *Superviseur* est l'unité centrale de Clif. Il orchestre les tests en modulant l'injection de charge puis rassemble les données produites par les sondes.
- Les *Sondes* sont des capteurs présents au niveau du *Système Sous Test (SUT)*. Elles permettent aussi bien de stocker des métriques généralistes (charge CPU, occupation mémoire) que des métriques plus spécifiques aux applications (nombre de requêtes par seconde, temps de réponse). Il s'agit donc d'un élément embarqué dans les machines de l'application soumises à la charge.
- Les *Blades* sont des éléments d'injection de charge qui simulent le comportement d'utilisateurs humains. Chaque blade peut moduler son injection. Le placement des blades par rapport au système sous test est primordial pour garantir la bonne tenue des tests. Il est effectivement nécessaire qu'elles soient placées dans le même réseau que le SUT afin que les flux réseaux ne constituent pas un goulet d'étranglement.

Les blades sont pilotées par le superviseur. Chacune d'elles présente donc une liaison en direction de celui-ci et les sondes suivent le même schéma. La figure 9.1 donne un exemple d'architecture de Clif.



Dans ce schéma, les rectangles de couleurs représentent des composants. Les VMs sont représentées par des rectangles oranges avec des bords arrondis. Ce type de représentation est utilisé dans l'ensemble des schémas d'architectures du reste du chapitre.

FIGURE 9.1 – Architecture de l'application Clif

Clif a été retenu en raison de l'importance du calibrage d'une application avant sa mise en production. Cela permet effectivement à une brique d'Analyse de savoir comment une application doit être mise à l'échelle. Son utilisation dans les milieux industriel et académique [148] en fait un cas d'usage intéressant puisque répandu.

9.1.3 Ganglia

Ganglia [75, 102, 104, 161] permet la surveillance de machines et d'applications. Il s'agit d'un canevas Open Source écrit en partie en C, en Perl et en PHP. Il est également capable de gérer des éléments en Python. A l'instar de Nagios [80] ou Zabbix, d'autres canevas de surveilance, Ganglia est très utilisé, que ce soit au niveau industriel ou académique (Grid5000). Il se compose de deux éléments principaux :

- Le *Ganglia Monitor Deamon (Gmond)* est un élément collecteur. Il permet de remonter les métriques de la machine sur laquelle il s'exécute. Il surveille égale-

ment les autres Gmonds du cluster. Les données des Gmonds sont envoyées au GmetadUI.

- Le *GmetadUI* comprend un *Ganglia Meta Deamon (Gmetad)*, un élément en charge de remonter et stocker dans une base de données locale les informations des Gmonds. Le GmetadUI comprend également une interface graphique PHP qui doit être colocalisée avec la base de données stockant les métriques.

L'architecture de Ganglia est en étoile : chaque Gmond a un liaison en direction du GmetadUI comme illustré par la figure 9.2.

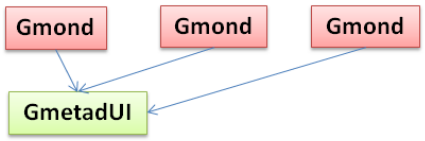


FIGURE 9.2 – Architecture applicative de ganglia

Ganglia a été retenu comme cas d'usage en raison de son utilisation répandue à des fins de supervision. Il s'agit d'un canevas pouvant par exemple être proposé comme un service de collecte de métriques par une plateforme de cloud. Ganglia peut également être vu comme une brique de Monitoring au sein d'une boucle autonomique MAPE-K visant à créer une élasticité automatique.

9.1.4 USB over IP

L'application USB over IP est un cas d'usage d'origine interne à Orange. Cette application permet de connecter des périphériques USB sur une Livebox sans avoir à charger des pilotes sur celle-ci. Les Livebox ont en effet des capacités matérielles limitées. D'autre part, inclure des pilotes à la volée pose d'importants problèmes de sécurité et de stabilité. L'idée est donc de déporter dans le cloud le traitement des périphériques USB. L'application se compose de deux types d'éléments :

- Les *Nœuds* sont des contrôleurs USB. De par une limitation du bus USB, ils ne peuvent gérer que 128 périphériques de façon concomitante bien que ces contrôleurs ne soient pas exigeants en termes de ressources matérielles. Une VM n'est donc jamais surchargée par un seul *Node*.

- Le *Répartiteur* est une unité en charge d'orienter les périphériques USB nouvellement connectés vers un nœud de traitement. Il s'agit d'une sorte de répartiteur de charge. Après cette première connexion, plus aucun flux ne passe par lui.

L'architecture d'USB over IP (USBoIP) comporte donc des nœuds tous liés en direction du répartiteur. La figure 9.3 en illustre un exemple.

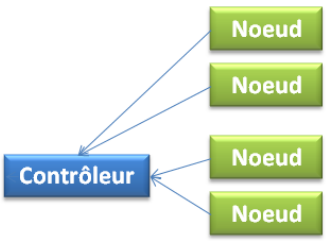


FIGURE 9.3 – Architecture de l'application USBoIP

L'intérêt d'USBoIP réside dans le fait que les scénarios optimaux d'élasticité pour cette application impliquent d'avoir une granularité plus fine que celle de la VM.

Cette section a présenté les cas d'usages retenus pour la validation des travaux de thèse. Ceux-ci comprennent des applications ayant des logiques métiers très différentes et des architectures tout aussi dissemblables. La section suivante présente des scénarios d'élasticité identifiés pour ces applications.

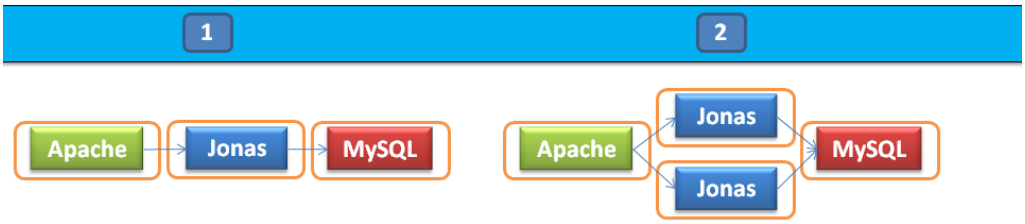
9.2 Scénarios d'élasticité

Un scénario d'élasticité est une succession d'architectures pour une application. Plus concrètement, durant son élasticité, une application va voir son architecture être modifiée en fonction de contraintes généralement en lien avec ses caractéristiques. Cette section présente un ensemble de scénarios d'élasticité pour les applications retenues comme cas d'usage et présentées ci-avant. Ces scénarios décrivent des évolutions architecturales pour chacune de ces applications. Ils visent à montrer des aptitudes de gestion de l'élasticité permettant de clairement différencier Vulcan par rapport à l'état de l'art actuel.

9.2.1 Scénario n˚1 : Elasticité multi-tiers élémentaire

Ce scénario est basique mais indispensable puisqu'il correspond à l'élasticité adressée par les solutions actuelles. Il s'agit d'une opération de croissance horizontale résultant en l'ajout d'un serveur sur le tiers métier. Appliqué à Springoo, ce scénario est réalisé au travers de l'ajout d'un nouveau serveur Jonas comme illustré par la figure 9.4. Cela permet pour l'application de pouvoir améliorer l'expérience utilisateur lorsque les serveurs existants commencent à être surchargés.

Ce scénario se justifie par sa représentativité tant au niveau des applications visant à être élastifiées, qu'au niveau de l'état de l'art actuel en matière de gestion de l'élasticité. Il s'agit d'un point de repère pour l'évaluation de Vulcan. Les scénarios qui suivent sont plus poussés.



Dans ce schéma, les rectangles de couleurs représentent des composants applicatifs, tandis que les rectangles oranges avec des bords arrondis correspondent aux VMs. Cette représentation est utilisée dans le reste des schémas de ce chapitre.

FIGURE 9.4 – Scénario d'élasticité n°1 : représentation de deux modèles par extension successifs.

9.2.2 Scénario n°2 : Elasticité à granularité composant

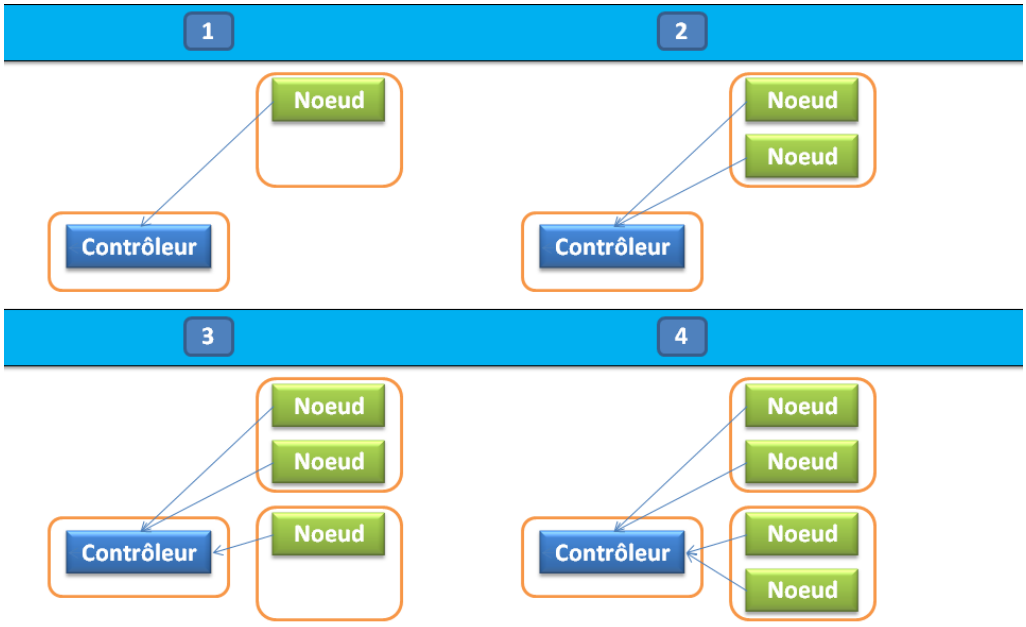
Ce scénario correspond de façon pratique à l'application USBoIP. L'idée est de permettre une élasticité qui ne fonctionne non pas par ajouts ou retraits de VMs mais plutôt par des ajouts ou des retraits de composants au sein des VMs. L'élasticité n'est donc plus à la granularité VM mais bel et bien à celle du composant. Dans le cas d'USBoIP, ce scénario vise à maximiser l'utilisation des ressources des VMs en regroupant plusieurs nœuds par VM. Il vise dans le même temps à minimiser le prix d'exécution de l'application. Cette optimisation existe certes déjà en partie avec une élasticité à granularité VM, mais cette optimisation peut être poussée plus loin grâce à ce scénario.

Toutefois, une VM a malgré tout des limites quant à ses capacités d'accueil en termes de nombres de nœuds placés. Il convient donc de respecter une cardinalité maximale. Le schéma 9.5 montre une succession de différents modèles par extension durant l'élasticité de l'application USBoIp.

9.2.3 Scénario n°3 : Elasticité verticale

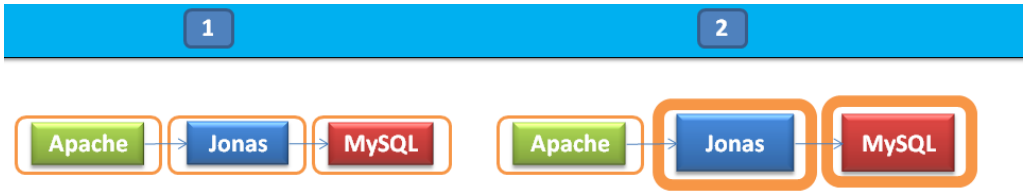
L'élasticité verticale est en quelque sorte le "parent pauvre" de l'élasticité puisque peu adressée actuellement. Il s'agit pourtant d'une possibilité formidable lorsqu'une forte rapidité d'exécution est requise. Comme le décrit la littérature scientifique, les pics de charge sont pour certains non-prévisibles : il est donc vital de pouvoir réagir rapidement lorsque l'un de ces pics se produit. Malgré le peu d'engouement suscité jusqu'à présent, ce scénario est aussi élémentaire que primordial.

Bien que cantonnée il y a peu aux seuls hyperviseurs dont Xen, ce type d'élasticité est dorénavant inclus dans les IaaS puisqu'OpenStack supporte ce type d'élasticité depuis la version Havana sortie le 17 Octobre 2013. Dans le cas de l'application Springoo, il peut par exemple être intéressant d'opérer une croissance verticale sur le tiers métier et/ou le tiers persistance, le temps d'un pic de charge comme illutré par la figure 9.6.



Ce schéma reprend la représentation graphique décrite pour la figure 9.4. Dans ce schéma, les VMs ne peuvent accueillir plus de deux nœuds. Il est donc nécessaire d'en rajouter comme durant la transition de l'état 2 à l'état 3.

FIGURE 9.5 – Scénario d'élasticité n°2 : représentation de quatre modèles par extension successifs.



Les serveurs Jonas et MySQL bénéficient ici de l'augmentation des capacités matérielles de leurs VMs respectives.

FIGURE 9.6 – Scénario d'élasticité n°3 : représentation de deux modèles par extension successifs.

9.2.4 Scénario n°4 : Elasticité profilée

L'élasticité profilée est un scénario d'origine interne à Orange. Il vise à moduler l'élasticité en fonction d'un profil. Il existe évidemment différents types de profils n'ayant pas les mêmes objectifs. Par exemple, un profil "Développement" a pour but de vérifier les fonctionnalités techniques et métier de l'application sans pour autant avoir besoin de bonnes performances. Ce profil est amené à réitérer souvent des opérations de dé-

ploiement/suppression d'une même application et celle-ci peut donc être déployé au sein d'une seule et même VM.

En revanche, un profil "Production" vise non plus à vérifier le fonctionnel de l'application, mais a besoin de bonnes performances. Le profil "Production" requiert donc un placement distribué sur plusieurs VMs. Par ailleurs, en fonction du profil, le déploiement peut être autorisé au sein d'une infrastructure mais pas d'une autre. Ce scénario peut être complété par deux sous-scénarios.

n°4 bis : Elasticité profilée et injection d'outils de plateforme de cloud

Ce scénario se présente comme étant un raffinement du scénario n°4. Il vise à injecter automatiquement des services dans une application. Les services de ce sous-scénario concernent ici l'injection de charge et la surveillance.

Un profil "Production" peut ainsi bénéficier d'une solution de surveillance telle que Ganglia afin de notamment veiller à la bonne santé de l'application. Un tel canevas permet également de surveiller l'évolution de certaines métriques ce qui permet d'envisager des améliorations à développer et/ou de réaliser une élasticité autonomique.

Un nouveau profil appelé "Test" permet de réaliser une injection de charge dans l'optique par exemple de calibrer l'application. Ce profil requiert une application ayant des contraintes de performances similaires à celles pouvant être obtenues en environnement de production. Dans ce sous-scénario, un canevas d'injection comme Clif est automatiquement ajouté à la plateforme sans que l'utilisateur n'ait à le spécifier.

En fonction du profil spécifié, des services peuvent ainsi être injectés automatiquement, ce qui permet :

- Une simplicité d'utilisation de la plateforme : chaque utilisateur déploie son application en spécifiant son profil et peut ainsi bénéficier de services sous-jacents offerts par la plateforme sans qu'il n'ait à se préoccuper de leur paramétrage et de leur élasticité.

- La fourniture d'une offre comportant une vaste gamme d'outils pour le fournisseur de plateforme de sorte à pouvoir se différencier face à la concurrence en permettant d'améliorer l'expérience utilisateur.

Ce sous-scénario s'appuie sur des considérations pragmatiques de séparations des préoccupations entre celles de l'utilisateur d'une plateforme de cloud et celles de son fournisseur. Si les solutions actuelles de gestion de l'élasticité permettent l'injection d'un service de surveillance, elles ne permettent pas d'injecter un canevas d'injection de charge. D'autre part, ces solutions de surveillance peuvent être propriétaires et leur utilisation payante.

Ce sous-scénario correspond à des cas d'usages internes à Orange mais également externes, notamment dans le projet OpenCloudWare.

Une première difficulté de sous-scénario consiste en la gestion des différents canevas injectés puisque ceux-ci ne doivent pas interférer les uns avec les autres sauf à le vouloir expressément. De base, deux profils "Production" doivent avoir des déploiements

de canevas de surveillance totalement indépendants. Il en est de même pour plusieurs déploiements avec le profil "Test" pour l'injection de charge. Il serait a priori mal venu que deux déploiement partagent le même Superviseur Clif ou la même interface de surveillance !

Une seconde difficulté réside dans le placement des sondes : celles-ci doivent être placées au sein des VMs de l'application. Leur propre placement dépend donc de celui des composants de l'application.

Scénario n°4 ter : Elasticité profilée, avec injection de services et maîtrise des placements & configurations

Ce second sous-scénario constitue un raffinement supplémentaire pour le scénario n°4. Il vise à aborder la problématique de la maîtrise de l'emplacement de destination de l'application mais également celle de la détermination les profils matériels des VMs. Concernant ce second point, il ne s'agit pas de remplacer une brique d'Analyse mais plutôt d'en compléter les demandes. Ce sous-scénario inclut également la détermination des VMs nécessitant une adresse IP publique puisque, de base, les IaaS ne fournissent que des adresses IP privées sauf demande explicite.

Une réalisation de ce sous-scénario concerne le profil "développement". L'application est uniquement placée dans un IaaS séparé de l'environnement de production, mais également au sein d'une VM dont le profil matériel doit malgré tout obéir à certaines contraintes.

Concernant les profils "Production" et "Test", ceux-ci requièrent que l'application soit placée en environnement de production, ce qui correspond à un placement dans un IaaS dédié avec des profils de VMs pouvant changer en fonction des composants. D'autre part, dans le cas d'une injection de charge, les injecteurs (e.g. les Blades de Clif) doivent être placés au sein du même IaaS que celui retenu pour l'application. Un profil de VM adapté doit également être déterminé.

Une première difficulté de ce sous-scénario réside une nouvelle fois dans la maîtrise des placements : les injecteurs de Clif doivent notamment être placés dans le même IaaS que l'application-cible. De même que dans le précédent sous-scénario, le placement des VMs des blades est induit a posteriori par celui des VMs de l'application.

Une seconde difficulté résulte des profils matériels des VMs : ceux-ci sont effectivement très différents entre les différents IaaS. La détermination du profil matériel de chaque VM nécessite donc de connaitre au préalable son IaaS de destination.

Cette section a présenté quatre scénarios d'élasticité issus des besoins des applications recensées comme cas d'usage pour la validation de Vulcan. Cette liste de scénarios n'a pas vocation à être exhaustive. Il s'agit là de scénarios choisis, toujours dans une optique de validation à partir de cas d'usages concrets. Dans la section suivante, ces scénarios permettent de situer Vulcan par rapport aux solutions actuelles de gestion de l'élasticité.

9.3 Positionnement par rapport à l'état de l'art

La section précédente a donné une liste de quatre scénarios correspondant aux besoins d'élasticité des applications retenues pour la validation de Vulcan. La gestion de ces différents scénarios permet de situer Vulcan par rapport aux solutions existantes, comme le montre le tableau 9.1 Le cas des solutions ADT et Elaas est un peu à part puisque ces canevas permettent d'implanter des logiques de gestion de l'élasticité. La mention "partiel" indique donc qu'à notre connaissance la solution ne gère pas ce scénario mais également que moyennant un coût de développement, un utilisateur expert peut parvenir à en permettre la logique.

Scénarios	n°1	n°2	n°3	n°4	n°4bis	n°4ter
Amazon AWS	**Oui**	Partiel	Non	Non	Non	Non
Microsoft Azure	**Oui**	Partiel	Non	Non	Non	Non
Cloud Foundry	**Oui**	Non	**Oui**	Non	Non	Non
Heroku	**Oui**	Non	**Oui**	Non	Non	Non
Jelastic	**Oui**	Non	**Oui**	Non	Non	Non
Env. Docker	**Oui**	Non	**Oui**	Non	Non	Non
Redhat OpenShift	**Oui**	Non	**Oui**	Non	Non	Non
ADT	**Oui**	Partiel	Partiel	Partiel	Partiel	Partiel
ElaaS	**Oui**	Partiel	Partiel	Partiel	Partiel	Partiel
soCloud	**Oui**	Partiel	**Oui**	Non	Non	Non
Vulcan	**Oui**	**Oui**	**Oui**	**Oui**	**Oui**	**Oui**

TABLE 9.1 – Gestion des scénarios d'élasticité présentés par différentes solutions d'élasticité

Vulcan est à notre connaissance la seule solution de gestion de l'élasticité qui soit capable de gérer l'intégralité des scénarios énoncés. Il faut toutefois bien admettre que ceux-ci n'ont pas été choisis innocemment puisqu'ils ont clairement pour but de différencier Vulcan par rapport à des solutions concurrentes. Néanmoins, ces scénarios sont clairement réalistes et représentatifs.

Conclusion

Ce chapitre a présenté certains des cas d'usage d'élasticité identifiés dans le cadre des travaux de thèse. Ceux-ci comprennent des applications et leurs scénarios d'élasticité : ce chapitre a permis d'expliquer la pertinence de leur choix. Cette pertinence qui se base notamment sur un critère de représentativité des besoins a été avérée durant ce chapitre.

L'identification de ces cas d'usage a permis de fournir une évaluation qualitative qui identifie clairement Vulcan comme une solution qui apporte une vraie avancée en matière de couverture de l'élasticité. Vulcan permet donc d'adresser des cas d'usage à la fois plus

complexes mais également non adressés actuellement malgré un besoin réel. Le chapitre qui suit donne une évaluation des performances de Vulcan sur les scénarios recensés.

Chapitre 10

Performances

Sommaire

Après que le chapitre précédent ait fourni une évaluation qualitative montrant une partie des capacités de Vulcan en matière de couverture de scénarios d'élasticité, ce chapitre propose une évaluation des performances. Ces mesures de performances visent tout d'abord à estimer la rapidité de calcul de Vulcan. Il s'agit de montrer que le prototype offre des performances suffisantes mais surtout réalistes : Vulcan ne doit pas introduire un overhead trop grand lors des traitements de l'élasticité. Ces mesures doivent également illustrer la couverture des scénarios d'élasticité évoqués dans le chapitre précédent. Enfin, les mesures effectuées visent à offrir un aperçu de la simplicité à utiliser Vulcan.

Pour la réalisation des mesures de performances, un protocole d'expérimentations a été mis en place. La sous-section suivante décrit le protocole de tests retenu.

10.1 Protocole de tests

Dans l'ensemble, les tests présentés ont consisté en des calculs de planification sans répercussion sur une application réelle au travers d'une brique de déploiement. Nous avons effectivement estimé que cela ne présente pas d'intérêt scientifique de montrer

systématiquement une répercussion complète dans la mesure où les temps de répercussion font l'objet d'une forte variabilité liée notamment au réseau et à l'éloignement des VMs. Cela aurait pour effet de diluer la vraie problématique sans ajouter d'informations utiles. Dans un souci de clarté pour le lecteur et de concision, seul un test avec une répercussion est présenté ici. Celui-ci permet de valider la capacité d'intégration de Vulcan dans la boucle MAPE-K tout en donnant un référentiel de temps pour les autres tests qui eux portent sur l'étude de la complexité de la planification. La complexité de planification réside avant tout dans son expressivité et dans la qualification de sa mise en œuvre. Cette qualification renvoie plus particulièrement à la capacité de gestion de multiples scénarios d'élasticité à l'aide de contraintes. Elle renvoie également à l'appréciation de l'évolution des performances face à différents facteurs tels que le nombre d'entités à gérer ou la multiplication des contraintes. En plus du test portant sur l'intégration, des tests mesurant la durée nécessaire aux calculs de la planification ont été menés. Pour valider la capacité de Vulcan à effectivement gérer les scénarios du chapitre précédent, ces tests ont donc consisté en la mise en œuvre de ces scénarios. Afin d'éliminer les erreurs statistiques, chaque test a été reproduit 200 fois. Pour chacun, les métriques relevées permettent de mesurer le temps pris par chaque phase de l'algorithme de planification proposé ce qui permet de montrer des effets intéressants.

Ces tests ont été menés dans l'optique d'être aussi réalistes que possible. Nous avons effectivement estimé que fournir des tests sur une plateforme matérielle haut-de-gamme n'est pas plus réaliste que de le faire sur des processeurs anciens. Dans la mesure du possible, ces tests ont donc été réalisés afin de coller à l'actualité matérielle actuelle tout en fixant un contexte de performances n'avantageant clairement pas les résultats.

La plateforme retenue est donc une configuration d'entrée de gamme pour l'année 2014. Celle-ci se compose d'un processeur Intel Pentium G1840 (double-cœur à 2,8GHz **sans *Turbo-Boost***), de 2Go de ram (DDR3 PC10600 Cas11, simple canal), d'un disque dur de 500Go (SATA2, 7200 tours/min) le tout sur une carte mère MSI H97M-G43 (chipset Intel H97). Cette plateforme fonctionne sous une Ubuntu Desktop 13.10 avec Java en version 1.7. Afin de faciliter la reproductibilité des tests, l'*Intel SpeedStep* a été désactivé (variation de la fréquence en fonction de l'occupation CPU). Il s'agit donc d'une plateforme minimaliste mais honnête puisqu'elle n'a ni le défaut d'embellir les résultats ni celui de les réduire de façon inconsidérée.

Pour les tests portant sur l'intégration VAMP-Vulcan, la plateforme de tests est un IaaS OpenStack interne à Orange, s'exécutant sur des nœuds comprenant deux processeurs Intel Xeon E5450 (Core2Quad 3,0GHz).

La suite de ce chapitre présente les résultats obtenus lors des tests mentionnés. Chaque section traitera d'un test et donnera quelques détails supplémentaires afin d'offrir une vision claire et complète du scénario et des modèles par intention utilisés.

10.2 Test n°1 : Intégration VAMP-Vulcan

Ce premier test est un scénario basique concernant l'application Springoo, le scénario n°1 de la liste donnée ci-avant. Celui-ci est constitué de deux étapes successives comme le montre la figure 10.1. La première étape consiste en le déploiement initial de Springoo composée d'un serveur Apache, d'un serveur Jonas et d'un serveur MySQL chacun ayant sa propre VM. A la fin de cette première étape, l'application Springoo est déployée et fonctionnelle. Une fois ce déploiement initial effectué, le test continue sur une seconde étape qui vise à opérer une opération d'élasticité. Il s'agit d'une opération de croissance horizontale sur le tiers métier qui résulte en l'ajout d'un nouveau serveur Jonas. L'application est alors modifiée pour inclure ce nouveau serveur.

De façon plus précise, ce test met en avant l'intégration de Vulcan avec VAMP. La plateforme de test est composée d'un Vulcan Engine avec une interface VulcanREST, d'un VAMP et du IaaS OpenStack interne à Orange. En début de première étape, le modèle par intention est renseigné auprès de Vulcan qui effectue son calcul de planification puis renvoie le modèle par extension correspondant au déploiement initial. Un connecteur traduit ensuite ce modèle par extension en un fichier OVF étendu, le formalisme de VAMP. VAMP procède alors au déploiement initial.

L'opération d'élasticité de la seconde phase suit un cheminement différent. Celle-ci répond effectivement à une demande de croissance horizontale provenant de la brique de décision implantée pour le test (i.e. une pseudo-brique d'Analyse). En suivant la logique de la boucle MAPE-K, cette décision est fournie à Vulcan. Vulcan complète alors le modèle par extension qui représente l'état actuel de l'application. Le nouveau modèle par extension obtenu est ensuite traduit en OVF étendu puis fourni à VAMP qui répercute les changements nécessaires. Au final, l'application Springoo se voit affectée d'un nouveau serveur Jonas placé dans sa propre VM.

Le modèle par intention utilisé pour cette expérimentation est donné par le listing 8.9 tandis que la figure 10.2 montre les mesures du temps nécessaire à la réalisation des différentes opérations.

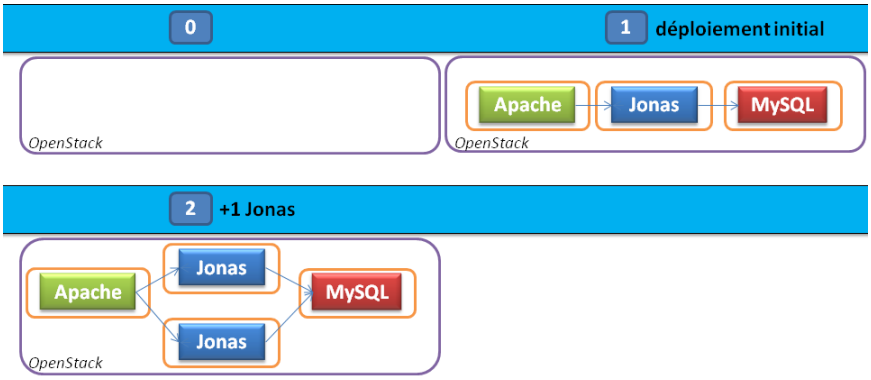


FIGURE 10.1 – Détail du scénario d'élasticité réalisé par l'intégration de VAMP et Vulcan.

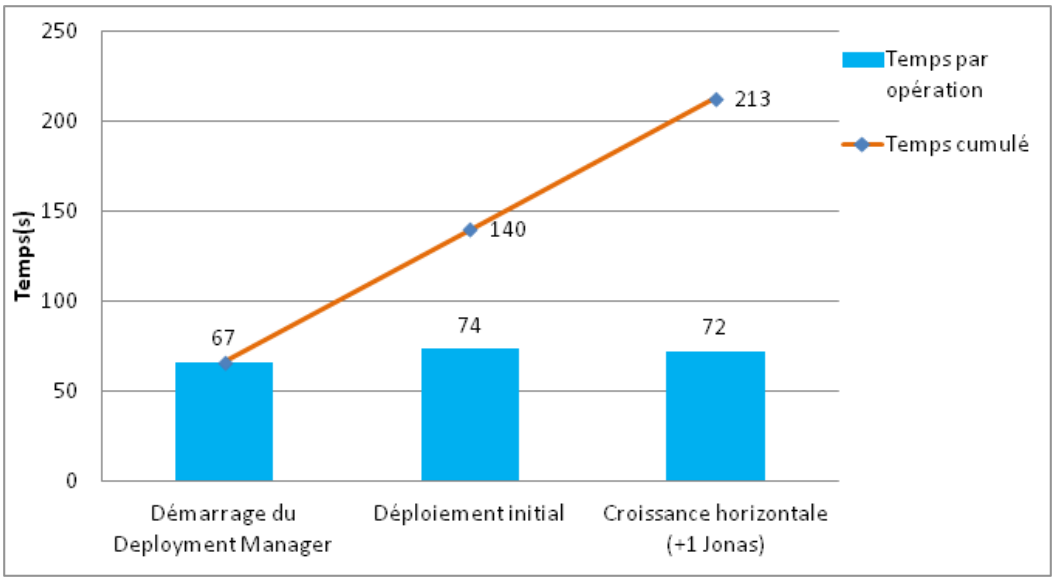


FIGURE 10.2 – Performances de l'intégration de Vulcan avec VAMP.

Le diagramme de la figure 10.2 présente trois mesures de temps. La première correspond au démarrage d'une instance de *Deployment Manager*, un gestionnaire de déploiement de VAMP qui s'exécute dans sa propre VM et qui a la charge d'une application. Il s'agit d'une mécanique interne à VAMP qui n'est donc pas géré par Vulcan. Le temps de démarrage mentionné dans la figure 10.2 inclue le boot de la VM et la configuration du Deployment Manager. Les deux autres mesures correspondent au temps nécessaire pour le déploiement des étapes 1) et 2) de la figure 9.4. Le déploiement initial ne prend ici qu'uniquement deux secondes de plus que l'opération de croissance horizontale en raison de la parallélisation de VAMP qui permet le déploiement de l'ensemble des trois VMS de l'application initiale en même temps. En comparaison, le temps nécessaire à Vulcan pour effectué ses calculs se chiffre à moins de 100ms.

Cette figure montre un temps d'approvisionnement et de configuration des VMs qui est de l'ordre de la minute. Cela permettra de discuter les performances de Vulcan dans la suite de ce document, sur des scénarios plus complexes.

10.3 Test n°2 : USBoIP

Ce test correspond au scénario d'élasticité n°2 : l'élasticité à la granularité composant. Le cas d'usage est USBoIp, une application nécessitant de placer plusieurs composants au sein d'une même VM tout en ne dépassant pas un nombre maximal. L'expérimentation a ici consisté à mesurer le temps nécessaire à la planification lors d'ajouts successifs de nœuds. La limite du nombre de nœuds par VM a été fixée dans cette expérimentation à quatre. Le modèle par extension initial est minimal puisqu'il ne comporte qu'un contrôleur et un noeud. Le contrôleur est placé dans sa propre VM.

Le modèle par intention utilisé pour ce test est exposé dans le listing 10.1. Les performances mesurées sont quant à elles exposées dans le digramme 10.3.

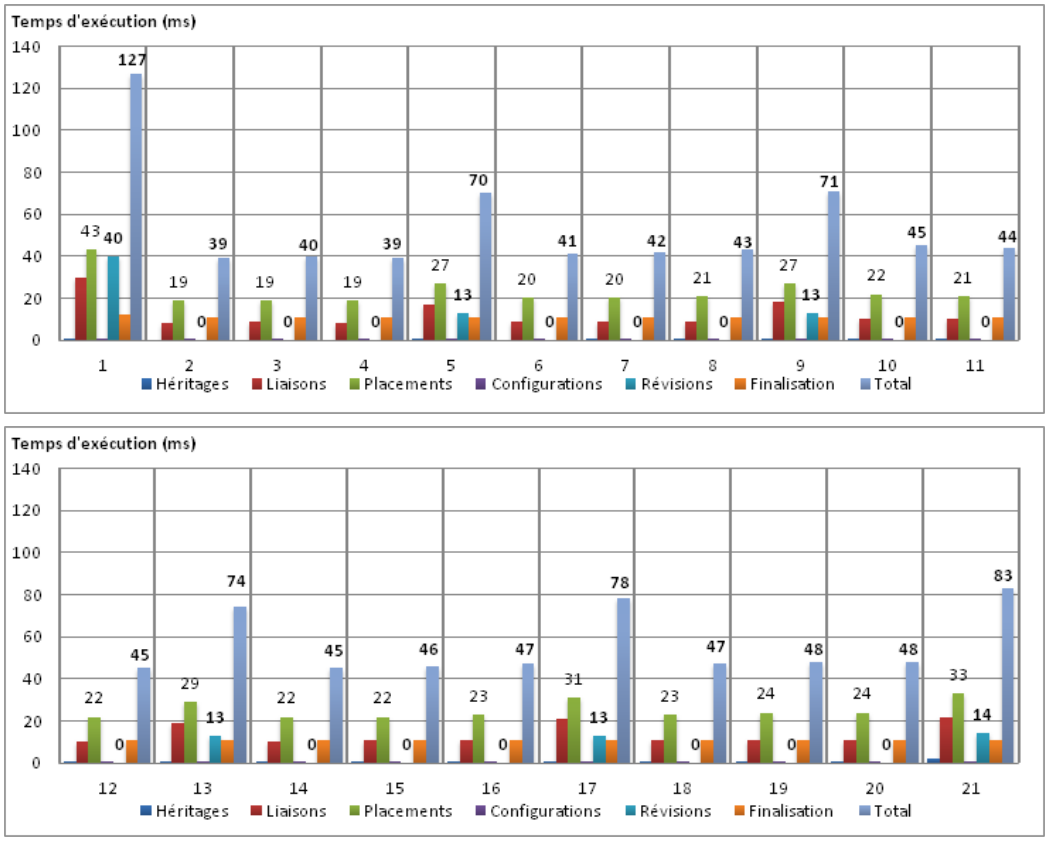


FIGURE 10.3 – Performances de Vulcan sur le Scénario n°2 pour l'application USBoIp : temps de calcul en millisecondes, en fonction du nombre final de nœuds

Listing 10.1 – Modèle par Intention pour l'application USBoIp

```
<intensional-model>
        <components>
                <component name="Controler" init="1"/>
                <!-- initialisation :
                un petit piege pour tester la contrainte de liaison! -->
                <component name="Node" init="0"/>
        </components>
        <containers>
                <container name="vm" init="0"/>
        </containers>

        <bindings>
                <binding-query>
                        local:bindAllExactlyOne("Node","Controler")
                </binding-query>
        </bindings>

        <placements>
                <placement-query>
                        local:placeManyInstancesPerContainer("Node", 4,
                            "vm")
                </placement-query>
                <!-- pour le placement du controleur -->
                <placement-query>
                        local:placeOneOne()
                </placement-query>
                <!-- suppression des VMs vides -->
                <placement-query>
                        local:purgeEmptyContainers("vm")
                </placement-query>
        </placements>

    <configurations>
                <configuration-query>
                        local:setPublicIpOn(("Controler"), "vm")
                </configuration-query>
    </configurations>

</intensional-model>
```

Le diagramme 10.3 montre les performances mesurées de Vulcan sur la machine de test en procédant à 20 ajouts successifs de 1 nœud. Sur ce diagramme, sont présentés les relevés moyens des temps de calculs exprimés en millisecondes et en fonction du nombre de nœuds obtenus après traitement de l'ajout.

Pour chaque relevé un ensemble de sept sous-mesures est exposé. Les cinq premières sous-mesures sont appelées *Héritages*, *Liaisons*, *Placements*, *Configurations* et *Révisions*. Elles indiquent la durée totale des calculs dans chacune des phases de l'algorithme de planification. La sous-mesure *Finalisation* indique quant à elle le temps mis pour finaliser

le modèle par extension, c'est-à-dire le temps nécessaire pour les traitements internes sur le modèle par extension en fin de déroulement d'algorithme. Enfin, la sous-mesure *Total* indique le temps total de calcul. Les mesures sont donc exposées avec une granularité relativement fine qui permet d'évaluer le comportement du Vulcan Engine durant les calculs. Dans les faits, ces mesures proviennent de l'agrégation de métriques plus fines qui si elles présentent un intérêt évident pour un cerveau-B, n'offrent en revanche pas une vision suffisamment concise pour ce manuscrit.

Le diagramme 10.3 montre que les relevés pour cette expérimentation sont compris entre 39 et 127 millisecondes. De façon plus précise, en dehors de l'initialisation qui est un peu à part (relevé n°1), les mesures ont une répartition bi-modale. Deux ensembles peuvent en effet être distingués : un premier dont les valeurs sont comprises entre 39 et 48 millisecondes et un second qui a des valeurs entre 70 et 83 millisecondes. Cette répartition s'explique par le fait que lors d'un ajout de nœud requérant une nouvelle VM comme illustré dans le cas de la figure 10.4, une révision est demandée en phase de Placements par la contrainte *local :placeManyInstancesPerContainer("Node", 4, "vm")*. Cette révision stipule qu'il manque un conteneur de type "vm" pour le nouveau nœud. Ce déroulement est bien visible sur le diagramme 10.3, au niveau des relevés n°5, 9, 13, 17 et 21 qui ont tous une sous-mesure pour les révisions qui a une valeur comprise entre 13 et 14ms, alors qu'elle vaut 0 pour les autres, exception faite de l'initalisation. Par ailleurs, comme une révision entraine une reprise de la séquence principale de l'algorithme depuis son début, les sous-mesures Héritages, Liaisons et Placements voient leur durée également augmenter. Le cas de l'initialisation est un peu à part dans ces relevés puisque comme l'illustre la figure 10.5, plusieurs révisions sont requises : une pour ajouter un premier nœud afin de satisfaire la contrainte de liaison, une autre pour l'ajout d'une VM pour le contrôleur, et une dernière pour l'ajout d'une VM pour le premier nœd, lui-même ajouté par une précédente révision. Le temps passé par le Vulcan Engine en traitement des Révisions passe ainsi à 40ms.

Dans ce scénario, le temps de calcul de Vulcan peut ainsi être considéré comme faible et même négligeable en comparaison du temps de démarrage et de configuration d'une VM dans un IaaS.

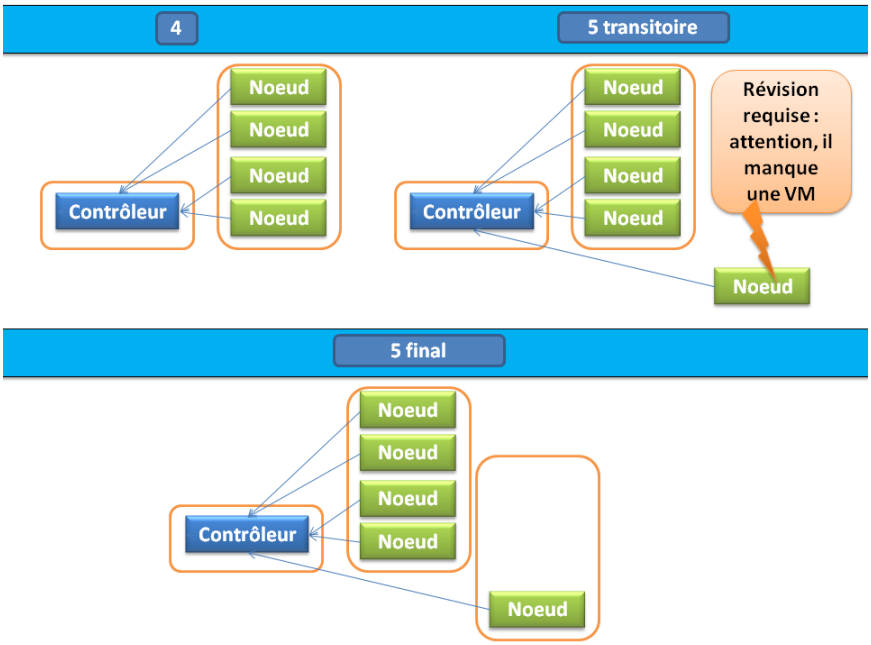


FIGURE 10.4 – Passage de l'état 4 à l'état 5 du test n°1 : une révision est requise

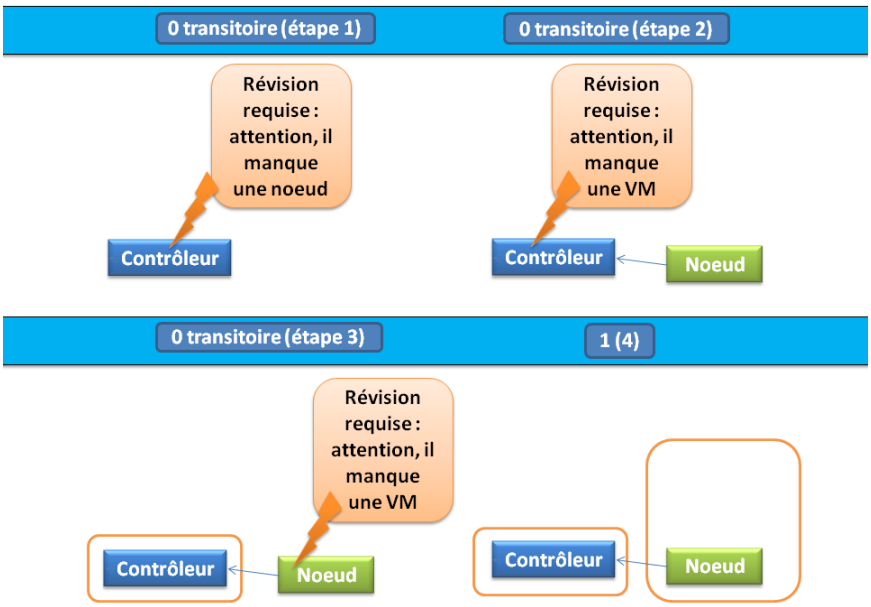


FIGURE 10.5 – Détail de l'initialisation du test n°1 et des révisions requises

10.4 Test n°3 : Elasticité verticale de l'application Springoo

Relatif au scénario n°3, ce test vise à montrer les capacités de Vulcan à gérer l'élasticité verticale. De façon plus précise, cela implique de :

1. Placer chaque composant dans une VM.
2. Placer chaque VM dans un IaaS.
3. Choisir le profil de VM adapté en fonction d'un profil matériel minimal et des profils proposés par le IaaS choisi.
4. Mettre à jour le profil de VM lorsque cela est requis comme c'est le cas lors d'une opération de (dé)croissance verticale.

Dans ce test, Vulcan doit donc résoudre plusieurs contraintes de placements et de configurations (pour le profil des VMs, en plus de celles déjà existantes pour les liaisons applicatives. La figure 10.6 montre les différentes étapes du test pour le modèle par intention exposé dans le listing 10.2.

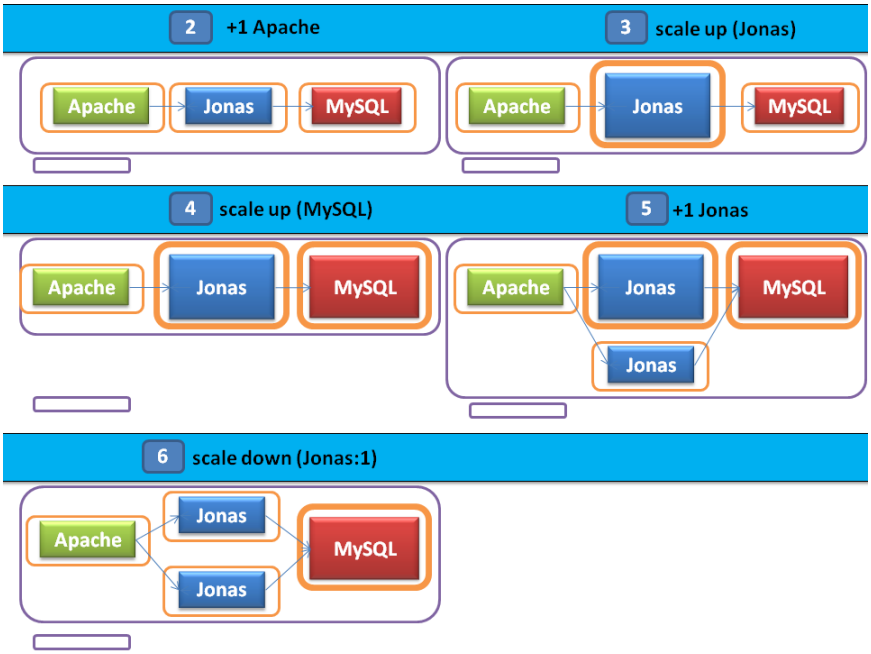


FIGURE 10.6 – Succession des différentes étapes du test n°3

Dans la figure 10.6, les IaaS sont représentés par des rectangles violets avec des coins arrondis. Dans ce test, deux Iaas sont utilisés afin d'illustrer un cas de choix de placement pour les VMs. Ces IaaS correspondent pour l'un au IaaS OpenStack précédemment utilisé pour l'intégration VAMP-Vulcan, et pour l'autre à un datacentre dernière génération d'Orange.

Listing 10.2 – Modèle par Intention du test n°3

```
1  <intensional-model>
2          <!-- les balises delimitant les requetes sont absentes pour
       faciliter la lecture -->
3          <components>
4                  <component name="Apache" init="0"/>
5                  <component name="Jonas" init="0"/>
6                  <component name="MySQL" init="0"/>
7          </components>
8          <containers>
9                  <container name="vm" init="0"/>
10                 <container name="iaas" init="0"/>
11         </containers>
12
13         <bindings>
14                 <!-- liaison Apache[1]-[1..*]Jonas -->
15                 local:bindExactlyOneAll("Apache","Jonas")
16                 <!-- liaison Jonas[1..*]-[1]MySQL -->
17                 local:bindAllExactlyOne("Jonas","MySQL")
18         </bindings>
19
20         <placements>
21                 <!-- placer chaque composant dans sa propre VM -->
22                 local:placeOneOne()
23                 <!-- aucune VM vide autorisee -->
24                 local:purgeEmptyContainers("vm")
25         <!-- placer chaque VM contenant un composant Apache, Jonas ou MySQL
26         dans un IaaS precis (reconnu par son identifiant logique) -->
27                 local:placeAllContainersContainingTypesOfInstances("vm",
                       "iaas",
28                 "id", "iaas_val-de-reuil",
29                  ("Apache","Jonas","MySQL"))
30         </placements>
31     <configurations>
32         <!-- profil minimal pour les VMs -->
33         local:setContainersProfilesByInstancesContained("vm",
34               ("Apache","Jonas","MySQL"),
35              <profile cpu="1" ram="4096" diskSize="20GB"/>)
36         <!-- mise a jour des profils de VMs : Elasticite Verticale -->
37         local:applyVerticalElasticity("vm", "iaas")
38         <!-- adresse ip publique sur tout composant Apache -->
39             local:setPublicIpOn(("Apache"),"vm")
40     </configurations>
41 </intensional-model>
```

Le lecteur pourra noter qu'aucun IaaS n'est stocké directement dans le modèle par intention : les IaaS peuvent ainsi être connus lors de l'exécution, de même que les profils matériels qu'ils proposent à leur clients. La prise en compte des IaaS est dynamique mais comme le montre cet exemple, il est possible de maîtriser le IaaS de destination, puisque la contrainte - écrite de la ligne 25 à la ligne 27 - renseigne un identifiant précis de IaaS

à utiliser : cela est lié à cette contrainte et résulte donc d'un choix utilisateur.

Le non-renseignement des IaaS dans le modèle par intention permet non seulement une connaissance dynamique des IaaS mais aussi de garantir une certaine facilité d'utilisation. La connaissance des IaaS disponibles est réalisée dans l'étape n°1 au travers d'une simple addition de conteneur de type IaaS par la même API que celle fournie à la brique d'Analyse. Pour chaque IaaS, les propriétés renseignées spécifient l'identifiant et les profils matériels disponibles. Le modèle par extension obtenu est visible dans le listing 10.3. Il mentionne les deux IaaS cités ci-avant.

Listing 10.3 – Modèle par Extension de l'étape n°1 du test n°3

```
<extensionalModel>
        <container type="iaas" name="2">
                <properties name="profiles">
                        <value>
<profile name="m1.tiny" ram="512" cpu="1" arch="x86" diskSize="4GB"/>
<profile name="m1.small" ram="1024" cpu="1" arch="x86" diskSize="10GB"/>
<profile name="m1.medium" ram="2048" cpu="2" arch="x86" diskSize="10GB"/>
                        </value>
                </properties>
                <properties name="id">
                        <value>openstack-dev</value>
                </properties>
        </container>
        <container type="iaas" name="1">
                <properties name="profiles">
                        <value>
<profile name="t2.micro" ram="1024" cpu="1" arch="x86" diskSize="0GB"/>
<profile name="t2.small" ram="2048" cpu="1" arch="x86" diskSize="0GB"/>
<profile name="t2.medium" ram="3840" cpu="1" arch="x86" diskSize="4GB"/>
<profile name="m3.large" ram="7680" cpu="2" arch="x86" diskSize="32GB"/>
<profile name="m3.xlarge" ram="15360" cpu="4" arch="x86" diskSize="80GB"/>
<profile name="m3.x2large" ram="30720" cpu="8" arch="x86"
    diskSize="160GB"/>
                        </value>
                </properties>
                <properties name="id">
                        <value>iaas_val-de-reuil</value>
                </properties>
        </container>
</extensionalModel>
```

Durant le test n°3, les profils des VMs sont donc choisis parmi ceux des IaaS renseignés. Le choix du IaaS est spécifié par contrainte dans le modèle par intention de même que la détermination du profil adapté pour les VMs. Au cours des tests, ce profil est modifié pour un serveur Jonas et le serveur MySQL comme illustré aux étapes n°3, 4 et 6 par le schéma 10.6. Les performances obtenues sont quant à elles visibles dans le diagramme 10.7. Celui-ci montre que les opérations d'élasticité se répartissent en trois ensembles :

1. Les étapes n°3, 4 et 6 ont des mesures et des répartitions très similaires. Celles-ci correspondent à des opérations de (dé)croissance verticale portant sur une VM. Les durées de la planification durant ces étapes se situent ici entre 133 et 153 millisecondes ce qui est rapide. L'étape n°1 est également similaire, mais son cas est un peu à part en raison de la nature de l'opération traitée.

2. Les étapes n°2 et 5 correspondent à des opérations d'élasticité horizontale avec pour la première l'ajout d'un composant Apache dans un modèle ne comprenant que des conteneurs de type iaas (ce qui entraine l'ajout d'un composant Jonas et d'un MySQL), et pour la seconde, l'ajout d'un composant Jonas au tiers métier. Ces deux étapes requièrent des révisions pour des ajouts de VMs et leurs durées mesurées sont respectivement de 199 et 230ms ce qui est également rapide.

3. L'étape n°1 qui est un peu à part en raison de sa nature.

Les relevés pour ce test n°3 montrent la rapidité d'exécution de Vulcan pour l'élasticité verticale. Il est à noter qu'un scénario non-montré dans ce test est adressé par Vulcan : **il s'agit d'une gestion concomitante de l'élasticité verticale ET horizontale**. Vulcan permet effectivement de réaliser concomitamment les deux types d'élasticité ce qui est un différenciateur supplémentaire.

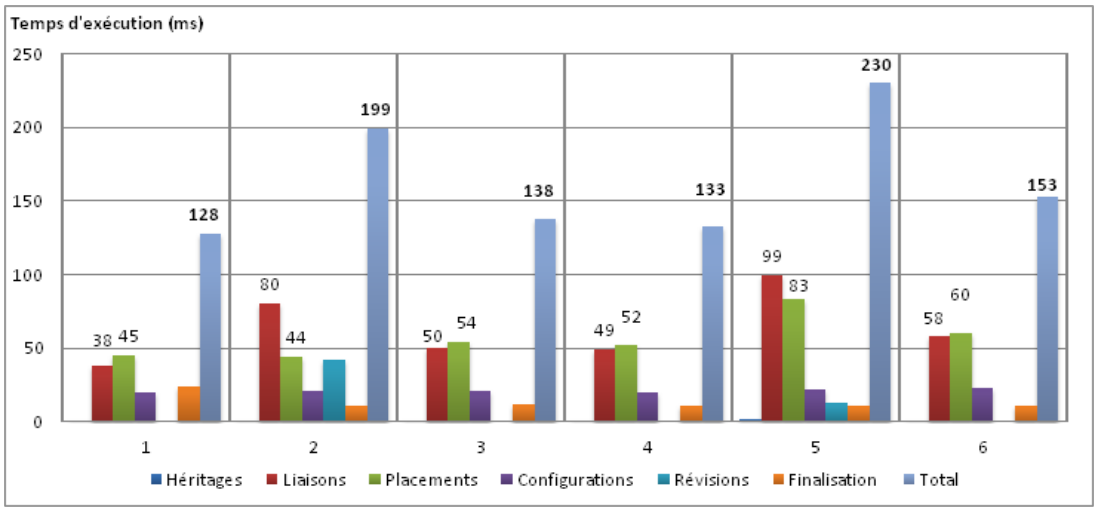


FIGURE 10.7 – Performances de Vulcan sur le Scénario n°3 pour l'application Springoo : le temps de calcul est donné en millisecondes, en fonction des étapes

Au cours du test n°3, Vulcan a démontré ses capacités de gestion de l'élasticité verticale. Plus que cela, il a démontré sa capacité à offrir une connaissance dynamique du cloud par une liste de IaaS fournie durant l'exécution. Grâce à cette connaissance à chaud Vulcan se définit comme une solution à la fois simple d'usage, dynamique et réaliste. Enfin, sa gestion concomitante des deux types d'élasticité est un avantage par rapport aux solutions existantes. Par ailleurs, il est à noter que des scénarios d'élasticité

à la granularité composant ont également concernés Springoo : à partir d'un déploiement initial sur deux VMs où un serveur Apache et un serveur Jonas étaient colocalisés sur la même VM, l'application était élastifiée en répartissant chacun des composants dans sa propre VM avant de revenir à l'état initial. Ce scénario a été mené à bien sur une instance réelle de Springoo grâce au Vulcan Deployer et sur un cluster d'hyperviseurs Virtualbox. Le test qui suit aborde une autre capacité différenciatrice de Vulcan : l'élasticité profilée.

10.5 Test n˚4 : Elasticité profilée

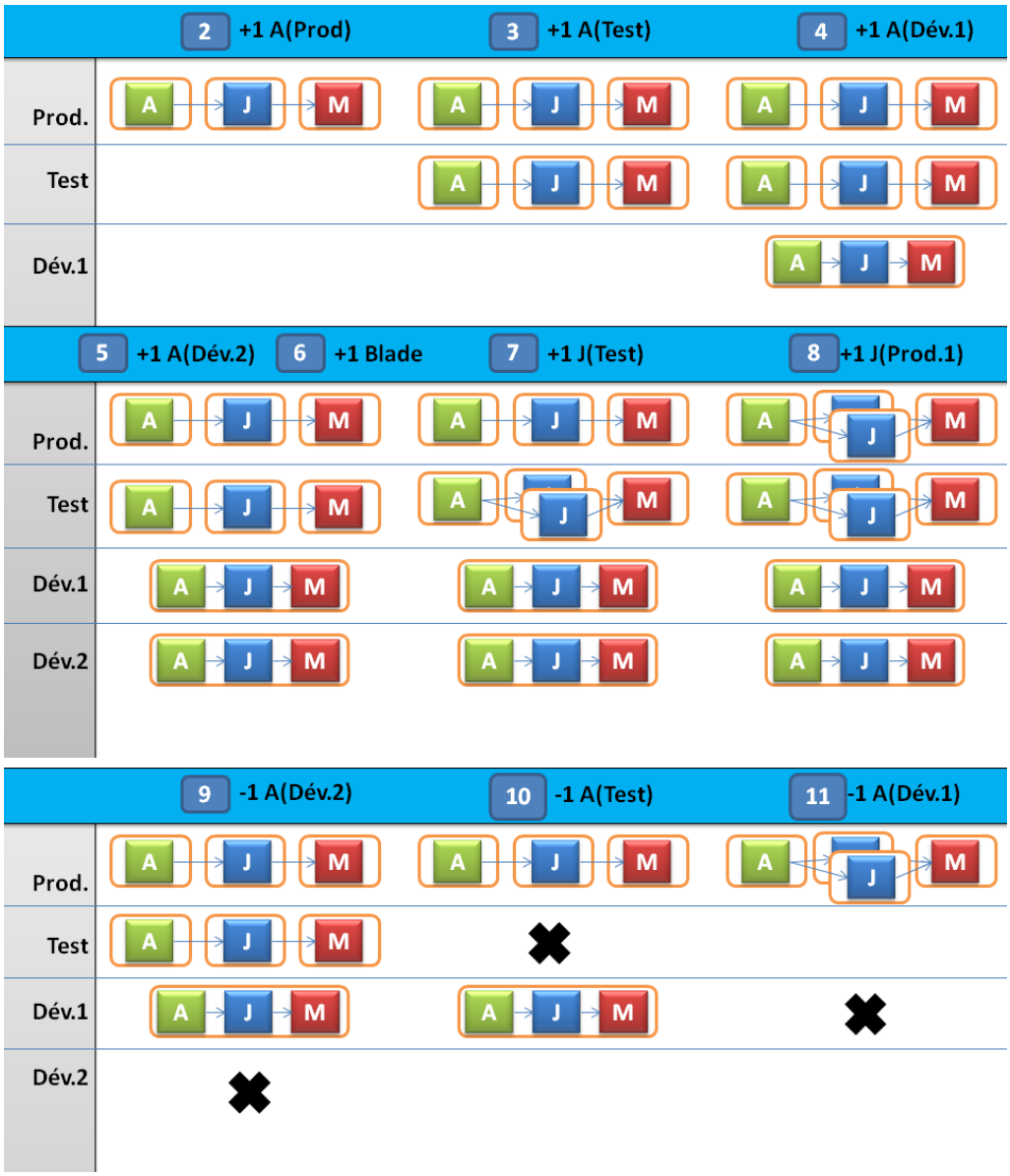
Ce quatrième test correspond au scénario n˚4. Ce scénario adresse le déploiement multiple de la même application avec des modalités qui évoluent au cours de son cycle de vie. Ce scénario est particulièrement pertinent face à l'émergence de l'approche DevOps [29, 78, 2] un mouvement qui vise à diminuer les incompréhensions et problèmes entre le développement (Dev) et l'exploitation (Ops). Il s'agit de favoriser la coopération entre des entités qui ont des objectifs opposés : stabilité pour l'exploitation, évolutions pour le développement. Le scénario n˚4 abonde dans le sens de ce mouvement en permettant le déploiement élastique multiple de la même application avec des sources pouvant varier.

Les différentes étapes suivies au cours du présent test sont illustrées par la figure 10.8. Comme le montre ce schéma, ce test a consisté en l'instanciation de l'application Springoo selon différents profils : *Production*, *Test* et *Développement*. Les profils *Production* et *Test* ne présentent ici aucune différence, il s'agit juste de mettre en place un contexte expérimental qui pourra être réutilisé pour les scénarios 4bis et 4ter afin de comparer les temps de calculs mesurés lorsque le nombre de contraintes augmente. Par ailleurs, les modèles par extension 5 et 6 sont identiques puisque si pour les scénarios 4bis et 4ter il s'agit d'ajouter une blade pour Clif, dans le cas du scénario n˚4 le composant blade n'est pas connu. Cela offre néanmoins l'avantage de vérifier le respect des contraintes d'héritage puisque ce *+1 Blade* ne débouche sur aucune modification pour le scénario n˚4 au contraire des scénarios 4bis et 4ter.

Le modèle par intention utilisé pour ce test est donné dans le listing 10.4. Celui-ci utilise des contraintes similaires à celles utilisées pour les précédents tests, mais avec une légère modification permettant de filtrer les profils. Pour améliorer la clarté, certaines balises ont été retirées.

Le graphique de la figure 10.9 présente les mesures de temps relevées pour les différentes opérations. Celles-ci se répartissent sur un intervalle compris entre 163 et 239ms pour les opérations entrainant une modification du modèle par extension. Les autres opérations sont l'initialisation (étape n˚1) et l'ajout d'un composant inconnu (étape n˚6). Concernant l'initialisation, celle-ci crée un modèle par extension vide puisque les champs *init* des composants et conteneurs du modèle par intention 10.4 ont une valeur nulle. Les durées observées sont donc ici rapides au regard du temps de démarrage d'une VM au sein d'un IaaS.

Avec les tests n˚1, 2, 3 et 4 Vulcan a montré son aptitude à gérer efficacement des scénarios standards et innovants. Toutefois, aucun d'eux n'a montré les limites de l'implantation actuelle du prototype. La suite de cette section vise donc à complexifier le scénario n˚4 avec les raffinements complémentaires des sous-scénarios 4bis et 4ter. L'objectif est d'étudier le comportement de l'implantation face à l'augmentation du nombre de contraintes complexes.



Pour chaque étape, l'opération d'élasticité à son origine est spécifiée à côté de son numéro. Les étapes 5 et 6 sont identiques puisque l'opération élastique demandée concerne un ajout de composant non reconnu : les contraintes d'héritages sont vérifiées car aucune modification n'est réalisée. Pour améliorer la lisibilité, le nom des composants est abrégé : A pour Apache, J pour Jonas et M pour MySQL

FIGURE 10.8 – Succession des différentes étapes du test n°4

Listing 10.4 – Modèle par Intention pour le scénario n°4

```
        <components>
                <component name="Apache" init="0"/>
                <component name="Jonas" init="0"/>
                <component name="MySQL" init="0"/>
        </components>
        <containers>
                <container name="vm" init="0"/>
        </containers>

        <bindings>
                local:bindExactlyOneAll_profile("Apache","Jonas","profile")
                local:bindAllExactlyOne_profile("Jonas","MySQL","profile")
        </bindings>

        <placements>
                local:placeAllInOneContainer_profile(
                        ("Apache","Jonas","Database"),
                        "vm", "group","dev")
                local:placeOneOne()
         local:purgeEmptyContainers("vm")
        </placements>
    <configurations>
                local:setPublicIpOn(("Controler"), "vm")
    </configurations>
```

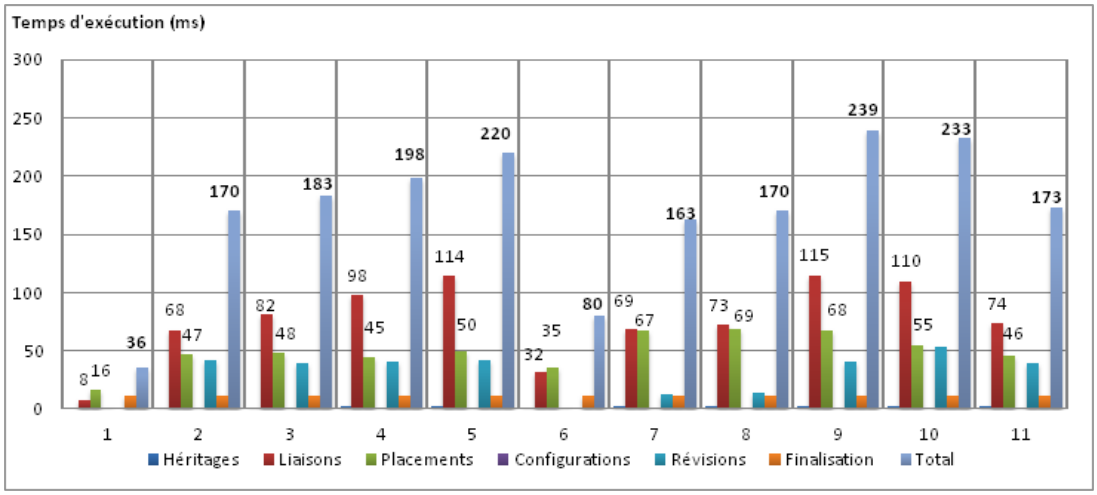


FIGURE 10.9 – Performances de Vulcan sur le Scénario n°4 pour l'application Springoo : le temps de calcul est donné en millisecondes, en fonction des étapes

10.5.1 Test n°4bis

Le test n°4bis correspond à l'ajout au test n°4 de composants et de contraintes permettant la mise en place d'un canevas d'injection de charge pour le profil "Test", et d'un canevas de supervision pour le profil "Production". De façon concrète, ce test correspond à l'injection automatisée de services élastiques tels que ceux pouvant être proposés par une plateforme de cloud. La figure 10.10 montre les modifications opérées pour les étapes n°5 et n°6 : pour le profil "Test", le canevas Clif est automatiquement injecté tandis que pour le profil "Production", il s'agit du canevas de surveillance Ganglia.

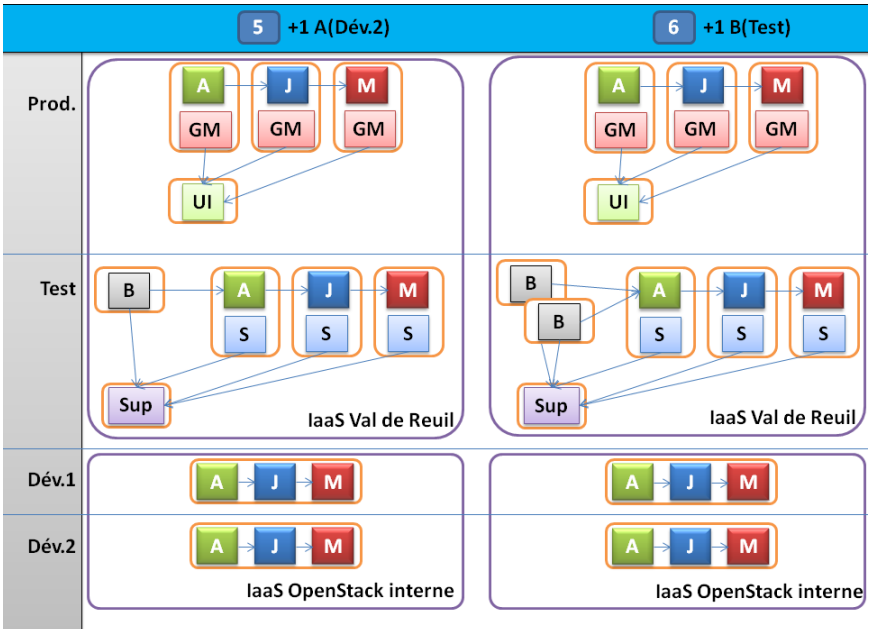


FIGURE 10.10 – Détail des étapes n°5 et 6 du test n°4bis

Comme le montre la figure 10.10, les composants sondes et Gmonds doivent être co-localisés avec les composants de l'application Springoo. Il s'agit d'une contrainte d'ordre 2 qui dépend du placement d'autres composants. Une autre contrainte vise a contrario à supprimer toutes les VMs dans lesquelles aucun composant de l'application Springoo n'est placé. Cela implique donc de supprimer les VMs qui n'hébergent que des composants des canevas Ganglia ou Clif. Il est à noter que cette contrainte obéit au scénario fixé. Celle-ci peut toutefois être modifiée comme par exemple, pour maintenir un pool de VMs démarrées en permanence afin de garantir une meilleure réactivité lors de l'élasticité.

La liste des contraintes rajoutées par rapport au test n°4 est renseignée dans le listing 10.5. Les performances mesurées sont présentées dans le graphique 10.11. Celles-ci se répartissent entre 301 et 1318ms. La durée de 1318ms rapportée à celle de l'approvisionnement d'une VM donne un rapport de 2,2% ce qui reste négligeable. Toutefois, ce test n°4bis montre un phénomène qui concerne la baisse des performances avec la

multiplication des contraintes. Ce phénomène est aussi normal qu'attendu puisqu'avec l'augmentation du nombre de contraintes, le nombre de calculs à effectuer augmente également notamment en raison des dépendances entre contraintes. Pour aller plus loin concernant ce point, la sous-section qui suit présente les performances de Vulcan sur le test n°4ter.

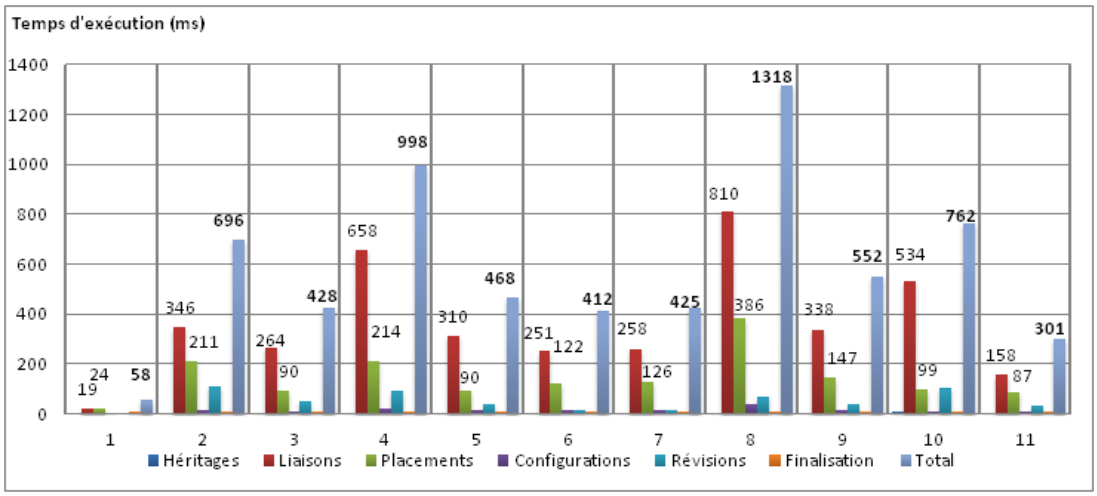


FIGURE 10.11 – Performances de Vulcan sur le Scénario n°4bis pour l'application Springoo : le temps de calcul est donné en millisecondes, en fonction des étapes

Listing 10.5 – Extraits du Modèle par Intention pour le scénario n°4bis

```
                [...]
    <component name="Blade" init="0"/>
    <component name="Probe" init="0"/>
    <component name="Supervisor" init="0"/>
    <component name="GmetadUI" init="0"/>
    <component name="Gmond" init="0"/>
                [...]
<bindings>
                [...]
    <!-- liaisons pour Clif -->
    local:bindAllExactlyOne_group("Blade","Supervisor","injection")
    local:bindAllOne_group("Probe","Supervisor","injection")
    local:bindAllExactlyOne_group("Blade","Apache","injection")
    <!-- liaison de Ganglia -->
    local:bindAllExactlyOne_group("Gmond","GmetadUI","monitoring")
    <!-- suppressions des VMs sans composant propre a Springoo -->
    local:purgeGivenTypesOfInstancesAloneInsideAContainer(
        ("Probe","Gmond"),"vm")
</bindings>

<placements>
                [...]
    <!-- gestion des sondes et Gmonds -->
    local:placeOneIntoEachContainer_group("Probe", "vm", "group", "test")
    local:placeOneOne_ForAllInstancesTypesExceptGivenOnes(("Probe","Gmond"))
    local:placeOneIntoEachContainer_group("Gmond", "vm", "group",
        "production")
    <!-- suppression des VMs vides -->
    local:purgeEmptyContainers("vm")
</placements>
<configurations>
    <!-- contraintes pour isoler  les differents canevas-->
    local:copyPropertiesToColocatedInstancesOfType_group("Probe", "vm",
        "injection", "group", "test")
    local:copyPropertiesToColocatedInstancesOfType_group("Gmond", "vm",
        "monitoring", "group", "production")
</configurations>
```

10.5.2 Test n˚4ter

Dans ce test le sous-scénario n˚4ter est mis en œuvre. Aux contraintes du sous-scénario n˚4bis, celui-ci ajoute une gestion plus fine du placements. Chaque VM est ainsi placée au sein d'un IaaS et le choix du profil matériel le plus proche des exigences exprimées pour la VM est réalisé. Ce choix est évidemment lié au IaaS de destination et des profils matériels proposés, mais également aux placements des composants, ce qui signifie que ce sous-scénario introduit des contraintes dépendant de la réalisation d'autres contraintes, elles-mêmes dépendant de précédentes. Ce sous-scénario introduit donc une forte complexité au niveau des dépendances des contraintes. Le listing 10.6 est un extrait du modèle par intention utilisé qui montre le différentiel au niveau des contraintes avec le précédent test. Les performances mesurées lors de ce test sont quant à elles présentées par la figure 10.12.

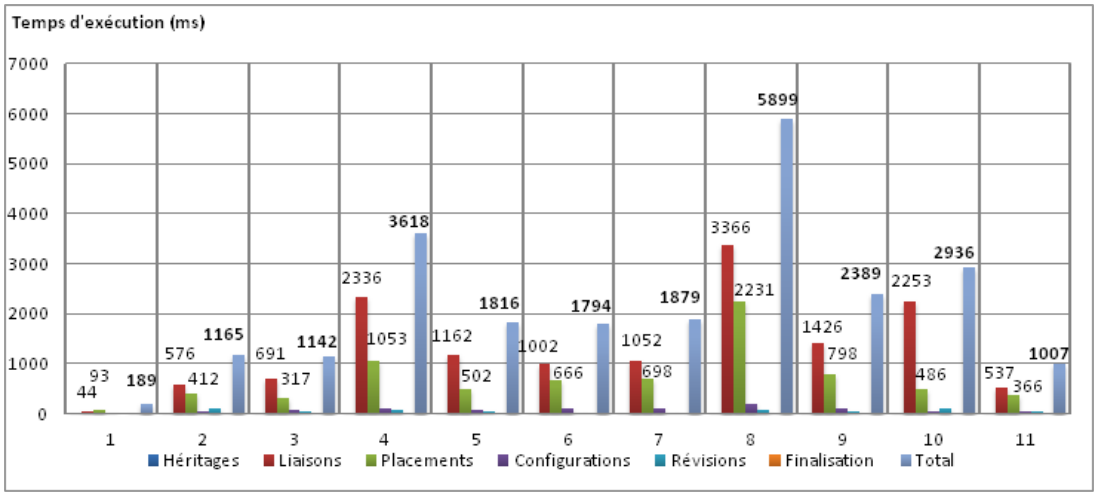


FIGURE 10.12 – Performances de Vulcan sur le Scénario n˚4ter pour l'application Springoo : le temps de calculs est donné en millisecondes, en fonction des étapes

Les mesures des durées pour la réalisation de ce test ont une valeur moyenne de 1142ms. De façon plus détaillée, les valeurs obtenues se répartissent entre 189ms pour l'initalisation du modèle par extension avec l'ajout des IaaS utlisés, et 5,9s. Cette dernière mesure est plus élevée en raison de l'augmentation du nombre de contraintes du modèle par intention. Même si cette durée est plus élevée que les autres, elle reste malgré tout raisonnable au regard de la réactivité moyenne d'un IaaS. En considérant que cette réactivité moyenne se situe à 60s, l'overhead introduit est de 9,8% sur cette mesure. Il est également nécessaire de rappeler que les tests ont été menés sur une plateforme d'entrée de gamme et que le scénario considéré est volontairement complexe. Il s'agit là d'une valeur plus importante que les autres mais raisonnable.

Listing 10.6 – Extraits du Modèle par Intention pour le scénario n°4ter

```
1                   [...]
2 <containers>
3     <container name="iaas" init="0"/>
4 </containers>
5         [...]
6 <placements>
7                   [...]
8     <!-- contraintes sur le iaas de destination en fonction du profil de
                  deploiement
9         - profil dev dans le IaaS OpenStack interne a Orange
10        - profils production et test dans le cloud de Val de Reuil -->
11    local:placeAllContainerOne_group("vm","iaas","group", "dev", "id",
          "openstack-dev")
12    local:placeAllContainerOne_group("vm","iaas","group", "production",
          "id", "iaas_val-de-reuil")
13    local:placeAllContainerOne_group("vm","iaas","group", "test", "id",
          "iaas_val-de-reuil")
14    <!-- contraintes des blades dans le meme IaaS que l application
15    (contrainte etendu aux superviseurs)-->
16    local:placeAccordingToAReferenceInstance("vm", "iaas", "Apache",
          "group", ("Blade","Supervisor"), "injection")
17    <!-- colocalisation dans le meme iaas des GmetadUIs et Springoo-->
18    local:placeAllContainersContainingTypesOfInstances("vm", "iaas", "id",
          "iaas_val-de-reuil", ("GmetadUI"))
19 </placements>
20 <configurations>
21        <!-- choix du profil materiel des VMs en fonction des composants-->
22     local:setContainersProfilesByInstancesContained("vm", ("GmetadUI"),
          <profile cpu="1" ram="1024" diskSize="20GB"/>)
23     local:setContainersProfilesByInstancesContained("vm", ("Blade"),
          <profile cpu="1" ram="1024" diskSize="4GB"/>)
24     local:setContainersProfilesByInstancesContained("vm", ("Supervisor"),
          <profile cpu="1" ram="1024" diskSize="20GB"/>)
25     <!-- choix du profils materiel des VMs en fonction des composants ET
          du profil de deploiement-->
26     local:setContainersProfilesByInstancesContained_group("vm",
          ("Jonas","Apache","Database"), <profile cpu="1" ram="4096"
          diskSize="20GB"/>, "group", "production")
27     local:setContainersProfilesByInstancesContained_group("vm",
          ("Jonas","Apache","Database"), <profile cpu="1" ram="4096"
          diskSize="20GB"/>, "group", "test")
28     local:setContainersProfilesByInstancesContained_group("vm", "Apache",
          <profile cpu="1" ram="2048" diskSize="4GB"/>, "group", "dev")
29     <!-- assignation des adresses IP publiques -->
30     local:setPublicIpOn(("Apache","Supervisor","GmetadUI"),"vm")
31 </configurations>
```

Toutefois, cette hausse du temps de calculs ne remet pas en cause la validation de l'approche proposée, non seulement puisque cette valeur n'est pas aberrante, mais également puisqu'il s'agit d'une limite propre à l'implantation actuelle mais qui peut facilement être corrigée. Dans ce sens, des expérimentations supplémentaires ont été menées afin d'évaluer les bienfaits d'une modification simple de l'implantation actuelle et qui repose en partie sur des fonctionnalités avancées déjà implantées. Toutefois, le cadre de recherche d'une thèse étant limité dans le temps, il n'a pas été possible de réaliser une implantation complète et stable de ces avancées.

Le test n°4ter a donc montré une des limites de l'implantation actuelle : celle-ci n'étant pas parallélisée, l'augmentation du nombre de contraintes résulte dans une augmentation de la durée des calculs de planification. Toutefois, le lecteur averti aura remarqué que ce scénario présente une caractéristique intéressante : l'architecture globale de l'application est en fait constituée de sous-graphes totalement indépendants. Ces sous-graphes n'interagissant pas, il est donc possible et réaliste d'en séparer les traitements. Concrètement, nous avons repris le test n°4ter en procédant à la planification en parallèle de ces sous-graphes ceci en procédant à un pré-traitement judicieux et réaliste du modèle par intention. La figure 10.13 montre les durées maximales observées et le gain de ces améliorations. Seules les étapes les plus couteuses sont données de façon à fournir une

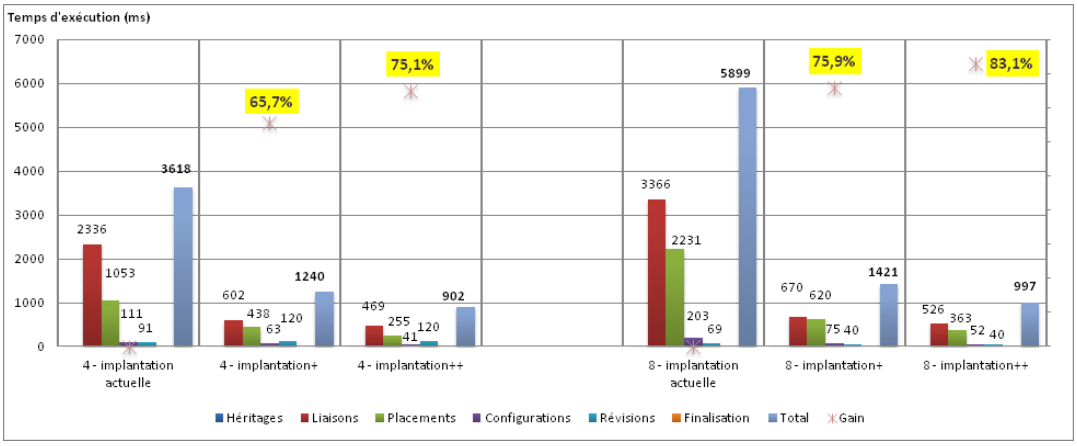


FIGURE 10.13 – Gains possibles de Vulcan en complétant l'implantation actuelle

évaluation de la durée maximale pour le scénario 4ter. La séparation proposée pour les traitements de la planification s'appuie sur la capacité d'auto-élasticité de Vulcan : plusieurs Vulcan Engines sont créés et un *Vulcan Planification Manager* en tête du cluster coordonne les calculs.

Dans ce graphe est faite la mention d'une implantation "+" et d'une implantation "++". La première consiste en une séparation basique des sous-graphes : le modèle par intention est inchangé et seule est opérée une répartition des profils sur plusieurs Vulcan Engines. La seconde est quant à elle plus poussée puisqu'elle effectue une introspection des contraintes du modèle par intention en ne retournant que le sous-ensemble minimal par profil. Concrètement, le Vulcan Engine en charge du profil "Développement" ne

recevra aucune contrainte concernant Ganglia, Clif ou celles concernant les autres profils. Les gains observés sont spectaculaires puisque l'implantation+ offre une réduction des temps de calculs de 65,7% pour l'étape 4 et de 75,9% pour l'étape 8 par rapport à l'implantation stable actuelle. L'implantation++ ajoute un gain supplémentaire de 27,3% et de 29,8% par rapport à l'implantation+, soit respectivement 75,1% et 83,1% de réduction du temps de calcul par rapport à l'implantation stable actuelle.

10.6 Conclusion des tests

Les tests menés dans le cadre de la validation des concepts de la thèse ont permis la vérification des qualités de Vulcan. Des cas d'usages différenciant Vulcan des solutions actuelles ont été montrés et des tests en ont évalué la gestion par l'implantation actuelle stable du prototype. **Ces tests ont été menés de façon non complaisante pour Vulcan**, en utilisant une plateforme d'entrée de gamme et des scénarios pouvant être complexes. **De façon très claire, les temps de calculs sont soit négligeables soit raisonnables** en comparaison de la réactivité moyenne d'un IaaS. D'autre part, des évolutions n'ayant pu être complètement menées à bien ont démontré du potentiel de Vulcan avec des gains très importants par rapport à l'implantation évaluée. **Au regard de ces évaluations, l'approche proposée est donc validée au niveau des performances**. Par ailleurs, les différents exemples de modèles par intention ont illustré **la simplicité d'usage de Vulcan** :

- Pour l'utilisateur débutant ou averti, il suffit d'utiliser des requêtes documentées et présentes dans des bibliothèques.
- L'utilisateur averti peut ajouter de nouvelles requêtes dans ces bibliothèques puisqu'elles sont extensibles.

La suite de ce manuscrit conclue et expose le perspectives d'avenir. Si certaines ont été évoquées ici, beaucoup d'autres existent grâce au potentiel de Vulcan et seront donc discutées.

Chapitre 11

Conclusion

Sommaire

Ce chapitre achève la présentation des travaux de thèse de ce manuscrit. Il conclut en trois sections successives afin de rappeler au lecteur les différents éléments de la proposition. Une première section commence par rappeler le contexte des travaux de thèse et leurs objectifs en vue de les ressituer. Ensuite, une deuxième section rappelle les contributions évoquées dans ce manuscrit et positionne la solution proposée par rapport aux critères utilisés dans la partie de ce manuscrit portant sur l'état de l'art. Enfin, comme les travaux menés ne sont qu'un point de départ, une dernière section en mentionne des perspectives envisagées.

11.1 Rappel du contexte et des motivations

Le contexte des travaux de thèse exposés dans ce manuscrit est le Cloud Computing, un paradigme récent permettant à tout-un-chacun d'accéder à des infrastructures dématérialisées au travers du réseau. Ces infrastructures permettent à leurs utilisateurs d'exécuter des programmes aussi appelées applications. Chaque application est caractérisée par son architecture, c'est-à-dire l'ensemble de ses composants, de leurs liaisons et

de leurs interactions avec le monde extérieur. Le Cloud Computing fournit un environnement d'exécution dont l'unité fondamentale de facturation et d'exécution est la Machine Virtuelle (VM).

Le cloud laisse ses utilisateurs libres de créer ou supprimer des VMs en fonction de leurs besoins, ce qui offre à ces mêmes utilisateurs un moyen d'optimiser leur consommation de VMs : l'élasticité. L'élasticité permet effectivement d'adapter les ressources consommées aux besoins d'une application. Concrètement, une application soumise à une forte charge peut ainsi se voir allouer plus de ressources (i.e. plus de VMs, ou des VMs plus puissantes) de sorte à augmenter son nombre de composants et conséquemment pouvoir soutenir la charge. Afin de pouvoir réellement bénéficier des avantages de l'élasticité, celle-ci doit être automatisée : une boucle automatique opère les allocations et désallocations nécessaires sans qu'aucun humain n'ait à intervenir directement.

Tous les grands acteurs du cloud proposent aujourd'hui leur solution permettant de gérer l'élasticité de façon automatique. Toutefois, celles-ci demeurent marquées par des problématiques d'enfermement propriétaire et de restriction de gestion de l'élasticité. De façon plus détaillée, ces solutions s'appuient sur des simplifications servant à diminuer la complexité de la tâche mais qui bridents clairement leur couverture de l'élasticité. Ces simplifications étant identifiées comme la source des restrictions évoquées, les travaux de thèse ont visé à adresser les problématiques qui leur sont liées. Pour cela, il est nécessaire de permettre la création d'architectures-cibles pour des applications durant leur élasticité avec toute la complexité que cela entraine : il s'agit d'offrir un mécanisme de calcul d'architectures à la fois simple d'utilisation, paramétrable, évolutif et agnostique vis-à-vis de l'application, de son architecture, de ses modifications élastiques et des clouds utilisés. En considérant la boucle MAPE-K, la thématique de la thèse visait à offrir une phase de Planification qui ait l'ensemble de ces caractéristiques et remplisse trois objectifs :

1. Définir un modèle architectural et un formalisme permettant la représentation et la spécification d'applications élastiques déployées dans le cloud.

2. Concevoir et prototyper les mécanismes permettant l'élasticité.

3. Valider l'approche par des cas d'usages réalistes.

La solution proposée répond à l'ensemble des objectifs posés, grâce aux contributions exposées. Le diagramme 11.1 présente l'évaluation de notre solution par rapport aux critères retenus pour dresser un état de l'art des solutions de gestion de l'élasticité des applications dans le cloud.

Grâce à leurs contributions, les travaux présentés permettent de maximiser l'ensemble de ces critères ce qui traduit une réponse à chacun des objectifs de la thèse. La section suivante résume les contributions de Vulcan, la solution proposée dans ce manuscrit.

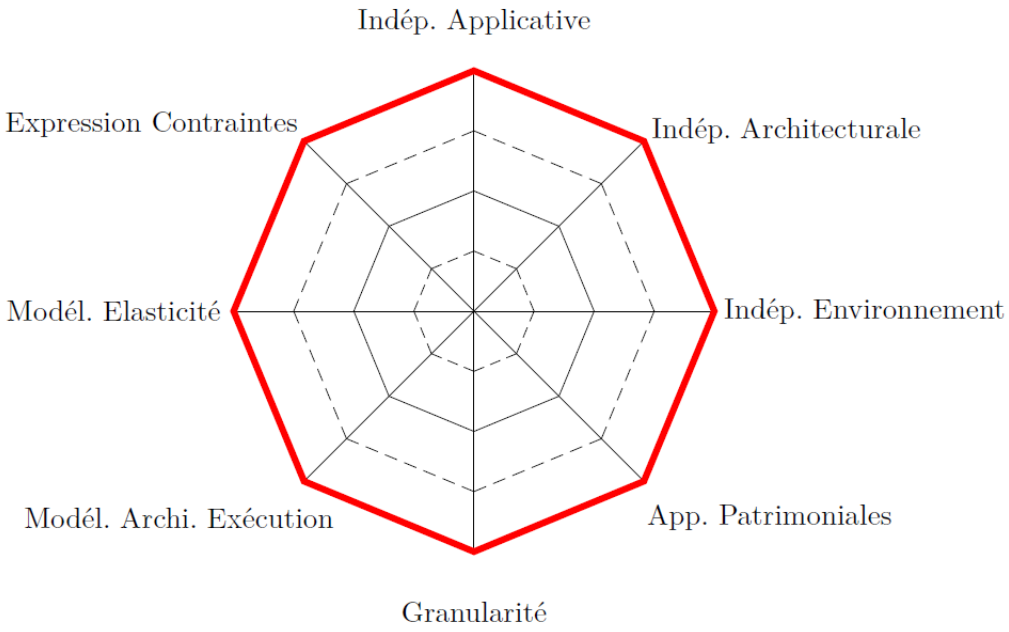


FIGURE 11.1 – Positionnement de Vulcan par rapport aux critères d'évaluation de l'état de l'art

11.2 Contributions

Les contributions de la thèse sont au nombre de trois. La première d'entre elles est un modèle pour les applications élastiques.

Un modèle d'applications élastiques

Ce modèle est en fait un agrégat de deux modèles : le *Modèle par Intention* et le *Modèle par Extension*. Le premier permet à l'utilisateur de décrire les contraintes architecturales de son application durant l'élasticité tandis que le second contient la connaissance de l'architecture courante de l'application, c'est-à-dire la connaissance de son état courant.

Le Modèle par Extension Ce modèle fait usage d'une description architecturale à base de composants et reprend une partie des concepts issus de Fractal. L'architecture de l'application est modélisée par des composants et des conteneurs. Elle mentionne également des placements, des liaisons et des paramètres de configuration. Il s'agit d'un modèle simple mais non simpliste qui propose une approche pragmatique permettant, au travers d'un nombre réduit mais suffisant de modifications, de parvenir à établir des architectures-cibles pour une application durant son élasticité. Le modèle par extension est au centre des calculs d'architectures-cibles puisque s'il est utilisé comme connaissance de l'architecture courante, il est également le siège des calculs effectués. Ces calculs sont

régis par des contraintes, celles renseignées dans la seconde partie du modèle global proposé : le Modèle par Intention.

Le Modèle par Intention Le modèle par intention est la description de l'ensemble des contraintes devant être satisfaites durant les calculs d'architectures-cibles. Il fait usage d'un formalisme innovant quant à son utilisation puisqu'il a recours à des requêtes ensemblistes pour l'écriture de contraintes. Ces requêtes permettent la description des modifications à opérer sur des composants ou conteneurs du modèle par extension, ceci afin de satisfaire des exigences de l'utilisateur.

Quatre types de contraintes ont été identifiées, elles concernent l'Héritage, les Liaisons, les Placements et les Configurations. A chacun de ces types correspond un ensemble fini de modifications du modèle par extension courant. Les ensembles des quatre types de contraintes sont à la fois disjoints et complémentaires, ce qui permet la mise en place de la deuxième contribution de cette thèse : un algorithme de Planification.

L'algorithme de Planification

La deuxième contribution des travaux de thèse exposés dans ce manuscrit consiste en un algorithme de Planification. Celui-ci est à base de raisonnement révisable, une notion issue de l'intelligence artificielle. Il offre des capacités extraordinaires de gestion des erreurs pouvant se produire lors des calculs d'architectures-cibles. Ce mode de raisonnement original permet la composition automatique de contraintes simples ce qui adresse des contraintes nettement plus complexes. Son utilisation permet à la solution proposée d'offrir une composition de contraintes compréhensible pour un administrateur d'application : celui-ci n'a plus qu'à énumérer un ensemble de contraintes simples afin d'adresser les cas d'élasticité les plus élaborées.

Ce mode de raisonnement offre, en plus de sa simplicité, une extension du champ des possibles en matière d'élasticité. Vulcan est ainsi caractérisé par une couverture inégalée jusqu'à présent en matière de gestion des scénarios d'élasticité.

Les deux premières contributions forment un ensemble cohérent et innovant dont les concepts ont été évalués et validés par la troisième contribution de la thèse qui est une première implantation de Vulcan.

L'implantation d'un prototype de validation

Les concepts des deux premières contributions ont été implantés dans un prototype. Ce prototype a permis de valider l'approche proposée grâce à ses performances, sa simplicité d'usage, son extensibilité et à sa flexibilité. Ces caractéristiques lui confèrent un potentiel réellement intéressant en matière de perspectives de recherche. L'implantation évaluée souffre de quelques limites qui ont été identifiées et des solutions ont été non seulement proposées mais également pré-évaluées.

Ainsi Vulcan a démontré ses capacités à offrir une élasticité repoussant les limites des solutions connues d'une façon vraiment significative. L'évaluation a validé les concepts de l'approche sur des cas d'usages réalistes et qui obéissent à de vrais besoins. Les

perspectives de Vulcan sont elles aussi réelles et sont justement exposées dans la section suivante.

11.3 Perspectives

Les perspectives de Vulcan découlent des limites actuelles identifiées mais également des nouvelles possibilités apportées par les caractéristiques intrinsèques de Vulcan. Cette section les décrit.

11.3.1 Le Prefetching

Une première perspective de recherche concerne le *prefetching*, c'est-à-dire le calcul d'architectures cibles par anticipation. L'idée serait de calculer en avance de phase les futures architectures-cibles possibles. Une façon d'aborder cette problématique pourrait être de réaliser un auto-apprentissage qui soit capable de prédire les futures décisions d'élasticité de sorte à maintenir un cache de pré-calculs. Une telle prédiction est par exemple présente dans les processeurs puisqu'ils utilisent de la prédiction de branchement afin de maintenir de bonnes performances : cette perspective est donc réaliste.

L'auto-élasticité de Vulcan est un élément facilitateur de cette perspective puisqu'elle permet de lancer plusieurs Vulcan Engines chacun pouvant pré-calculer une architecture cible par anticipation d'une ou plusieurs décisions d'opérations élastiques. La complexité de cette perspective réside dans la prédiction des bons ensembles d'opérations élastiques.

11.3.2 Gestion du changement de version à chaud

Une seconde perspective est la gestion avancée du changement de version à chaud. Il s'agit d'adresser la mise à jour des éléments applicatifs durant l'exécution de l'application. Cette thématique est effectivement primordiale pour des opérationnels devant garantir la stabilité d'une application au cours du temps, y compris durant les phases de mises à jour. Les mécanismes d'élasticité de Vulcan ont une granularité plus fine que les solutions actuelles puisqu'elle se situe au niveau du composant et de sa configuration. Cela fait de Vulcan une approche de choix pour ce genre de problématiques.

11.3.3 Problématiques de fiabilité et de résilience

Les problématiques de la fiabilité et de la résilience peuvent être abordés en suivant trois aspects.

Le premier concerne la résistance aux pannes des VMs de Vulcan. Vulcan s'exécute effectivement sur une ou plusieurs VMs pour gérer des applications dans le cloud et il peut arriver que ses VMs entrent en erreur. Un mécanisme de reprise sur panne serait donc une avancée puisqu'il permettrait à Vulcan d'être fiabilisé. L'auto-élasticité de Vulcan constitue un embryon de cette fiabilité mais il manque encore le partage d'état ainsi que la détection de panne. L'un comme l'autre sont réalisables et il serait même possible d'offrir à terme un système complet de résilience pour Vulcan.

Le deuxième aspect de ces problématiques est relatif aux calculs du Vulcan Engine puisque si l'implantation actuelle permet une grande tolérance aux pannes grâce à son algorithme à base de raisonnement révisable, la terminaison des calculs n'est effective que si les contraintes sont correctement écrites. L'implantation actuelle permet déjà un suivi de son fonctionnement par un cerveau-B tel que décrit par Minsky. Cependant la réalisation complète d'un tel cerveau-B serait une étape déterminante afin de permettre une fiabilité des calculs et même une résilience en proposant un mécanisme intelligent de reprise sur panne. Il serait par exemple possible de détecter des boucles voir même de les anticiper à partir des contraintes renseignées par l'utilisateur dans le modèle par intention.

Le troisième et dernier aspect de la fiabilité et de la résilience concerne les applications elles-mêmes. Vulcan est effectivement un moteur de calcul dynamique d'architectures qui peut facilement être étendu. Si Vulcan permet déjà d'assurer la fiabilité d'une application en créant par exemple une architecture hautement disponible, il peut être étendu afin de mettre en place les mécanismes nécessaires à la résilience des applications.

11.3.4 Optimisation des performances

Une quatrième perspective concerne l'optimisation des performances. Là encore, la réalisation complète d'un cerveau-B serait une façon pour un utilisateur d'entrevoir comment améliorer les contraintes qu'il crée dans la mesure où cet élément permettrait un suivi très fin des phases de calculs. Le cerveau-B pourrait même interagir avec une sorte de système expert pouvant conseiller l'utilisateur durant la création de contraintes. L'optimisation peut également être abordée par des mécanismes tels que ceux pré-testés en fin d'évaluation. Il s'agirait de minimiser les calculs à effectuer de sorte à gagner en performances. Ce type d'approche peut également être complété par une phase de compilation des requêtes puisque l'implantation actuelle repose sur du langage interprété.

11.3.5 Perspectives industrielles

Outre ces perspectives orientées recherche, Vulcan présente des perspectives au niveau de son outillage dans une optique industrielle. Par exemple, un modeleur graphique permettant de créer un modèle par intention peut être envisagé et créé très facilement. Par ailleurs l'amélioration des outils relatifs au debug et à la composition de modèles par intention serait un plus, apprécié des développeurs.

Bibliographie

[1] Amazon cloudformation sur le site web d'amazon. https://aws.amazon.com/fr/cloudformation/.

[2] Description de l'approche DevOps. http://www.rajiv.com/blog/2009/03/17/technology-department/.

[3] Documentation du routeur en langage go de cloudfoundry. https://github.com/cloudfoundry/gorouter.

[4] Documentation sur le site web de la communauté cloud foundry. http://docs.cloudfoundry.org/.

[5] Documentation sur le site web de la plateforme jelastic. http://docs.jelastic.com/.

[6] Documentation sur le site web d'heroku. https://devcenter.heroku.com/.

[7] Dépôt du code du projet dokku. https://github.com/progrium/dokku.

[8] FScript-FPath. http://fractal.ow2.org/fscript/index.html.

[9] Présentation des travaux portant sur le cloud computing par siemens (SATURN2010). http://www.sei.cmu.edu/library/assets/presentations/Cloud%20Computing%20Architecture%20-%20Gerald%20Kaefer.pdf.

[10] Site officiel du projet OpenStack. http://www.openstack.org/.

[11] Site web d'Amazon EC2. https://aws.amazon.com/fr/ec2/.

[12] Site web de la plateforme deis. http://deis.io/.

[13] Site web de l'offre Office 365. http://office.microsoft.com/fr-001/.

[14] Site web de microsoft azure. http://www.microsoft.com/windowsazure/.

[15] Site web de redhat openshift. https://www.openshift.com/.

[16] Site web de salesforce. https://www.salesforce.com/heroku/.

[17] Site web du projet docker. `https://www.docker.io/`.

[18] Site web du projet flynn. `https://flynn.io/`.

[19] *2012 IEEE Network Operations and Management Symposium, Maui, HI, USA, April 16-20, 2012*. IEEE, 2012.

[20] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. Deconstructing amazon ec2 spot instance pricing. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 304–311, Nov 2011.

[21] Divyakant Agrawal, Amr El Abbadi, Sudipto Das, and Aaron J. Elmore. Database scalability, elasticity, and autonomy in the cloud - (extended abstract). In Jeffrey Xu Yu, Myoung-Ho Kim, and Rainer Unland, editors, *DASFAA (1)*, volume 6587 of *Lecture Notes in Computer Science*, pages 2–15. Springer, 2011.

[22] H.T. al Feel and M.H. Khafagy. Ocss : Ontology cloud storage system. In *Network Cloud Computing and Applications (NCCA), 2011 First International Symposium on*, pages 9–13, Nov 2011.

[23] Ahmed Ali-Eldin, Johan Tordsson, and Erik Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *NOMS* [19], pages 204–212.

[24] Mourad Amziani, Kais Klai, Tarek Melliti, and Samir Tata. Time-based evaluation of service-based business process elasticity in the cloud. In *CloudCom (1)*, pages 573–580. IEEE, 2013.

[25] Mourad Amziani, Tarek Melliti, and Samir Tata. Formal modeling and evaluation of stateful service-based business process elasticity in the cloud. In Robert Meersman, Hervé Panetto, Tharam S. Dillon, Johann Eder, Zohra Bellahsene, Norbert Ritter, Pieter De Leenheer, and Dejing Dou, editors, *OTM Conferences*, volume 8185 of *Lecture Notes in Computer Science*, pages 21–38. Springer, 2013.

[26] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds : A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[27] B. Baikie and L. Hosman. Green cloud computing in developing regions moving data and processing closer to the end user. In *Telecom World (ITU WT), 2011 Technical Symposium at ITU*, pages 24–28, Oct 2011.

[28] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[29] L. Bass, R. Jeffery, H. Wada, I Weber, and Liming Zhu. Eliciting operations requirements for applications. In *Release Engineering (RELENG), 2013 1st International Workshop on*, pages 5–8, May 2013.

[30] Hubert Baumeister, Michele Marchesi, and Mike Holcombe, editors. *Extreme Programming and Agile Processes in Software Engineering, 6th International Conference, XP 2005, Sheffield, UK, June 18-23, 2005, Proceedings*, volume 3556 of *Lecture Notes in Computer Science*. Springer, 2005.

[31] Z. Bellahsene. Identifying virtual composite objects : a step forward to update object views. In *Database and Expert Systems Applications, 1997. Proceedings., Eighth International Workshop on*, pages 523–528, Sept 1997.

[32] Anton Beloglazov and Rajkumar Buyya. Energy efficient allocation of virtual machines in cloud data centers. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 577–578, Washington, DC, USA, 2010. IEEE Computer Society.

[33] P.V. Beserra, A. Camara, R. Ximenes, A.B. Albuquerque, and N.C. Mendonca. Cloudstep : A step-by-step decision process to support legacy application migration to the cloud. In *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2012 IEEE 6th International Workshop on the*, pages 7–16, Sept 2012.

[34] V.K. Bhise and A.S. Mali. Ec2 instance provisioning for cost optimization. In *Advances in Computing, Communications and Informatics (ICACCI), 2013 International Conference on*, pages 1891–1895, Aug 2013.

[35] B Boehm. A spiral model of software development and enhancement. *SIGSOFT Softw. Eng. Notes*, 11(4) :14–24, August 1986.

[36] Sara Bouchenak, Noel De Palma, Daniel Hagimont, and Christophe Taton. Autonomic management of clustered applications. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1–11. IEEE, 2006.

[37] James Broberg, Rajkumar Buyya, and Zahir Tari. Metacdn : Harnessing 'storage clouds' for high performance content delivery. *Journal of Network and Computer Applications*, 32(5) :1012 – 1022, 2009. Next Generation Content Networks.

[38] Laurent Broto, Daniel Hagimont, Patricia Stolf, Noel Depalma, and Suzy Temate. Autonomic management policy specification in tune. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 1658–1663. ACM, 2008.

[39] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Softw., Pract. Exper.*, 36(11-12) :1257–1284, 2006.

[40] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging it platforms : Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Comp. Syst.*, 25(6) :599–616, 2009.

[41] Eddy Caron, Luis Rodero-Merino, Frédéric Desprez, and Adrian Muresan. Auto-Scaling, Load Balancing and Monitoring in Commercial and Open-Source Clouds. Rapport de recherche RR-7857, INRIA, February 2012.

[42] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, pages 577–589, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[43] Lin Wah Chan, René Hexel, and Lian Wen. Rule-based behaviour engineering : Integrated, intuitive formal rule modelling. In *Australian Software Engineering Conference*, pages 20–29. IEEE, 2013.

[44] Gang Cheng, Yongxin Zhu, Guoguang Rong, and Meikang Qiu. Prototyping high efficiency cloud computing architecture : Implementation of a content delivery network server on fpga. In *Computing and Convergence Technology (ICCCT), 2012 7th International Conference on*, pages 1120–1124, Dec 2012.

[45] Shang-Wen Cheng. *Rainbow : Cost-Effective Software Architecture-Based Self-Adaptation*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, May 2008. `http://reports-archive.adm.cs.cmu.edu/anon/isr2008/CMU-ISR-08-113.pdf`.

[46] Shang-Wen Cheng, An-Cheng Huang, David Garlan, Bradley R. Schmerl, and Peter Steenkiste. Rainbow : Architecture-based self-adaptation with reusable infrastructure. In *ICAC*, pages 276–277. IEEE Computer Society, 2004.

[47] M.B. Chhetri, S. Chichin, Quoc Bao Vo, and R. Kowalczyk. Smart cloud broker : Finding your home in the clouds. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 698–701, Nov 2013.

[48] M. Condry, U. Gall, and P. Delisle. Open service gateway architecture overview. In *Industrial Electronics Society, 1999. IECON '99 Proceedings. The 25th Annual Conference of the IEEE*, volume 2, pages 735–742 vol.2, 1999.

[49] Ivica Crnkovic, Séverine Sentilles, Aneta Vulgarakis, and Michel R. V. Chaudron. A classification framework for software component models. *IEEE Trans. Software Eng.*, 37(5) :593–615, 2011.

[50] Krzysztof Czarnecki, Chang Hwan Peter Kim, and Karl Trygve Kalleberg. Feature models are views on ontologies. In *Proceedings of the 10th International on Software Product Line Conference*, SPLC '06, pages 41–51, Washington, DC, USA, 2006. IEEE Computer Society.

[51] Bruno Dillenseger. Clif, a framework based on fractal for flexible, distributed load testing. *Annales des Télécommunications*, 64(1-2) :101–120, 2009.

[52] Yannis Dimopoulos, Pavlos Moraitis, and Alexis Tsoukiàs. Argumentation based modeling of decision aiding for autonomous agents. In *IAT*, pages 99–105. IEEE Computer Society, 2004.

[53] Corentin Dupont, Thomas Schulze, Giovanni Giuliani, Andrey Somov, and Fabien Hermenier. An energy aware framework for virtual machine placement in cloud federated data centres. In Marco Ajmone Marsan, Suresh Goyal, Shugong Xu, Antonio Fernández, Milan Prodanovic, and Ken Christensen, editors, *e-Energy*, page 4. ACM, 2012.

[54] Xavier Etchevers. *Déploiement d'applications patrimoniales en environnements de type informatique dans le nuage*. PhD thesis, Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique, Décembre 2012.

[55] Xavier Etchevers, Thierry Coupaye, Fabienne Boyer, and Noel De Palma. Self-configuration of distributed applications in the cloud. In Liu and Parashar [100], pages 668–675.

[56] Z. Fadika and M. Govindaraju. Delma : Dynamically elastic mapreduce framework for cpu-intensive applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 454–463, May 2011.

[57] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. In Carlo Ghezzi, Mehdi Jazayeri, and Alexander L. Wolf, editors, *ICSE*, pages 407–416. ACM, 2000.

[58] Fermín Galán, Américo Sampaio, Luis Rodero-Merino, Irit Loy, Victor Gil, and Luis Miguel Vaquero. Service specification in cloud environments based on extensions to open standards. In Jan Bosch and Siobhán Clarke, editors, *COMSWARE*, page 19. ACM, 2009.

[59] Guilherme Galante and Luis Carlos Erpen De Bona. A survey on cloud computing elasticity. In *UCC*, pages 263–270. IEEE, 2012.

[60] Sergio García-Gómez, Manuel Escriche-Vicente, Pablo Arozarena-Llopis, Francesco Lelli, Yehia Taher, Christof Momm, Axel Spriestersbach, Jürgen Vogel, Andrea Giessmann, Frederic Junker, Miguel Jiménez-Gañán, József Bíró, Goulven Le-Jeune, Michel Dao, Stephane P. Carrie, Jörg Niemöller, and Dimitri Mazmanov. 4caast : Comprehensive management of cloud services through a paas. In *ISPA*, pages 494–499. IEEE, 2012.

[61] B. Gayathri. Green cloud computing. In *Sustainable Energy and Intelligent Systems (SEISCON 2012), IET Chennai 3rd International on*, pages 1–5, Dec 2012.

[62] Hamoun Ghanbari, Bradley Simmons, Marin Litoiu, Cornel Barna, and Gabriel Iszlai. Optimal autoscaling in a iaas cloud. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, pages 173–178, New York, NY, USA, 2012. ACM.

[63] Hamoun Ghanbari, Bradley Simmons, Marin Litoiu, and Gabriel Iszlai. Exploring alternative approaches to implement an elasticity policy. In Liu and Parashar [100], pages 716–723.

[64] Patrick Goldsack, Julio Guijarro, Steve Loughran, Alistair Coles, Andrew Farrell, Antonio Lain, Paul Murray, and Peter Toft. The smartfrog configuration management framework. *SIGOPS Oper. Syst. Rev.*, 43(1) :16–25, January 2009.

[65] Luis Miguel Vaquero Gonzalez, Luis Rodero-Merino, Juan Caceres, and Maik A. Lindner. A break in the clouds : towards a cloud definition. *Computer Communication Review*, 39(1) :50–55, 2009.

[66] D.N. Gordin and A.J. Pasik. Set-oriented constructs for rule-based systems. In *Artificial Intelligence Applications, 1991. Proceedings., Seventh IEEE Conference on*, volume i, pages 76–80, Feb 1991.

[67] T. Gunarathne, Tak-Lon Wu, J. Qiu, and G. Fox. Mapreduce in the clouds for science. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 565–572, Nov 2010.

[68] Daniel Hagimont, Patricia Stolf, Laurent Broto, and Noel De Palma. Component-based autonomic management for legacy software. In Yan Zhang, Laurence Tianruo Yang, and Mieso K. Denko, editors, *Autonomic Computing and Networking*, pages 83–104. Springer US, 2009.

[69] Jing Han, E. Haihong, Guan Le, and Jian Du. Survey on nosql database. In *Pervasive Computing and Applications (ICPCA), 2011 6th International Conference on*, pages 363–366, Oct 2011.

[70] Rui Han, Li Guo, Moustafa Ghanem, and Yike Guo. Lightweight resource scaling for cloud applications. In *CCGRID*, pages 644–651. IEEE, 2012.

[71] Salim Hariri and Alan Sill, editors. *ACM Cloud and Autonomic Computing Conference, CAC '13, Miami, FL, USA - August 05 - 09, 2013*. ACM, 2013.

[72] Masum Z. Hasan, Edgar Magana, Alexander Clemm, Lew Tucker, and Sree Lakshmi D. Gudreddi. Integrated and autonomic cloud resource scaling. In *NOMS* [19], pages 1327–1334.

[73] Sijin He, Li Guo, Yike Guo, Chao Wu, M. Ghanem, and Rui Han. Elastic application container : A lightweight approach for cloud resource provisioning. In *Advanced Information Networking and Applications (AINA), 2012 IEEE 26th International Conference on*, pages 15–22, March 2012.

[74] A. Hegedus, A. Horvath, I. Rath, and D. Varro. A model-driven framework for guided design space exploration. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 173–182, Nov 2011.

[75] Fabien Hermenier. *Gestion dynamique des tâches dans les grappes, une approche à base de machines virtuelles*. PhD thesis, Ecole Nationale Supérieure des Techniques Industrielles et des Mines de Nantes, Novembre 2009.

[76] Z. Hill and M. Humphrey. A quantitative analysis of high performance computing with amazon's ec2 infrastructure : The death of the local cluster ? In *Grid Computing, 2009 10th IEEE/ACM International Conference on*, pages 26–33, Oct 2009.

[77] Martin Hofmann. *Extensional constructs in intensional type theory*. CPHC/BCS distinguished dissertations. Springer, 1997.

[78] S. Hosono and Y. Shimomura. Application lifecycle kit for mass customization on paas platforms. In *Services (SERVICES), 2012 IEEE Eighth World Congress on*, pages 397–398, June 2012.

[79] He Huang, Liqiang Wang, Byung-Chul Tak, Long Wang, and Chunqiang Tang. Cap3 : A cloud auto-provisioning framework for parallel processing using on-demand and spot instances. In *IEEE CLOUD*, pages 228–235. IEEE, 2013.

[80] Emir Imamagic and Dobrisa Dobrenic. Grid infrastructure monitoring system based on nagios. In *Proceedings of the 2007 Workshop on Grid Monitoring*, GMW '07, pages 23–28, New York, NY, USA, 2007. ACM.

[81] A Jain, M. Mishra, S.K. Peddoju, and N. Jain. Energy efficient computing- green cloud computing. In *Energy Efficient Technologies for Sustainability (ICEETS), 2013 International Conference on*, pages 978–982, April 2013.

[82] R. Kanagasabai, Le Duy Ngan, Yuzhang Feng, A. Veeramani, J.K.C. En, C.C. Keong, F.S. Tsai, and A. Andrzejak. Ec2bargainhunter : It's easy to hunt for cost savings on amazon ec2 ! In *Services (SERVICES), 2013 IEEE Ninth World Congress on*, pages 480–487, June 2013.

[83] Matthias Keller, Christoph Robbert, and Manuel Peuster. An evaluation testbed for adaptive, topology-aware deployment of elastic applications. In Dah Ming Chiu, Jia Wang, Paul Barford, and Srinivasan Seshan, editors, *SIGCOMM*, pages 469–470. ACM, 2013.

[84] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1) :41–50, 2003.

[85] M.H. Khafagy and H.T.A. Feel. Distributed ontology cloud storage system. In *Network Cloud Computing and Applications (NCCA), 2012 Second Symposium on*, pages 48–52, Dec 2012.

[86] Kais Klai and Samir Tata. Formal modeling of elastic service-based business processes. In *IEEE SCC*, pages 424–431. IEEE, 2013.

[87] Thomas Knauth and Christof Fetzer. Scaling non-elastic applications using virtual machines. In Liu and Parashar [100], pages 468–475.

[88] Yousri Kouki. *Approche dirigée par les contrats de niveaux de service pour la gestion de l'élasticité du "nuage"*. PhD thesis, Ecole Nationale Supérieure des Mines de Nantes, Décembre 2013.

[89] Yousri Kouki and Thomas Ledoux. Csla : A language for improving cloud sla management. In Frank Leymann, Ivan Ivanov, Marten van Sinderen, and Tony Shan, editors, *CLOSER*, pages 586–591. SciTePress, 2012.

[90] Pavlos Kranas, Vasilios Anagnostopoulos, Andreas Menychtas, and Theodora A. Varvarigou. Elaas : An innovative elasticity as a service framework for dynamic management across the cloud stack layers. In Leonard Barolli, Fatos Xhafa, Salvatore Vitabile, and Minoru Uehara, editors, *CISIS*, pages 1042–1049. IEEE, 2012.

[91] Lobna Kriaa, Aimen Bouchhima, Wassim Youssef, Frédéric Pétrot, Anne-Marie Fouillart, and Ahmed Amine Jerraya. Service based component design approach for flexible hardware/software interface modeling. In *IEEE International Workshop on Rapid System Prototyping*, pages 156–162. IEEE Computer Society, 2006.

[92] Philippe Kruchten. Tutorial : Introduction to the rational unified process&reg ;. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 703–703, New York, NY, USA, 2002. ACM.

[93] Tony Lacy-Thompson. *Informix-SQL - a tutorial and reference.* Prentice Hall, 1969.

[94] Palden Lama and Xiaobo Zhou. Autonomic provisioning with self-adaptive neural fuzzy control for percentile-based delay guarantee. *ACM Trans. Auton. Adapt. Syst.*, 8(2) :9 :1–9 :31, July 2013.

[95] Cheng Li, I. Goiri, A. Bhattacharjee, R. Bianchini, and T.D. Nguyen. Quantifying and improving i/o predictability in virtualized systems. In *Quality of Service (IWQoS), 2013 IEEE/ACM 21st International Symposium on*, pages 1–6, June 2013.

[96] Yishan Li and S. Manoharan. A performance comparison of sql and nosql databases. In *Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference on*, pages 15–19, Aug 2013.

[97] Chia-Feng Lin, Muh-Chyi Leu, Chih-Wei Chang, and Shyan-Ming Yuan. The study and methods for cloud based cdn. In *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2011 International Conference on*, pages 469–475, Oct 2011.

[98] Yu-Shiang Lin, Chun-Yuan Lin, and Yeh-Ching Chung. Gpu-based cloud service for multiple sequence alignments with regular expression constrains. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 741–746, Dec 2012.

[99] Li Ling, Ma Xiaozhen, and Huang Yulan. Cdn cloud : A novel scheme for combining cdn and cloud computing. In *Measurement, Information and Control (ICMIC), 2013 International Conference on*, volume 01, pages 687–690, Aug 2013.

[100] Ling Liu and Manish Parashar, editors. *IEEE International Conference on Cloud Computing, CLOUD 2011, Washington, DC, USA, 4-9 July, 2011*. IEEE, 2011.

[101] Kuan Lu, R. Yahyapour, P. Wieder, C. Kotsokalis, E. Yaqub, and A.I. Jehangiri. Qos-aware vm placement in multi-domain service level agreements scenarios. In *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, pages 661–668, June 2013.

[102] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system : Design, implementation and experience. *Parallel Computing*, 30 :2004, 2003.

[103] John McDermid and Knut Ripken. Life cycle support in the ada environment. *Ada Lett.*, III(1) :57–62, July 1983.

[104] G.A McGilvary, J. Rius, I Goiri, F. Solsona, A Barker, and M. Atkinson. C2ms : Dynamic monitoring and management of cloud infrastructures. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 1, pages 290–297, Dec 2013.

[105] Peter Mell and Tim Grance. Cloud Computing Definition, June 2009. `http://csrc.nist.gov/groups/SNS/cloud-computing/index.html`.

[106] M. Menzel, M. Klems, Hoang Anh Le, and S. Tai. A configuration crawler for virtual appliances in compute clouds. In *Cloud Engineering (IC2E), 2013 IEEE International Conference on*, pages 201–209, March 2013.

[107] Mathias Meyer. Continuous integration and its tools. *IEEE Software*, 31(3) :14–16, 2014.

[108] Maged M. Michael, José E. Moreira, Doron Shiloach, and Robert W. Wisniewski. Scale-up x scale-out : A case study using nutch/lucene. In *IPDPS*, pages 1–8. IEEE, 2007.

[109] Marvin Minsky. *La Société de l'Esprit*. InterEditions, 1988.

[110] F. Moscato, R. Aversa, B. Di Martino, T. Fortis, and V. Munteanu. An analysis of mosaic ontology for cloud resources annotation. In *Computer Science and Information Systems (FedCSIS), 2011 Federated Conference on*, pages 973–980, Sept 2011.

[111] V.S.K. Nagireddi and S. Mishra. A naive approach for cloud service discovery mechanism using ontology. In *Parallel Computing Technologies (PARCOMPTECH), 2013 National Conference on*, pages 1–7, Feb 2013.

[112] Hideo Nishimura, Eriko Iwasa, Michio Irie, Satoshi Kondoh, Masashi Kaneko, Takeshi Fukumoto, Masami Iio, and Kiyoshi Ueda. Applying flexibility in scale-out-based web cloud to future telecommunication session control systems. In *ICIN*, pages 1–7. IEEE, 2012.

[113] T. Nishiyama, S. Yamagiwa, and T. Hisamitsu. Prototyping gpu-based cloud system for iodp core image database. In *Networking and Computing (ICNC), 2011 Second International Conference on*, pages 327–331, Nov 2011.

[114] Noel De Palma, Sara Bouchenak, Fabienne Boyer, Daniel Hagimont, Sylvain Sicard, and Christophe Taton. Jade, un environnement d'administration autonome. *Technique et Science Informatiques*, 27(9-10) :1225–1252, 2008.

[115] Fawaz Paraiso. *soCloud : une plateforme multi-nuages distribuée pour la conception, le déploiement et l'exécution d'applications distribuées à large échelle*. These, Université des Sciences et Technologie de Lille - Lille I, June 2014.

[116] Fawaz Paraiso, Nicolas Haderer, Philippe Merle, Romain Rouvoy, and Lionel Seinturier. A federated multi-cloud paas infrastructure. In Rong Chang, editor, *IEEE CLOUD*, pages 392–399. IEEE, 2012.

[117] Fawaz Paraiso, Philippe Merle, and Lionel Seinturier. Managing elasticity across multiple cloud providers. In *Proceedings of the 2013 International Workshop on Multi-cloud Applications and Federated Clouds*, MultiCloud '13, pages 53–60, New York, NY, USA, 2013. ACM.

[118] Fawaz Paraiso, Philippe Merle, and Lionel Seinturier. socloud : a service-oriented component-based paas for managing portability, provisioning, elasticity, and high availability across multiple clouds. *Computing*, pages 1–27, 2014.

[119] Carlos Parra, Daniel Romero, Sébastien Mosser, Romain Rouvoy, Laurence Duchien, and Lionel Seinturier. Using constraint-based optimization and variability to support continuous self-adaptation. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 486–491, New York, NY, USA, 2012. ACM.

[120] R. Pawlak, S. Lefebvre, Z. Kazi-Aoul, and R. Chiky. Cloud elasticity for implementing an agricultural weather service. In *New Technologies of Distributed Systems (NOTERE), 2011 11th Annual International Conference on*, pages 1–8, May 2011.

[121] Yanbin Peng, Zhijun Zheng, and Xueqin Jiang. Revising defeasible autonomic computing system. In *Proceedings of the 3rd International Conference on Anti-Counterfeiting, Security, and Identification in Communication*, ASID'09, pages 476–479, Piscataway, NJ, USA, 2009. IEEE Press.

[122] Frédéric Pétrot, Denis Hommais, and Alain Greiner. A simulation environment for core based embedded systems. In *Annual Simulation Symposium*, pages 86–91. IEEE Computer Society, 1997.

[123] Pooja and A Pandey. Virtual machine performance measurement. In *Engineering and Computational Sciences (RAECS), 2014 Recent Advances in*, pages 1–3, March 2014.

[124] Clément Quinton, Nicolas Haderer, Romain Rouvoy, and Laurence Duchien. Towards multi-cloud configurations using feature models and ontologies. In *Proceedings of the 2013 International Workshop on Multi-cloud Applications and Federated Clouds*, MultiCloud '13, pages 21–26, New York, NY, USA, 2013. ACM.

[125] Clément Quinton, Daniel Romero, and Laurence Duchien. Cardinality-based feature models with constraints : a pragmatic approach. In Tomoji Kishi, Stan Jarzabek, and Stefania Gnesi, editors, *SPLC*, pages 162–166. ACM, 2013.

[126] Clément Quinton, Daniel Romero, and Laurence Duchien. Handling Constraints in Cardinality-Based Feature Models : The Cloud Environment Case Study. Rapport de recherche RR-8478, INRIA, February 2014.

[127] Clément Quinton, Romain Rouvoy, and Laurence Duchien. Leveraging Feature Models to Configure Virtual Appliances. In *CloudCP - 2nd International Workshop on Cloud Computing Platforms - 2012*, volume 2, pages 1–6, Bern, Suisse, April 2012.

[128] Clément QUINTON, Daniel ROMERO, and Laurence DUCHIEN. Automated selection and configuration of cloud environments using software products lines principles. In *IEEE CLOUD*, 2014.

[129] Romain Rouvoy and Philippe Merle. Un langage de description et de vérification de motifs d'architecture : Fractal adl. In Isabelle Borne, Xavier Crégut, Sophie Ebersold, and Frédéric Migeon, editors, *LMO*, pages 49–64. Hermès Lavoisier, 2007.

[130] W. W. Royce. Managing the development of large software systems : Concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[131] Americo Sampaio and Nabor Mendonça. Uni4cloud : An approach based on open standards for deployment and management of multi-cloud applications. In *Proceedings of the 2Nd International Workshop on Software Engineering for Cloud Computing*, SECLOUD '11, pages 15–21, New York, NY, USA, 2011. ACM.

[132] P. Saripalli, G.V.R. Kiran, R.R. Shankar, H. Narware, and N. Bindal. Load prediction and hot spot detection models for autonomic cloud computing. In *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pages 397–402, Dec 2011.

[133] K. Sathiyamurthy, S. Anne Marie Sophia, A. Ashalatha, and P. Sujitha. Automated reasoning tool for the detection of race conditions in web services. In *Conference on Computational Intelligence and Multimedia Applications, 2007. International Conference on*, volume 2, pages 61–65, Dec 2007.

[134] Mina Sedaghat, Francisco Hernández-Rodriguez, and Erik Elmroth. A virtual machine re-packing approach to the horizontal vs. vertical elasticity trade-off for cloud autoscaling. In Hariri and Sill [71], page 6.

[135] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable sca applications with the frascati platform. In *IEEE SCC*, pages 268–275. IEEE Computer Society, 2009.

[136] Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni, and Jean-Bernard Stefani. A component-based middleware platform for reconfigurable service-oriented architectures. *Softw., Pract. Exper.*, 42(5) :559–583, 2012.

[137] Farhan Bashir Shaikh and Sajjad Haider. Security threats in cloud computing. In *ICITST*, pages 214–219. IEEE, 2011.

[138] Upendra Sharma, Prashant J. Shenoy, and Sambit Sahu. A flexible elastic control plane for private clouds. In Hariri and Sill [71], page 4.

[139] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale : Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 5 :1–5 :14, New York, NY, USA, 2011. ACM.

[140] Shi Shu, Xiang Shen, Yongxin Zhu, Tian Huang, Shunqing Yan, and Shiming Li. Prototyping efficient desktop-as-a-service for fpga based cloud computing architecture. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 702–709, June 2012.

[141] A Skendzic and B. Kovacic. Microsoft office 365 - cloud in business environment. In *MIPRO, 2012 Proceedings of the 35th International Convention*, pages 1434–1439, May 2012.

[142] Xiang Song, Jian Yang, and Haibo Chen. Architecting flash-based solid-state drive for high-performance i/o virtualization. *IEEE Computer Architecture Letters*, 99(RapidPosts) :1, 2013.

[143] Sean Stolberg. Enabling agile testing through continuous integration. In *Proceedings of the 2009 Agile Conference*, AGILE '09, pages 369–374, Washington, DC, USA, 2009. IEEE Computer Society.

[144] V. Subramanian, Hongyi Ma, Liqiang Wang, En-Jui Lee, and Po Chen. Rapid 3d seismic source inversion using windows azure and amazon ec2. In *Services (SERVICES), 2011 IEEE World Congress on*, pages 602–606, July 2011.

[145] K. Suttisirikul and P. Uthayopas. Accelerating the cloud backup using gpu based data deduplication. In *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, pages 766–769, Dec 2012.

[146] Clemens A. Szyperski. Component technology - what, where, and how ? In Lori A. Clarke, Laurie Dillon, and Walter F. Tichy, editors, *ICSE*, pages 684–693. IEEE Computer Society, 2003.

[147] A. Tahamtan, S.A. Beheshti, A. Anjomshoaa, and A.M. Tjoa. A cloud repository and discovery framework based on a unified business and cloud service ontology. In *Services (SERVICES), 2012 IEEE Eighth World Congress on*, pages 203–210, June 2012.

[148] Alain Tchana, Bruno Dillenseger, Noel De Palma, Xavier Etchevers, Jean-Marc Vincent, Nabila Salmi, and Ahmed Harbaoui. Self-scalable benchmarking as a service with automatic saturation detection. In David M. Eyers and Karsten Schwan, editors, *Middleware*, volume 8275 of *Lecture Notes in Computer Science*, pages 389–404. Springer, 2013.

[149] Alain Tchana, Daniel Hagimont, Laurent Broto, and Michel Daydé. Autonomic resource management in a Cloud infrastructure (regular paper). In *International Conference on Computer Science and Information Technology (CSIT), Yerevan, Armenia, 26/09/2011-30/09/2011*, page (electronic medium), http ://www.sci.am, septembre 2012. National Academy of Science of Armenia.

[150] Alain Tchana, Suzy Temate, Laurent Broto, and Daniel Hagimont. Autonomic resource allocation in a j2ee cluster. *Utility and Cloud Computing, India*, 2010.

[151] Alain Tchana, Suzy Temate, Laurent Broto, and Daniel Hagimont. TUNeEngine : An Adaptable Autonomic Administration System (regular paper). In *International conference on soft computing and software engineering, San Francisco, 01/03/2013-02/03/2013*, page (electronic medium). Journal of Soft Computing and Software Engineering, 2013.

[152] Alain-B Tchana. *Système d'administration autonome adaptable : application au Cloud*. PhD thesis, Ecole Doctorale Mathématiques, Informatique, Télécommunications (MITT), November 2011.

[153] Cheng Tian, Ying Wang, Feng Qi, and Bo Yin. Decision model for provisioning virtual resources in amazon ec2. In *Network and service management (cnsm), 2012 8th international conference and 2012 workshop on systems virtualiztion management (svm)*, pages 159–163, Oct 2012.

[154] S. Tilkov and S. Vinoski. Node.js : Using javascript to build high-performance network programs. *Internet Computing, IEEE*, 14(6) :80–83, Nov 2010.

[155] André Tost and José De Jesús. Scalability and elasticity for virtual application patterns in IBM PureApplication System. Technical report, IBM Corporation, 09 2013.

[156] Mahamadou Abdoulaye Toure. *Administration d'applications réparties à grande échelle*. PhD thesis, Ecole Doctorale Mathématiques, Informatique, Télécommunications (MITT), Juin 2010.

[157] Giang Son Tran, Alain Tchana, Laurent Broto, and Daniel Hagimont. Two-Level Cooperation in Autonomic Cloud Resource Management (regular paper). In *International Conference on Intelligent and Automation System (ICIAS) 2013, Ho Chi Minh City, Vietnam, 23/02/2013-24/02/2013*, page (on line), http ://www.joace.org/, février 2013. Journal of Automation and Control Engineering.

[158] B.G. Tudorica and C. Bucur. A comparison between several nosql databases with comments and notes. In *Roedunet International Conference (RoEduNet), 2011 10th*, pages 1–5, June 2011.

[159] Luis Miguel Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. Dynamically scaling applications in the cloud. *Computer Communication Review*, 41(1) :45–52, 2011.

[160] Nedeljko Vasić, Dejan Novaković, Svetozar Miučin, Dejan Kostić, and Ricardo Bianchini. Dejavu : Accelerating resource allocation in virtualized environments. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 423–436, New York, NY, USA, 2012. ACM.

[161] R.A Vyas, H.B. Prajapati, and V.K. Dabhi. Embedding custom metric in ganglia monitoring system. In *Advance Computing Conference (IACC), 2014 IEEE International*, pages 793–797, Feb 2014.

[162] B. Wadhwa and A Verma. Energy saving approaches for green cloud computing : A review. In *Engineering and Computational Sciences (RAECS), 2014 Recent Advances in*, pages 1–6, March 2014.

[163] Feng Wang, Jiangchuan Liu, and Minghua Chen. Calms : Cloud-assisted live media streaming for globalized demands with time/region diversities. In *INFOCOM, 2012 Proceedings IEEE*, pages 199–207, March 2012.

[164] Wenting Wang, Hao peng Chen, and Xi Chen. An availability-aware approach to resource placement of dynamic scaling in clouds. In Rong Chang, editor, *IEEE CLOUD*, pages 930–931. IEEE, 2012.

[165] J. Wei. How wearables intersect with the cloud and the internet of things : Considerations for the developers of wearables. *Consumer Electronics Magazine, IEEE*, 3(3) :53–56, July 2014.

[166] Jennifer Widom and S. J. Finkelstein. Set-oriented production rules in relational database systems. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pages 259–270, New York, NY, USA, 1990. ACM.

[167] Dutreilh Xavier, Kirgizov Sergey, Melekhova Olga, Malenfant Jacques, Rivierre Nicolas, and Truck Isis. Using reinforcement learning for autonomic resource allocation in clouds : Towards a fully automated workflow. In *Proceedings of the 2011 IARIA 7th International Conference on Autonomic and Autonomous Systems*, ICAS '11, pages 67–74, Washington, DC, USA, 2011. IEEE Computer Society.

[168] Yingyi Yang, Jin Yang, Fagui Liu, and Qi Duan. A cloud-based development platform for services and bundles of internet of things. In *Dependable, Autonomic and Secure Computing (DASC), 2013 IEEE 11th International Conference on*, pages 379–385, Dec 2013.

[169] L. Youseff, M. Butrico, and D. Da Silva. Toward a unified ontology of cloud computing. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1–10, Nov 2008.

[170] Jianjun Yu and Tongyu Zhu. Towards dynamic resource provisioning for traffic mining service cloud. In *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*, pages 1296–1301, Aug 2013.

[171] Hang Yuan, C.-C.J. Kuo, and I Ahmad. Energy efficiency in data centers and cloud-based multimedia services : An overview and future directions. In *Green Computing Conference, 2010 International*, pages 375–382, Aug 2010.

[172] M. Zhang, R. Ranjan, A. Haller, D. Georgakopoulos, and P. Strazdins. Investigating decision support techniques for automating cloud service selection. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 759–764, Dec 2012.

[173] Wenjuan Zhao and Kai Yang. Principles of content-distributed network and its applications. In *System Science, Engineering Design and Manufacturing Informatization (ICSEM), 2011 International Conference on*, volume 1, pages 229–232, Oct 2011.

[174] Jianlong Zhong and Bingsheng He. Towards gpu-accelerated large-scale graph processing in the cloud. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 1, pages 9–16, Dec 2013.

[175] Jiehan Zhou, T. Leppanen, E. Harjula, M. Ylianttila, T. Ojala, Chen Yu, Hai Jin, and L.T. Yang. Cloudthings : A common architecture for integrating the internet of things with cloud computing. In *Computer Supported Cooperative Work in Design (CSCWD), 2013 IEEE 17th International Conference on*, pages 651–657, June 2013.

Annexe : Exemple de déroulement de l'algorithme

Cette annexe vise à décrire le déroulement de l'algorithme de planification durant un scénario d'élasticité. Il s'agit d'expliquer l'enchaînement des différentes phases sur un cas non-trivial. Le scénario retenu est une opération de décroissance horizontale portant sur un Jonas au sein d'une application Springoo. La figure 2 illustre les modifications et révisions calculées sur le modèle par extension lors de l'exécution de l'algorithme. Le modèle par intention utilisé est celui donné par le listing 8.9. Pour faciliter la compréhension, celui-ci est rappelé dans le listing1.

Le déroulement observé est comme suit :

1. De l'étape 0 à l'étape 0.c, la séquence principale est débutée jusqu'à ce qu'une révision soit demandée en phase de gestion des contraintes de liaisons durant l'étape 0.b. Cette demande de révision porte le code 0x0001 et mentionne que le composant Apache :2 a une contrainte (décrite à la ligne 13 du modèle par intention) qui ne peut être satisfaite. Dans l'étape 0.c, le traitant de demande de révision correspondant au code 0x0001 établit une révision en fonction de la description de la révision mais également à partir du contexte : la dernière modification consistant à supprimer l'instance Jonas :4, l'instance Apache :2 est superflue et il faut la supprimer elle aussi.

2. De l'étape 0.d à l'étape 0.g, un nouveau déroulement de la séquence principale de l'algorithme de planification est exécuté. Dans l'étape 0.g, une révision est requise puisque deux VMs sont vides : cela viole la contrainte de la ligne 25 du modèle par intention. Une demande de révision portant sur ces conteneurs avec le code 0X1001 est donc retournée dans l'étape 0.f. Cette demande est gérée durant l'étape 0.g qui retourne une révision requérant la suppression des VMs vides.

3. De l'étape 0.h à 1, la séquence principale se déroule complètement sans demande de révision. Une nouvelle architecture-cible a été établie. Celle-ci respecte l'ensemble des contraintes du modèle par intention utilisé pour cet exemple.

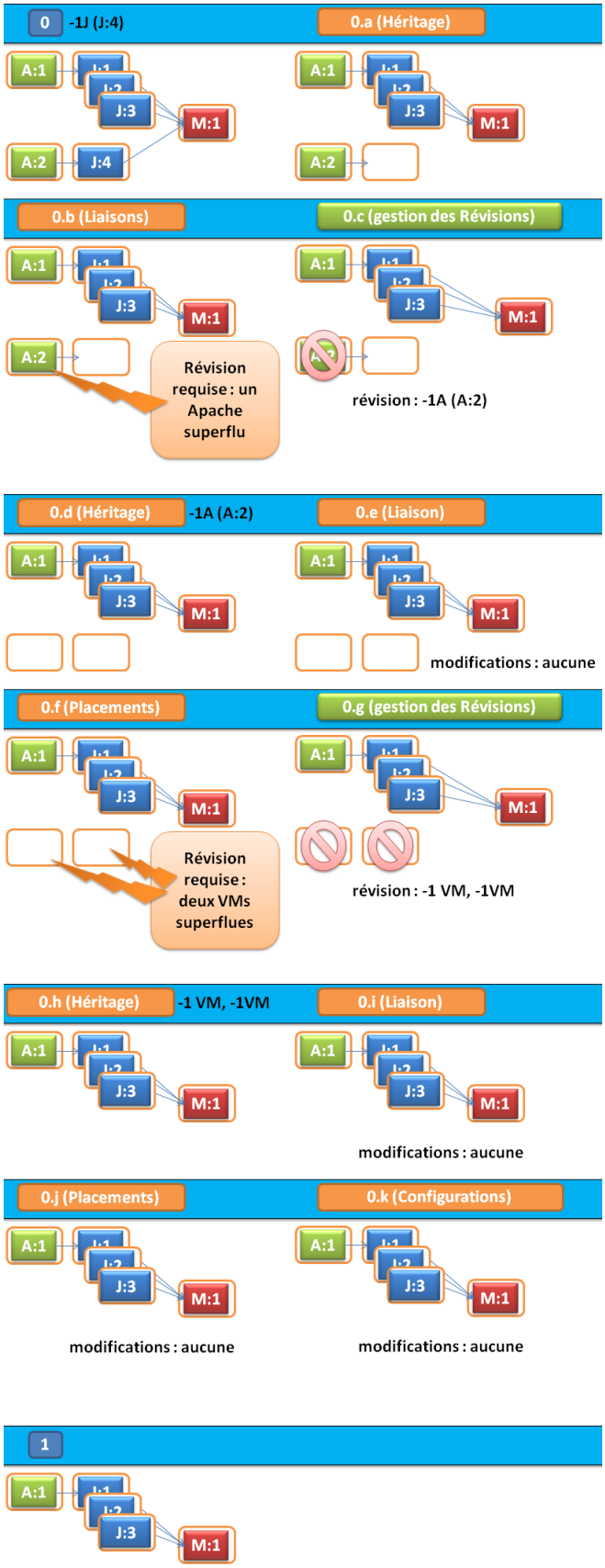


FIGURE 2 – Détail du déroulement de l'algorithme de planification pour un scénario de décroissance horizontale pour l'application Springoo

Listing 1 – Modèle par intention utilisé pour la réalisation du scénario illustré par la figure 2

```
<intensional-model>
        <components>
                <component name="Apache" init="1"/>
                <component name="Jonas" init="1"/>
                <component name="MySQL" init="1"/>
        </components>
        <containers>
                <container name="vm" init="0"/>
        </containers>

        <bindings>
                <binding-query>
                        local:bindExactlyOneAll("Apache","Jonas")
                </binding-query>
                <binding-query>
                        local:bindAllExactlyOne("Jonas","MySQL")
                </binding-query>
        </bindings>

        <placements>
                <placement-query>
                        local:placeOneOne()
                </placement-query>
                <placement-query>
                        local:purgeEmptyContainers("vm")
                </placement-query>
        </placements>

    <configurations>
                <configuration-query>
                        local:setPublicIpOn(("Apache"), "vm")
                </configuration-query>
    </configurations>
</intensional-model>
```