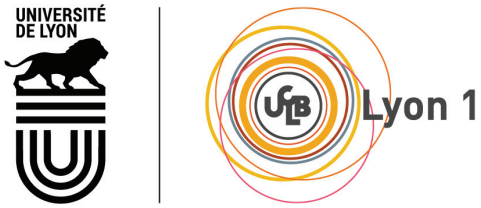
Méthodologie de provisionnement automatique d'applications métier orientées service sur les environnements cloud

Hind Benfenatki



N° d'ordre NNT : 2016LYSE1282

THÈSE DE DOCTORAT DE L'UNIVERSITÉ DE LYON
opérée au sein de
l'Université Claude Bernard Lyon 1

École Doctorale ED512
InfoMaths

Spécialité de doctorat : Informatique

Soutenue le 07/12/2016, par :
Hind BENFENATKI

Méthodologie de provisionnement automatique d'applications métier orientées service sur les environnements cloud

---

Devant le jury composé de :

M. Claude GODART, Professeur des universités, Université de Lorraine	Président du jury
Mme. Corine CAUVET, Professeur des universités, Université Aix-Marseille	Rapporteur
M. Khalil DRIRA, Directeur de recherche CNRS, Toulouse	Rapporteur
M. Ladjel BELLATRECHE, Professeur des universités, ENSMA Poitiers	Examinateur
M. Abder KOUKAM, Professeur des universités, UTBM Belfort-Montbéliard	Examinateur
Mme. Parisa GHODOUS, Professeur des universités, Université Lyon 1	Directrice de thèse
Mme. Aîcha-Nabila BENHARKAT, Maître de conférences, INSA de Lyon	Co-Directrice
Mme. Catarina FERREIRA DA SILVA, Maître de conférences, Université Lyon 1	Co-Directrice

UNIVERSITE CLAUDE BERNARD - LYON 1

Président de l'Université	M. le Professeur Frédéric FLEURY
Président du Conseil Académique	M. le Professeur Hamda BEN HADID
Vice-président du Conseil d'Administration	M. le Professeur Didier REVEL
Vice-président du Conseil Formation et Vie Universitaire	M. le Professeur Philippe CHEVALIER
Vice-président de la Commission Recherche	M. Fabrice VALLÉE
Directrice Générale des Services	Mme Dominique MARCHAND

COMPOSANTES SANTE

Faculté de Médecine Lyon Est – Claude Bernard	Directeur : M. le Professeur G.RODE
Faculté de Médecine et de Maïeutique Lyon Sud – Charles Mérieux	Directeur : Mme la Professeure C. BURILLON
Faculté d'Odontologie	Directeur : M. le Professeur D. BOURGEOIS
Institut des Sciences Pharmaceutiques et Biologiques	Directeur : Mme la Professeure C. VINCIGUERRA
Institut des Sciences et Techniques de la Réadaptation	Directeur : M. X. PERROT
Département de formation et Centre de Recherche en Biologie Humaine	Directeur : Mme la Professeure A-M. SCHOTT

COMPOSANTES ET DEPARTEMENTS DE SCIENCES ET TECHNOLOGIE

Faculté des Sciences et Technologies	Directeur : M. F. DE MARCHI
Département Biologie	Directeur : M. le Professeur F. THEVENARD
Département Chimie Biochimie	Directeur : Mme C. FELIX
Département GEP	Directeur : M. Hassan HAMMOURI
Département Informatique	Directeur : M. le Professeur S. AKKOUCHE
Département Mathématiques	Directeur : M. le Professeur G. TOMANOV
Département Mécanique	Directeur : M. le Professeur H. BEN HADID
Département Physique	Directeur : M. le Professeur J-C PLENET
UFR Sciences et Techniques des Activités Physiques et Sportives	Directeur : M. Y.VANPOULLE
Observatoire des Sciences de l'Univers de Lyon	Directeur : M. B. GUIDERDONI
Polytech Lyon	Directeur : M. le Professeur E.PERRIN
Ecole Supérieure de Chimie Physique Electronique	Directeur : M. G. PIGNAULT
Institut Universitaire de Technologie de Lyon 1	Directeur : M. le Professeur C. VITON
Ecole Supérieure du Professorat et de l'Education	Directeur : M. le Professeur A. MOUGNIOTTE
Institut de Science Financière et d'Assurances	Directeur : M. N. LEBOISNE

Remerciements

Mes sincères et chaleureux remerciements s'adressent en premier lieu et particulièrement à Mesdames : Parisa GHODOUS, ma Directrice de thèse, Aîcha Nabila BENHARKAT et Catarina FERREIRA DA SILVA, mes Co-directrices.

Pour la confiance qu'elles m'ont accordée en acceptant et en me faisant le grand honneur de diriger cette thèse. Je tiens notamment à leur exprimer mon admiration pour leur passion du travail, leurs disponibilité et écoute constantes, et pour tout le temps qu'elles m'ont consacré pour la réalisation de ce travail tout au long de ces quatre années de thèse. Leur aisance, leur rigueur scientifique et les précieux conseils prodigués m'ont été d'une grande aide. J'ai pris un grand plaisir à travailler à leurs côtés et à partager des moments aussi précieux que ceux passés lors des multiples réunions de travail. Je les remercie de m'avoir intégrée au sein de leur équipe. Qu'elles trouvent ici, l'expression de ma sincère reconnaissance et gratitude et de mon profond respect.

Je tiens à remercier les membres du jury de l'intérêt qu'ils ont porté à mon travail en acceptant de faire partie de ce jury.

Je remercie Mme. Corine CAUVET, Professeur à l'Université d'Aix-Marseille et M. Khalil DRIRA, Directeur de recherche CNRS à Toulouse, de me faire l'honneur d'être les rapporteurs de ce travail.

Je remercie également M. Ladjel BELLATRECHE, Professeur à l'ENSMA Poitiers, M. Claude GODART, Professeur à l'Université de Lorraine, et M. Abder KOUKAM, Professeur à l'UTBM Belfort-Montbéliard de me faire l'honneur d'être les examinateurs de ce travail.

J'adresse un grand merci à M. Zakaria MAAMAR avec qui j'ai eu la chance de travailler en collaboration et qui malgré son éloignement m'a si sympathiquement accompagnée par ses conseils éclairés et surtout par sa patience.

Je remercie également M. Okba KAZAR pour son implication et ses recommandations.

Mes remerciements à tous ceux et celles qui ont contribué de près ou de loin à la réalisation de ce travail.

A l'ensemble des membres de l'équipe Service Oriented Computing, pour la dy-

namique au sein du groupe que ce soit sur le plan scientifique ou sur le plan humain.

A toutes les personnes que j'ai eu la chance et le plaisir durant ces années de thèse de connaître et de côtoyer et avec qui j'ai partagé un bureau, un café ou un repas mais aussi d'agréables moments. Je pense notamment à Hanane, Sarah, Hamza, Malik, Kamel, Gavin, Cheikh, Hayri. . . .

A ma famille en témoignage de mon affection. Malgré mon éloignement, leur amour et leur confiance m'ont permis d'aller au bout de ma mission.

A ma très chère maman qui m'a inculqué les bonnes valeurs et l'envie de toujours aller au-delà des obstacles rencontrés. Son amour inconditionnel, sa constante présence et ses précieux conseils sont le fondement de ce travail. Merci d'avoir cru en moi.

A ma sœur Amina, qui m'a toujours été d'un soutien sans faille et d'une grande aide dans tout ce que j'entreprends malgré la distance qui nous sépare.

A mon papa chéri qui nous a quittées trop tôt et qui a fait en sorte que je puisse bénéficier d'une éducation remarquable et avec qui j'avais encore tant à partager, bien qu'il m'ait déjà tellement donné. Par ce travail, je poursuis ma progression sur le chemin vers lequel il m'a toujours guidée.

Abstract

Service-oriented computing and cloud computing offer many opportunities for developing and deploying applications. On the one hand, service-oriented computing allows to compose several functionalities from distributed services developed by different organizations. On the other hand, cloud computing allows to provision on demand scalable development and deployment environments. In this research work, we propose and describe a Methodology for AutomateD prOvisioning of cloud-based service-oriented busiNess Applications (MADONA). MADONA covers the whole application's lifecycle and is based on cloud orchestration tools that manage the deployment, the configuration, and the composition of business services.

Our objective is to propose a methodology to automatically provision service-oriented applications while reducing the necessary technical knowledge required from the user. To this end, we bring three major contributions. Firstly, our system automates the whole application provisioning. In fact, MADONA phases are fully automated. The user intervenes only in requirement elicitation and when the application is deployed and ready to use. The business application is automatically generated by composing business services, and deployed in an automatically preselected IaaS.

Secondly, we enrich the description of services by integrating concepts describing services' interactions. Service description languages usually describe services as isolated components and do not consider the interactions between services. We define composition interactions which describe for each business service, its necessary services and the services with which it can be composed.

Thirdly, we allow the user to express her requirements abstracting composition and deployment technical details. To this end, we define a RequIrement VocAbuLary (RIVAL) to formalize user's functional and non-functional requirements.

The methodology has been implemented and evaluated qualitatively and quantitatively following several scenarios showing its faisability.

Keywords : Cloud computing - Linked services - Linked USDL - Provisioning of service-orientd business applications - Service description

Résumé

Le développement orienté service et le cloud computing offrent plusieurs opportunités au développement et au déploiement d'applications. D'un côté, le développement orienté service permet de composer des fonctionnalités issues de services distribués, développés par différentes organisations. D'un autre côté, le cloud computing permet de provisionner des environnements évolutifs de développement et de déploiement, à la demande. Dans ce travail de recherche, nous proposons et décrivons une méthodologie de provisonnement automatique d'applications métier, orientées service sur le cloud. Nous avons appelé cette méthodologie MADONA (Methodology for AutomateD prOvisioning of cloud-based service-oriented busiNess Applications). MADONA couvre tout le cycle de vie de provisionnement d'applications. Elle est basée sur un orchestrateur de services pour la gestion de la configuration, du déploiement, et de la composition de services métier.

Notre objectif est de réduire les connaissances techniques nécessaires de l'utilisateur pour le provisionnement d'applications métier. Pour ce faire, nous apportons trois contributions majeures. Premièrement, nous automatisons ce provisionnement. En effet, les phases de MADONA sont complètement automatisées. L'utilisateur n'intervient que pour exprimer son besoin et pour utiliser l'application métier automatiquement générée par la composition de services métier, et déployée sur une IaaS présélectionnée.

Deuxièmement, nous enrichissons la description des services par des concepts liés aux relations d'un service. Les langages de description de services décrivent le plus souvent ces derniers comme des entités isolées et ne considèrent pas les relations entre services. Nous avons défini les relations de composition qui décrivent pour chaque service métier les services nécessaires à son bon fonctionnement, et les services avec lesquels il peut être composé.

Troisièmement, nous permettons à l'utilisateur d'exprimer son besoin à un haut niveau d'abstraction des détails techniques de composition et de déploiement. Nous avons pour cela défini un vocabulaire pour formaliser ces besoins fonctionnels et non fonctionnels.

La méthodologie a été prototypée et évaluée qualitativement et quantitativement suivant plusieurs scenarii montrant sa faisabilité.

Mots clés : Cloud computing - Linked services - Linked USDL - Provisionnement

d'applications métier orientées service - Description des services

Table des matières

Liste des figures

Liste des Tableaux

Chapitre 1

Introduction

Sommaire

Ce chapitre présente une introduction générale du travail de thèse. Il décrit le contexte, les problématiques traitées, et les contributions scientifiques.

1.1 Contexte du travail de recherche

Le cloud computing a émergé comme une nouvelle façon d'accéder, à la demande, à des ressources informatiques (logiciels, plateformes, et infrastructures) en tant que services fournis sur Internet. Aujourd'hui, le marché du cloud computing français est estimé à 5,9 milliards d'euros selon une étude faite par Markess[1] [mar]. Ce chiffre d'affaire a été multiplié par 6 en dix ans (2007-2016). Selon cette étude, 57% des services cloud utilisés sont des SaaS, contre 8% de PaaS et 35% d'IaaS.

Dans les environnements cloud, la gestion des ressources informatiques est automatisée côté fournisseur et donc dissimulée au client. Kevin Marks de Google [gooa] disait en parlant d'Internet "*Nous ne nous soucions pas de la destination des messages...le nuage nous dissimulait cette information*". Cela est encore d'actualité pour le cloud. En effet, les responsabilités du client et donc les tâches qui lui sont permises diffèrent en fonction du service utilisé. Cela restreint le contrôle que l'on a sur l'application développée, mais (i) implique une accessibilité et une simplicité d'utilisation des Technologies de l'Information (TI), et (ii) réduit le temps et la complexité de mise en place d'une application.

Le développement orienté service, par la composition automatique de services Web, a permis d'automatiser les interactions entre services distribués [Mri07], et de construire des applications à moindre coût tout en diminuant les connaissances métier requises. En effet, la composition de services permet de sous-traiter une tâche à des composants logiciels développés à cet effet. A titre d'exemple, une entreprise dont le cœur de métier est le consulting informatique peut sous-traiter la gestion des salaires à un service de gestion des salaires. Cela permet de faire collaborer des individus détenant des savoir-faire différents et des connaissances diverses pour la réalisation d'un objectif global.

Les Architectures Orientées Service (AOS) et le cloud computing constituent une bonne combinaison pour le provisionnement (développement et déploiement) d'applications métier. D'un côté, AOS offre une façon de réduire les connaissances techniques et métier par la composition de services existants. D'un autre côté, le cloud computing provisionne de manière élastique et à la demande, des environnements de déploiement pour ces applications.

Le travail de recherche de cette thèse traite de l'automatisation du provisionnement d'applications métier dans les environnements cloud. Nous considérons qu'une application métier est une application destinée à être utilisée par des intervenants métier dans le but d'effectuer des tâches aidant au bon fonctionnement d'une entreprise.

[1] Etude présentée le 6 juillet 2016 lors de la Cloud Week de Paris. Cette étude a été effectuée auprès de 145 prestataires.

Les applications métier font collaborer plusieurs composants indépendants qui répondent aux différentes fonctionnalités requises. Nous faisons référence à ces composants comme des services métier. Ces derniers, sont de plus grande granularité que les services Web. Ils représentent des packages applicatifs qui peuvent être déployés sur plusieurs infrastructures. Une plus grande granularité réduit le nombre de services à manipuler [Lüf14] lors du développement d'applications avec beaucoup de fonctionnalités.

Les services métier sont différents des services Web. En effet, un service Web assure une fonctionnalité particulière appliquée sur des entrées. Un service métier assure des fonctionnalités plus globales. A titre d'exemple, un service Web peut assurer la correction d'orthographe d'un texte en entrée, alors que, dans le même contexte, un service métier assurera une fonctionnalité de mailing, i.e., correction orthographique de mail, envoi/réception de mails, archivage etc. Un service métier peut être implémenté par l'invocation de plusieurs services Web.

L'objectif de ce travail est de réduire les connaissances techniques de l'utilisateur ainsi que l'intervention humaine nécessaires pour le provisionnement d'applications métier. L'utilisateur pourra exprimer son besoin fonctionnel et non fonctionnel à un haut niveau d'abstraction des détails techniques, puis, une application composite sera générée et déployée automatiquement sur un environnement cloud.

1.2 Problématiques et contributions

Les objectifs que nous visons, soulèvent trois problématiques majeures qui se déclinent sur les trois axes suivants :

Problématique 1 : L'objectif de ce travail étant de minimiser les connaissances techniques requises de l'utilisateur pour le provisionnement d'applications, la première question que nous nous sommes posée est : quels sont les besoins fonctionnels et non fonctionnels non techniques à considérer pour le provisionnement d'applications cloud orientées service? Comment modéliser ou spécifier ces besoins?

Contribution 1 : Pour répondre à cette problématique, nous avons défini le vocabulaire "RequIrement VocAbuLary" (RIVAL), pour formaliser la description des besoins fonctionnels et non fonctionnels de l'utilisateur pour l'application souhaitée. RIVAL permet de décrire les fonctionnalités désirées par l'utilisateur, ses contraintes de qualité de services, et ses préférences de déploiement et de coût. Les besoins de l'utilisateur sont exprimés à un haut niveau d'abstraction se distançant ainsi des détails techniques de composition et de déploiement.

Problématique 2 : L'étude de la composablité d'un service métier nécessite la connaissance des relations potentielles du service. Même si certains langages de description de

services permettent de décrire certaines relations de services, nous n'avons pas trouvé de langage qui permette de décrire en même temps les relations qu'un service *peut* et celles qu'il *doit* avoir avec les autres services. Cela nous a amené à nous demander comment adapter et/ou étendre les langages de description de services afin de permettre la sélection et la composition automatiques et dynamiques des services métier.

Contribution 2 : Nous avons étendu le langage Linked Unified Service Description Language (Linked USDL) [linb] pour la description des services. Notre choix pour ce langage a été guidé par le fait que ce dernier permet de couvrir les aspects métier, techniques et opérationnels dans la description de services. De plus, ce langage repose sur les principes du Linked Data [lina]. Il permet de combiner des ontologies et vocabulaires existants, ce qui rend simples son extension et adaptation. Cependant, Linked USDL ne décrit pas les relations qu'un service *peut* et/ou *doit* avoir avec un autre. Notre challenge était donc d'étendre ce langage afin de décrire ces dernières.

Problématique 3 : Comment développer et déployer une application métier orientée service dans des environnements cloud avec un minimum d'intervention humaine et de connaissances techniques?

Contribution 3 : Nous proposons une méthodologie qui illustre les différentes phases de provisionnement automatique d'applications cloud, celle-ci s'intitule "Methodology for AutomateD prOvisioning of cloud-based service-oriented busiNess Applications" (MADONA). L'intervention humaine ainsi que les connaissances techniques sont réduites puisque les phases de MADONA sont automatisées. En effet, l'utilisateur intervient seulement lors de l'élicitation des besoins et lorsque l'application est déployée et prête à être utilisée. MADONA prône une collaboration implicite par la composition et la réutilisation de services métier. A partir des besoins de l'utilisateur, MADONA génère des plans de composition de l'application composite à déployer sur le cloud. Un plan de composition représente une application abstraite qui compose des services métier et leur assigne une IaaS pour leur déploiement. Chaque service métier impliqué dans un plan de composition est automatiquement composé avec les services avec lesquels il doit l'être. De plus, deux services métier ne sont composés que si une relation de composition est possible entre les deux, cette relation étant connue depuis la description de chaque service métier. Lorsqu'aucun service ne répond aux besoins fonctionnels de l'utilisateur, ce dernier peut intégrer de nouveaux services métier à la marketplace. Nous entendons par marketplace un dépôt de services métier et IaaS.

Contribution 4 : Nous avons développé un prototype de notre approche méthodologique pour vérifier son applicabilité. Ce prototype a été testé suivant différents scenarii et évalué qualitativement et quantitativement.

1.3 Organisation du manuscrit

Ce manuscrit est composé de huit chapitres répartis sur deux parties. La première partie présente un état de l'art sur le développement d'applications cloud orientées service. Cette partie est composée de deux chapitres :

- Dans le Chapitre 2, nous étudions et analysons les différents travaux de l'état de l'art relatifs aux (i) langages de description des services, et (ii) aux langages de description des besoins de l'utilisateur en vue du développement d'applications.
- Dans le Chapitre 3, nous étudions et analysons les différents travaux de l'état de l'art relatifs aux approches et environnements de développement et de déploiement d'applications cloud.

La deuxième partie décrit les contributions de cette thèse. Elle est composée de quatre chapitres :

- Le Chapitre 4 décrit l'extension du langage Linked USDL pour la description des services de la marketplace. Cette extension vise à décrire entre autres les relations d'un service métier permettant ainsi d'automatiser sa composition. Ce chapitre décrit aussi RIVAL, le vocabulaire permettant de décrire les besoins de l'utilisateur en vue du provisionnement d'applications cloud. Un scénario illustratif est décrit dans ce chapitre. Il permet d'illustrer la description des services de la marketplace, les besoins de l'utilisateur, et les phases de MADONA (Chapitre 5).
- Le Chapitre 5 décrit MADONA, notre méthodologie de provisionnement d'applications métier qui compose automatiquement des services métier et déploie l'application générée sur le cloud.
- Le Chapitre 6 décrit la mise en œuvre de notre approche méthodologique sous forme d'un environnement de provisionnement d'applications métier orientées service sur le cloud. Cet environnement a été prototypé. Son implémentation ainsi que les choix technologiques sont décrits dans ce chapitre.
- Le Chapitre 7 décrit les tests effectués afin d'évaluer notre travail.

Enfin, le Chapitre 8 conclut ce travail de thèse et présente des perspectives.

Partie I

État de l'art

Chapitre 2

Langages de description de services et spécification d'applications orientées services

Sommaire

2.1 Introduction

Nous allons introduire dans ce chapitre les concepts utilisés dans ce travail de recherche. Nous identifions ensuite les différents travaux traitant :

(i) Des langages et des ontologies de description de services. Plusieurs travaux se sont penchés sur la description de différents types de services, à savoir les services Web, les services métier, et les services cloud. Nous avons classifié ces travaux en fonction de leur description des relations d'un service comme suit :

- La description des services comme des entités isolées. Nous avons distingué dans ces travaux, ceux décrivant des services Web et ceux décrivant des services cloud. Chaque travail est analysé en fonction du type de description offerte (ex : technique, sémantique, non fonctionnelle).
- La description des services et de leurs interactions, c'est à dire, les informations relatives à la composition d'un service avec un autre. Nous distinguons deux catégories dans ces travaux : (i) ceux décrivant **les relations établies** entre services (ayant existé dans le passé), et (ii) ceux décrivant **les relations potentielles** d'un service.

(ii) De l'élicitation des besoins pour le développement d'applications. La description des besoins lors du développement d'applications diffère en fonction de la méthode de développement utilisée. En effet, dans un développement en cascade, tous les besoins sont identifiés au début du cycle de vie et documentés sous forme de spécifications, alors que dans une méthode agile, les besoins évoluent au fur et à mesure des itérations.

Nous avons fait la distinction dans ce chapitre entre l'élicitation des besoins lors d'un développement classique et l'élicitation des besoins lors du développement d'applications orientées service.

Enfin, nous analysons et discutons les différents travaux de l'état de l'art afin de positionner notre travail par rapport aux solutions existantes.

2.2 Définitions des concepts utilisés

Dans cette section, nous introduisons notre vision des concepts de services que nous classifions. Nous introduisons également la composition de services métier et le cloud computing, concepts constituant le cadre de notre recherche.

2.2.1 Les services

L'INSEE (Institut National de la Statistique et des Etudes Economiques) définit un service comme étant la mise à disposition d'une capacité technique ou intellectuelle [ins]. Dans cette section, nous classifions les différents types de service, puis nous portons une attention particulière aux services métier considérés dans ce travail. La classification que nous proposons se fait en fonction des types de services et de leur mode de déploiement.

2.2.1.1 Selon le type de service

Les services peuvent être distingués en fonction de leur type. Lemos et al. [LDB15] classifient les services en trois catégories comme suit :

- Services de données : ce sont des sources de données sur le Web qui sont représentées sous format RDF (Resource Description Framework) [rdf]. Ce dernier permet une description sémantique des ressources Web et de leurs méta-données sous la forme de triplets : ressource, propriété, et valeur. Ces sources de données peuvent être agrégées pour répondre à une requête complexe.

- Services ayant une logique applicative : ces derniers fournissent des fonctionnalités métier à d'autres applications [SK03] (ex : service de paiement en ligne).

- Interface utilisateur : représente les widgets utilisés lors de la création d'une interface. Ces widgets permettent non seulement la génération de l'interface désirée mais génèrent aussi le code pour le traitement de la tâche qui lui est associée comme c'est le cas pour un widget de login ou de gestion des maps.

Dans le cadre de la définition du langage de description de services Linked Unified Service Description Language (Linked USDL) [linb], quatre types de services sont considérés comme suit :

- Les services humains qui représentent des services visant à identifier les besoins humains et améliorer la prestation de services et la qualité de vie des populations [Wha] (ex : service de conseil).

- Les services métier : Luftengger [Lüf14] définit les services métier comme *"des fonctionnalités métier encapsulées qui apportent de la valeur au consommateur. L'interaction entre le service métier et le consommateur est effectuée par* ***des entrées/sorties bien définies****"*. Selon Rittgen [Rito6], *"un service métier* ***agrège un ensemble d'opérations utilisées par des processus métier****, et fournit ainsi une interface entre les systèmes métier et les systèmes d'information"*. La majorité des services métier sont de plus grande granularité qu'un service logiciel [Lüf14] (ex : réservation de voyage pour un service métier, réservation de vol pour un service logiciel).

- Les services logiciels comme les services Web et RESTful.

- Les services infrastructures comme les services de stockage.

Les services infrastructures, logiciels, métier, et humains peuvent être implémentés par l'invocation de plusieurs services Web.

2.2.1.2 Selon le mode de déploiement du service

Les services peuvent être classés en deux catégories en fonction de leur mode de déploiement :

- Les services déployés chez le fournisseur et qui sont directement invoqués par des programmes comme c'est le cas pour les services Web, ou directement utilisables par l'utilisateur comme c'est le cas pour les SaaS (Software as a Service).

- Les services packagés et prêts à être déployés sur différentes infrastructures notamment les infrastructures cloud. TOSCA (Topology and Orchestration Specification for Cloud Applications) [BBKL14] et SOCCA (Service-Oriented Cloud Computing Architecture) [TSB10] permettent respectivement de déployer et de construire les applications SaaS par l'assemblage de ce type de services, qui contrairement à ceux définis dans les AOS traditionnelles, représentent des packages qui peuvent être déployés sur différents clouds.

Nous considérons dans ce travail les services métier comme étant les briques de base constituant une application métier. Nous les définissons comme suit : les services métier représentent des packages applicatifs, fournissant des fonctionnalités métier, qui sont développés par des tiers. Ils sont stockés sur différentes plateformes, sont déployables sur différentes infrastructures cloud, et composables avec d'autres services. Leurs déploiement et composition se font à l'aide de scripts. Les modules d'un ERP (Enterprise Ressource Planning) sont des exemples de services métier.

2.2.2 Composition de services Web et de services métier

La composition automatique de services consiste à combiner et à lier plusieurs services atomiques faiblement couplés afin de construire un service composite à plus forte valeur. Chaque service peut ainsi servir plusieurs applications [why].

Selon Lemos et al. [LDB15] les éléments principaux à considérer lors de la composition de services Web sont les suivants :

- Accès au service : définit les mécanismes permettant d'invoquer des opérations, d'envoyer des notifications et d'intercepter des évènements.

- Gestion de la conversation : définit le protocole d'ordre d'utilisation des opérations d'un service. A titre d'exemple, pour un service de paiement, les opérations peuvent suivre l'ordre suivant : l'authentification, l'initiation du processus de paiement, et l'attente de confirmation.

- Flux de contrôle : ordonne les activités de composition (séquentiel, parallèle...).

- Flux de données : décrit la propagation des données entre les différents services inclus dans une composition.

- Transformation des données : consiste à rendre compatibles les données propagées entre services, i.e., les sorties d'un service A compatibles avec le format des entrées du service B. Ce processus se fait en changeant le format, en fragmentant, ou encore en fusionnant les données afin qu'elles puissent être utilisées comme entrées pour le service B.

La composition de services métier que nous considérons est différente de la composition des services Web. En effet, elle ne représente pas un processus métier et ne se fait donc pas par la gestion des flux de données et de contrôle entre les services. Composer des services métier revient à configurer un service de façon à ce qu'il puisse fonctionner en collaboration avec un autre service. A titre d'exemple, composer une base de données avec un moteur de wiki revient à construire les tables et les relations de la base de données afin qu'elles correspondent aux besoins du moteur de Wiki, puis de rendre la base de données accessible par le moteur de Wiki par la communication de son adresse IP. De ce fait, il n'est pas possible d'évaluer la composabilité de deux services métier de la même façon que pour les services Web (par le matching des entrées/sorties). Afin de connaître la composabilité d'un service métier, la connaissance des services avec lesquels il doit être composé, et ceux avec lesquels il peut l'être est nécessaire.

2.2.3 Cloud computing

Le cloud offre plusieurs avantages au développement d'applications [CVT10], notamment, parce qu'il offre des services ergonomiques et simples d'utilisation, permet une allocation élastique des ressources, diminue le coût d'exploitation et de mise en œuvre de nouvelles applications, et facilite la collaboration à moindre coût. Plusieurs définitions du cloud computing existent dans la littérature. Nous retenons celle proposée par le "National Institute of Standards and Technology" (NIST) [MG11], qui stipule que *"le cloud computing est un modèle permettant d'accéder au travers du réseau, et à la demande à un ensemble de ressources informatiques partagées et configurables (exemples : réseaux, serveurs, stockage, applications, et services), qui sont rapidement mobilisables et libérables* ***avec un effort minimal d'administration ou d'intervention du fournisseur****"*.

Selon le NIST [MG11], le cloud computing est distingué par cinq caractéristiques, trois modèles de service, et quatre modèles de déploiement.

2.2.3.1 Caractéristiques essentielles

Les cinq caractéristiques essentielles qui différencient le cloud computing des autres environnements de calcul distribués sont les suivantes [MG11] :

1. Accès libre-service et à la demande : le provisionnement en ressources se fait sans interaction avec le fournisseur de service.

2. Accès ubiquitaire au réseau : les ressources sont accessibles sur le réseau depuis des plateformes hétérogènes (tablettes, stations de travail, smartphones, etc).

3. Mise en commun des ressources : les ressources sont mises en commun et assignées aux différents clients de manière dynamique suivant le modèle multi-tenants[1]. Le client n'a pas le contrôle sur l'emplacement exact des ressources fournies. Cependant, ce dernier peut spécifier l'emplacement souhaité pour ses ressources à un niveau plus élevé d'abstraction tel que le pays ou le continent.

4. Elasticité rapide : les ressources fournies sont élastiques, i.e., s'adaptent aux besoins évolutifs (croissants ou décroissants) de l'utilisateur. Les capacités de calcul semblent être illimitées.

5. Service mesuré : les services consommés sont mesurés en fonction du type de service (le stockage, la bande passante...). Cette mesure est gérée de manière automatique côté fournisseur et rapportée à l'utilisateur afin de lui permettre de contrôler et monitorer sa consommation.

2.2.3.2 Modèles de services

Les services cloud ont été classifiés par le NIST en trois modèles en fonction de la nature du service, à savoir logiciel, plateforme ou infrastructure [MG11].

1. SaaS (Software as a Service) : le modèle SaaS permet à l'utilisateur d'utiliser une application logicielle développée, déployée sur une infrastructure cloud et gérée par le fournisseur. La gestion de l'infrastructure sous-jacente à l'application tels que le réseau et les serveurs, est totalement abstraite pour l'utilisateur. Ce dernier peut néanmoins configurer ou personnaliser l'application. Beaucoup de logiciels sont aujourd'hui proposés en tant que service, comme les messageries en ligne, les gestionnaires de la relation client "Customer Relationship Management" (CRM), les logiciels de stockage comme flickr, etc.

2. PaaS (Platform as a Service) : le modèle PaaS permet à l'utilisateur de développer et déployer des applications sur l'infrastructure cloud du fournisseur, en utilisant des langages de programmation et des outils supportés par ce dernier. L'utilisateur ne gère pas et ne contrôle pas l'infrastructure sous-jacente à l'application, mais exerce un contrôle sur les applications déployées. Google App Engine [gooc] et CloudFoundry [clob] sont des exemples de PaaS.

[1] L'architecture multi-tenants est une architecture logicielle qui permet à plusieurs tenants (locataires) d'utiliser une seule instance de logiciel, et cela par l'accommodation de leurs besoins au travers de configuration [ACK12]

3. IaaS (Infrastructure as a Service) : le modèle IaaS permet à l'utilisateur d'allouer des capacités de calcul en vue du déploiement et de l'exécution d'applications. L'utilisateur ne gère pas et ne contrôle pas l'infrastructure sous-jacente mais exerce un contrôle sur les systèmes d'exploitation, le stockage, et les applications déployées. Amazon EC2 [amab] et Windows Azure [win] sont des exemples d'IaaS.

Aujourd'hui, nous parlons de tout comme service "Everything as a Service" (XaaS). Nous pouvons alors considérer d'autres modèles de services cloud, tels que le "Data as a Service" et le "Business as a Service" qui permettent respectivement d'externaliser la gestion des données et des processus métier d'une entreprise.

2.2.3.3 Modèles de déploiement

Les services cloud peuvent être déployés suivant quatre modèles [MG11] :

1. Privé : *"l'infrastructure cloud est fournie à l'usage exclusif d'une seule organisation. Elle peut être détenue, gérée et exploitée par l'organisation, un tiers, ou une combinaison d'entre eux. Elle peut exister au sein des locaux de l'organisation ou en dehors"* [MG11].
2. Communautaire : *"l'infrastructure cloud est fournie à l'usage exclusif d'une communauté spécifique d'utilisateurs d'organisations qui se partagent les mêmes préoccupations. Elle peut être détenue, gérée, et exploitée par un ou plusieurs des organismes de la communauté, un tiers, ou une combinaison d'entre eux. Elle peut exister au sein ou en dehors des locaux des différents organismes de la communauté"* [MG11].
3. Public : *"l'infrastructure cloud est fournie pour une utilisation ouverte pour le grand public. Elle peut être détenue, gérée et exploitée par une entreprise, une institution académique, un organisme gouvernemental, ou une combinaison d'entre eux. Elle existe dans les locaux du fournisseur cloud"* [MG11].
4. Hybride : *"l'infrastructure cloud est une composition de deux ou plusieurs infrastructures cloud distinctes (privé, communautaire, ou public). Celles-ci demeurent des entités uniques, mais sont liées par des technologies standardisées ou propriétaires qui permettent la portabilité des données et des applications entre les différentes infrastructures cloud* [MG11].

2.3 Langages et ontologies de description des services

Les travaux traitant de la description de services décrivent, pour la plupart, les services comme des entités isolées. Ils sont décrits dans la Section 2.3.1. Cependant, des travaux existent dans la littérature permettant de décrire les interactions que peut avoir un service avec les autres. Ils sont décrits en Section 2.3.2.

2.3.1 Description des services comme des entités isolées

Les paramètres considérés dans la description des services diffèrent en fonction du type de service décrit. Dans ce qui suit, nous faisons la distinction entre les langages et les ontologies de description des services Web et des services cloud.

2.3.1.1 Description des services Web

Plusieurs standards ont été adoptés pour la description et la publication des services Web tels que le langage Web Service Description Language (WSDL) [wsd] et l'annuaire Universal Description Discovery and Integration (UDDI) [udd]. Les langages de description de services tel que WSDL se concentrent sur la description non sémantique de l'aspect interface des composants logiciels [CDKB11] au détriment des aspects non fonctionnels. D'Ambrogio [D'A06] a étendu le langage WSDL en définissant un méta-modèle intégrant les caractéristiques QoS (Quality of Service) dans la description des services. Becha et al. [BA14], définissent un catalogue de paramètres non fonctionnels pour la composition de services Web, ainsi que les algorithmes permettant d'agréger ces services en prédisant leurs paramètres non fonctionnels. La valeur de chaque paramètre non fonctionnel est calculée comme une probabilité. Par exemple, la disponibilité d'un service est calculée comme la probabilité que ce service réponde en un temps satisfaisant sur une période donnée. Le temps de réponse est répertorié en trois catégories (bon, moyen, mauvais) et à chaque catégorie est associée une probabilité d'exécution du service sur un certain nombre d'unités de temps.

Plusieurs langages et ontologies ont été proposés pour enrichir sémantiquement la description d'un service Web. Selon [Mri07], et [Ald11], nous pouvons distinguer deux approches :

- La définition de nouvelles ontologies pour la description des services Web, tel que : "Web Ontology Language for Web services" (OWL-S) [MPM+04], qui a été construit sur la base du langage "Web Ontology Language" (OWL) [WMS04]. Ce dernier étend RDF par la définition de notions telles que les propriétés d'équivalence, de contraire, et d'identiques, entre deux ressources. OWL-S permet de décrire ce que fait le service Web, comment l'utiliser, et comment y accéder sur la base de trois éléments :

  - Le "Service Profile" décrit les fonctionnalités d'un service Web.
  - Le "Process Model" décrit l'interaction du consommateur du service avec le service décrit en termes des entrées/sorties du service, ses pré-conditions, et

ses résultats.

- Le "Service Grounding" décrit les détails techniques de l'invocation du service, en termes de protocole de communication, des ports utilisés, et de l'encodage des données.

- L'incorporation d'annotations au langage de description afin de permettre une description sémantique d'un service. Le langage "Semantic Annotations for WSDL" (SAWSDL) [saw] permet d'annoter les descriptions WSDL en vue de permettre une description sémantique de ses éléments. Cette annotation permet de lier une description d'un concept d'une ontologie à un élément WSDL. SAWSDL décrit comment une annotation doit être réalisée sans pour autant imposer le choix de l'ontologie de description à utiliser. Pour annoter une description WSDL, SAWSDL a défini trois attributs :

  - L'attribut "modelReference" permet d'associer un ou plusieurs concepts d'une ontologie métier à un concept du langage WSDL.
  - Les attributs "loweringSchemaMapping" et "liftingSchemaMapping" de SAWSDL spécifient le mapping des données sémantiques de l'utilisateur vers un fichier XML et inversement.

2.3.1.2 Description des services cloud

Certains travaux [AMBT14], [HS10], [KS11], [TBAT12], portent sur la structuration du cloud computing sous forme d'ontologie afin de satisfaire la découverte de services cloud. Ils définissent des ontologies et taxonomies pour décrire les fonctionnalités de ces derniers de façon détaillée (ex : Amazon EC2 est une IaaS qui offre du réseau, de la mémoire etc). Cela permet une découverte et une sélection précises. Cependant, cela contraint l'utilisateur à avoir des connaissances techniques sur les services manipulés. En l'occurrence, l'utilisateur doit connaître le type d'instance, et/ou les ressources nécessaires dans le cas d'un service IaaS.

Liu et al. [LZ11] proposent Cloud#, un langage permettant de modéliser l'organisation interne d'une IaaS afin de montrer aux utilisateurs comment les services sont fournis et les applications déployées (allocation des ressources, scalabilité, etc). Le but de ce langage est d'augmenter la transparence sur le fonctionnement des services cloud, et la confiance des utilisateurs des services cloud en leurs fournisseurs. Les fournisseurs utilisent Cloud# pour modéliser formellement leurs services en décrivant le comportement de gestion de l'IaaS (ex : comment l'IaaS gère la scalabilité des ressources,

description du mécanisme de monitoring, etc). De ce fait, à partir de ces modèles, les clients peuvent par exemple, vérifier si les ressources de leurs applications sont allouées équitablement et si leurs données sont isolées correctement.

Taher et al. [TNL+12], définissent des langages pour la description des XaaS sous forme de blueprints (Description abstraite d'une offre de service cloud) comme suit :

- Blueprint Definition Language (BDL) : décrit les aspects opérationnels (ex : type de service, interfaces et opérations), les capacités de performance (ex : temps de réponse minimal et maximal), les ressources nécessaires en fonction du profil de charge du service, et les politiques permettant de régir les conditions qu'émet le consommateur du service à l'utilisation de ce dernier.

- Blueprint Constraint Language (BCL) : décrit formellement les différents types de politiques capturées par BDL, tels que les termes SLA (Service Legal Agreement), les contraintes de déploiement, de localisation des données, de sécurité, d'auditabilité, etc.

- Blueprint Manipulation Language (BML) : fournit un ensemble d'opérateurs pour manipuler, comparer, et accomplir un mapping entre blueprints qui sont définis dans BDL.

BDL et BCL permettent la description des ressources nécessaires à un service et ses contraintes de déploiement permettant ainsi d'automatiser son déploiement, mais ne décrivent pas les relations de composition des services, i.e, les services avec lesquels le service décrit peut et/ou doit être composé.

Dans le cadre de l'étude de l'intégration de SaaS avec d'autres applications, Sun et al. [SZC+07] définissent SaaS Description Language (SaaS DL) pour décrire les services SaaS. Les auteurs considèrent qu'un SaaS peut être considéré comme un service Web complexe. Ils reposent alors sur WSDL pour la description des interfaces de programmation des services SaaS. Ils étendent WSDL pour modéliser les caractéristiques des SaaS en intégrant trois nouveaux concepts :

- Politique de personnalisation : un service SaaS étant le plus souvent multi-tenants, il s'appuie sur la personnalisation par la configuration pour garantir une application qui correspond aux besoins de chaque client. Les politiques de personnalisation décrites par le fournisseur de SaaS témoignent de la capacité de personnalisation du service.

- Politique de facturation : en plus des paramètres de facturation liés aux services Web, l'usage de stockage et les types de souscription aux SaaS doivent être con-

sidérés. Un modèle est proposé pour organiser les items de facturation (décrit une règle de facturation qui peut être propre à chaque fournisseur) et leurs relations (décrit comment la composition d'items est effectuée pour calculer le coût de services intégrés) dans une politique de facturation.

- Modèle relationnel des objets de données : permet de connaître l'impact qu'a une personnalisation d'une donnée sur une autre. Par exemple, supprimer une donnée peut avoir une influence majeure sur une autre donnée. Ce modèle est représenté comme un diagramme entité/relation où une relation est définie par le triplet : objet source, objet destination, et sa cardinalité.

SaaS DL permet de considérer la personnalisation et la facturation d'un SaaS lors de son intégration avec d'autres applications. Néanmoins, il ne permet pas la représentation des relations de composition entre différents services.

Le Cloud Service Measurement Index Consortium (CSMIC) [CSM] a défini des Service Measurement Index (SMI) qui sont un ensemble d'indicateurs clés de performance, et fournit une méthode standardisée pour mesurer et comparer un service cloud suivant sept métriques : la responsabilité (ex : l'étique du fournisseur, l'auditabilité), l'agilité (ex : l'adaptabilité, l'élasticité, la portabilité), la garantie (ex : la disponibilité, la tolérance aux fautes), le coût (ex : le processus de facturation), la performance (ex : l'interopérabilité, le temps de réponse), la sécurité et vie privée (ex : l'intégrité des données, la perte des données), et l'utilisabilité (ex : l'accessibilité, l'installabilité, la facilité d'apprentissage). A titre d'exemple, pour l'évaluation de l'utilisabilité, des paliers de temps d'apprentissage sont définis en fonction du type de tâche. A chaque palier est associé un poids. Plus le temps est petit, plus le poids est grand.

La description des services comme des entités isolées ne permet pas de connaître les interactions que peut et doit avoir un service donné. Les services métier représentent des packages applicatifs, il n'est donc pas possible d'évaluer leur composabilité comme pour les services Web (matching des entrées/sorties). La connaissance des relations possibles et contraintes d'un service métier s'avère nécessaire à l'automatisation de sa composition.

2.3.2 Description des services et de leurs interactions

Malgré l'enrichissement des langages de description de services (d'un point de vue sémantique et non fonctionnel) comme vu dans les travaux cités précédemment, la description se concentre sur le service décrit et ne donne aucune connaissance ni information sur les services avec lesquels il peut interagir. En effet, les langages de description

de service décrivent ces derniers comme des composants isolés. Certains travaux néanmoins, tels que [NLPvdH12] et [MWB+11], visent à décrire **les relations établies** (qui ont lieu, ou ont eu lieu dans le passé) ou **potentielles** d'un service avec les autres. Une relation potentielle peut être contrainte ou possible. Elle est contrainte si le service décrit ne fonctionne pas correctement sans l'établissement de la relation. Elle est possible si au contraire son établissement ajoute de la valeur au service décrit.

"Web Service Modeling Ontology" (WSMO) [wsm] représente une ontologie de description sémantique de services Web. WSMO décrit les concepts suivants :

- Les ontologies qui représentent la terminologie de domaine de connaissance utilisée.

- Les buts et les objectifs servent à décrire les fonctionnalités désirées par les utilisateurs.

- Les services Web représentent des entités computationnelles donnant accès à un service. A chaque service Web est associée une description de ses fonctionnalités, ses interfaces (orchestration et chorégraphie), et ses paramètres non fonctionnels (ex : le coût du service, la sécurité, etc). Une orchestration décrit l'utilisation d'autres services Web (ou objectifs si les services Web ne sont pas connus), par le service Web décrit pour atteindre son objectif. Une chorégraphie décrit les règles de transition d'un état à un autre (ex : si la carte de crédit n'est pas valide, le paiement est refusé).

- Les médiateurs permettent de résoudre les problèmes d'interopérabilité (incompatibilité) entre les différents éléments de WSMO. Une incompatibilité peut concerner les données (lorsque les services Web utilisent différentes ontologies), le protocole (pour faire communiquer différents services Web), et le processus (pour combiner les services Web).

L'orchestration dans WSMO permet de décrire la décomposition fonctionnelle du service décrit. Elle ne décrit pas les compositions (possibles ou contraintes) entre le service décrit et les autres.

Maamar et al. [MWB+11] s'inspirent des réseaux sociaux pour définir un modèle de découverte de services Web. Le réseau d'un service se construit par la capture des différentes interactions qui ont lieu entre les services Web. Trois types d'interaction sont considérés pour la construction des réseaux d'un service : (i) les services se substituant à ce dernier en cas de panne, (ii) les services avec lesquels il a été en compétition lors d'une découverte, et (iii) les services avec lesquels il a été composé (collaboration). Un

poids est affecté à chaque relation de collaboration. Ce poids est corrélé avec le nombre de fois que les deux services ont été directement composés. Les réseaux d'un service permettent donc de (i) recommander les services avec lesquels il pourrait être composé, (ii) recommander les services qui pourraient le remplacer dans le cas d'une panne (substitution), et (iii) être conscient des services avec lesquels il est en compétition dans le cas d'une sélection. La construction de ce réseau social est un processus itératif et continu qui est initié lorsque le service participe pour la première fois à une composition.

La relation de collaboration permet de connaître les services avec lesquels un service S_i a été composé par le passé. Cette relation permet de connaître la composabilité directe des services mais ne fait pas la distinction entre une contrainte et une possibilité de composition. De plus, la connaissance des relations de collaboration se construit au fur et à mesure des compositions d'un service. Donc, elle ne permet pas de décrire les compositions possibles de manière exhaustive.

Jorge Cardoso [Car13] modélise lui aussi les réseaux d'un service suivant un modèle multi-niveaux comme suit :

- Le rôle. Un service impliqué dans une relation peut prendre le rôle de client, de fournisseur, de service compétiteur, ou de service complémentaire. Deux services sont compétiteurs s'ils appartiennent à des entreprises concurrentes. Un service A est compétiteur du service B si un client qui a accès aux services A et B, évalue A mieux que B. Un service complémentaire augmente la fonctionnalité d'un autre service. Un service A est complémentaire d'un service B si les clients évaluent mieux le service B lorsqu'il est utilisé avec le service A.

- Le niveau de la relation établie (ex : niveau opération, niveau acteur, etc).

- L'implication décrit le degré d'interaction entre les services d'une relation. Cela peut être exprimé en nombre de clients utilisant l'association de deux ou plusieurs services.

- La comparaison fonctionnelle entre services. Un service peut être équivalent, similaire, différent, une généralisation ou une spécialisation d'un autre service.

- L'association de services décrit l'agrégation et la composition de services. En d'autres termes, cela revient à décrire les composants d'un service composite.

- La causalité décrit l'influence des fluctuations d'un indicateur de performance d'un service sur un autre.

L'association permet de décrire les compositions établies entre services constituant ainsi le service composite décrit. L'association ne fait pas la distinction entre les compositions possibles et contraintes. La complémentarité entre deux services implique une relation possible de composition entre ces derniers. Elle ne permet cependant pas de décrire les possibilités de composition de manière exhaustive, étant donné que l'information est construite sur la base de l'évaluation des clients de l'utilisation des services.

Nguyen et al. [NLPvdH12] décrivent les services cloud (XaaS) en utilisant des blueprints. Le terme blueprint étant un anglicisme utilisé pour représenter souvent un plan détaillé correspondant à un dessin technique, nous pouvons faire appel au blueprint pour décrire des processus, des flux, des relations, des dépendances etc. Un blueprint est utilisé dans ce travail pour décrire les offres et les besoins d'un service. A chaque offre, un ou plusieurs besoins peuvent être associés. La notion de besoins englobe l'environnement de déploiement tel que le serveur Web pour une application Web, les services dont a besoin le service décrit pour fonctionner comme une base de données pour une application CRM, ou encore les ressources nécessaires (ex : une machine virtuelle avec un système d'exploitation en particulier). Deux types d'offres sont considérés. Une offre est dite offre source si elle est prête à l'emploi, i.e., elle ne contient pas de besoins non satisfaits. Un besoin est satisfait s'il existe une offre dans la marketplace qui correspond à la fonctionnalité du besoin. Une offre est dite offre cible si elle est en cours de développement et contient des besoins qui ne sont pas satisfaits par d'autres offres. Une offre cible est en plus décrite par une contrainte globale de qualité de service, par exemple un temps de réponse inférieur à 5ms. Le mécanisme de composition décrit dans [NLPvdH12] est un processus itératif qui associe pour chaque offre cible des offres sources la satisfaisant jusqu'à ce que toutes les offres soient de type offres sources. Cela est représenté dans un Modèle d'Instance de Blueprints (MIB). Lorsque l'utilisateur requête un blueprint via une requête SPARQL (SPARQL Protocol and RDF Query Language), le MIB lui retourne les offres répondant à sa requête avec la liste des résolutions possibles pour le blueprint.

Les besoins d'un service XaaS définissent une relation de contraintes souples. Ils représentent des contraintes de par la nécessité pour un service donné de communiquer avec une base de données, ou d'être déployé sur un certain type de machine virtuelle. Ces contraintes sont souples puisqu'elles sont décrites à un haut niveau d'abstraction de détails techniques (ex : serveur d'application, base de données, machine virtuelle Linux). En effet, MySQL et PostgreSQL répondent toutes deux au besoin "bases de données". De plus, le concept de besoin ne fait pas la distinction entre les types de besoins considérés. De ce fait, nous ne pouvons pas automatiser la composition et le

déploiement de services. Les possibilités de composition ne sont pas décrites dans ce travail.

Zhang et al. [ZRH+12] décrivent une ontologie de cloud computing "Cloud Computing Ontology" (CoCoOn) de description des services IaaS. L'ontologie décrit les services cloud de manière fonctionnelle et non fonctionnelle. Le principal concept de la partie fonctionnelle est le "cloudResource" qui englobe le SaaS, PaaS et IaaS. Seule la partie IaaS a été implémentée. Les auteurs classifient une IaaS en trois catégories : calcul, réseau, et stockage. Le calcul est le concept principal des services IaaS. Le réseau, et le stockage sont souvent attachés aux ressources de calcul. Les auteurs définissent les propriétés "isAttached" et "isAttachable". Ces dernières décrivent respectivement un attachement établi et un attachement possible entre une ressource de calcul et de stockage. De la même manière, les propriétés "hasNetwork" et "hasSupportedNetwork" permettent de décrire un attachement établi et possible entre une ressource réseau et une ressource de calcul. La partie non fonctionnelle inclut les propriétés non fonctionnelles (ex : le prix, le fournisseur, le modèle de déploiement), et les paramètres de qualité de service.

Les propriétés "isAttachable" et "hasSupportedNetwork" décrivent **les relations potentielles possibles d'attachement entre deux services IaaS**. Elles ne décrivent pas de possibilités de composition entre services métier.

Le langage USDL (Unified Service Description Language) [usd] permet de décrire les services techniques (WSDL, REST) et métier de manière à permettre aux services de devenir négociables et consommables. Ce langage a été initié entre autres par Attensity et SAP Research, et proposé pour standardisation auprès du W3C. La description de services avec USDL se fait selon trois perspectives : métier, opérationnelle, et technique. La perspective métier couvre entre autres la qualité du service, les modèles de prix, et les contraintes légales. La perspective opérationnelle concerne les opérations d'un service et ses fonctionnalités. La perspective technique inclut les protocoles de transport, de messages, d'échanges de méta-données, et de sécurité.

Le langage Linked USDL [linb], [CP15], [linc] est un langage sémantique fondé sur le langage USDL. Il permet de décrire les services humains (ex : conseil), les services métier (ex : demande d'achat), les services logiciels (ex : services WSDL et RESTful), les services infrastructure (ex : CPU et services de stockage), etc. Linked USDL est basé sur les principes du Linked Data [lina], i.e, il réutilise plusieurs vocabulaires liés, ce qui facilite son extension.

L'objectif de Linked USDL est d'offrir un langage qui permet une description ouverte, adaptable, et extensible des services en utilisant une gestion décentralisée. Linked USDL repose sur le vocabulaire RDF. Ce dernier est très utilisé pour structurer des don-

nées dans des endpoints tel que dbpedia [dbp], qui sont interrogés avec des requêtes SPARQL. Les services décrits avec Linked USDL sont identifiés grâce aux URIs HTTP. L'utilisation des URIs fournit une manière simple de créer des identifiants uniques pour les services. Un accès uniforme, global, et standard aux descriptions de services est fourni par les URLs HTTP et RDF [CBB+13].

Linked USDL est divisé en cinq modules qui ont différents niveaux de maturité. Chaque module est un ensemble de concepts et de propriétés qui les lient. Afin de réduire la complexité de la modélisation du service, seuls les modules requis sont utilisés. Nous illustrons dans la Figure 2.1 quelques classes de Linked USDL. Une description détaillée de tous les modules est disponible dans [linc]. Les cinq modules sont les suivants :

- USDL-core : permet de décrire les aspects opérationnels d'un service. Les classes du vocabulaire Good Relations [goob] "gr:serviceOffering" et "gr:location" sont utilisées dans ce module pour décrire une offre de services et sa localisation. Une offre de services inclut un ou plusieurs services. La classe "skos:concept" décrit la classification du service décrit (ex : IaaS, SaaS, etc).

- USDL-price : permet de décrire les structures de prix d'un service. La classe "gr:PriceSpecification" est importée du vocabulaire Good Relations. Elle permet de décrire le prix d'un composant de prix "PriceComponent" et d'un plan de prix "PricePlan". Un plan de prix décrit les différentes options de paiement d'un service (ex : payer un prix fixe pour une certaine prestation ou payer en fonction de l'utilisation). Un composant de prix décrit les honoraires inclus dans un plan de prix (ex : les frais d'adhésion au service, les frais périodiques, et les prix de souscription à certaines options). Le prix d'un plan de prix représente la somme des honoraires de ses composants. Les propriétés "hasPriceCap" et "hasPriceFloor" permettent de décrire respectivement les limites supérieures et inférieures des prix attribués à un plan de prix. De la même manière, les propriétés "hasComponentCap" et "hasComponentFloor" décrivent respectivement les limites supérieures et inférieures des prix attribués à un composant de prix.

- USDL agreement : permet de décrire la qualité du service fourni tels que le temps de réponse et la disponibilité. La classe "AgreementTerm" décrit un terme du contrat SLA. La classe "Guarantee" permet de décrire les garanties sur les propriétés d'un service. La classe "Compensation" décrit une alternative au cas où les garanties décrites ne sont pas respectées.

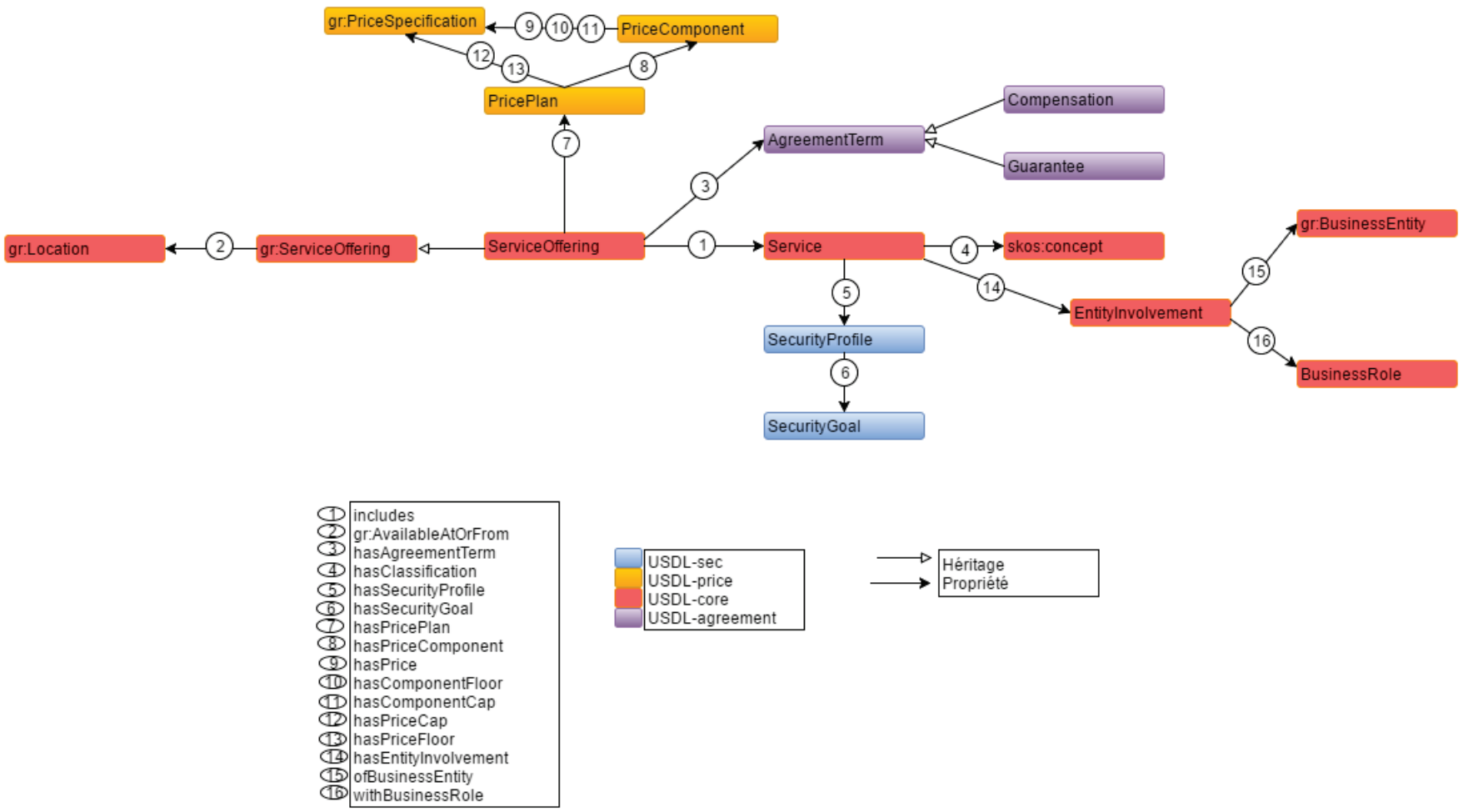


Figure 2.1: Vue macroscopique de Linked Unified Service Description Language

- USDL-sec : permet de décrire les propriétés liées à la sécurité d'un service via la classe "SecurityProfile". La classe "SecurityGoal" décrit le plus haut niveau d'abstraction des propriétés de sécurité telles que la confidentialité et l'authentification.

- USDL-ipr : permet de décrire les droits d'utilisation d'un service.

Le type de relations entre services décrit dans Linked USDL est fait grâce à la classe "usdl-core:ServiceOffering". Cette dernière permet de décrire les services combinés constituant une offre de service. Plusieurs offres de services peuvent exister associant plus ou moins les mêmes services. Une offre de services décrit une composition possible des services impliqués. Cependant, toutes les compositions possibles ne sont pas nécessairement décrites par une offre de service, étant donné qu'une offre de services dépend du fournisseur.

2.3.3 Analyse des langages de description de services

Nous avons rapporté dans cette sous-section des travaux décrivant des langages de description de services. Certains de ces travaux considèrent les services comme des entités isolées. D'autres en revanche, modélisent les relations établies et/ou potentielles entre services.

Le Tableau 2.1 classifie les travaux cités dans cette sous-section en fonction du type de description modélisée selon les critères suivants :

C1 : Type de service décrit.

C2 : Description technique.

C3 : Description sémantique.

C4 : Description de la qualité des services.

C5 : Description des relations d'un service.

Parmi ces travaux, Linked USDL est celui permettant la description d'un grand nombre de services (métier, cloud, Web) alors que les autres langages sont spécifiques. De plus, Linked USDL offre une grande couverture des aspects métier, techniques et opérationnels. Cependant, Linked USDL ne décrit pas les relations potentielles d'un service. Il décrit plutôt les relations établies entre services constituant une offre de service. Néanmoins, Linked USDL permet une extension facile par l'ajout de nouveaux concepts. Nous nous sommes donc basés sur ce langage pour la description des services de la

Tableau 2.1: Classification des travaux de description de services en fonction du type de description

Travaux	**C1**	**C2**	**C3**	**C4**	**C5**
WSDL [wsd]	Services Web	+			
QoS for WSDL [D'A06]	Services Web	+		+	
[BA14]	Services Web			+	
OWL-S [MPM+04]	Services Web	+	+		
SAWSDL [saw]	Services Web	+	+	Peut être faite par l'instanciation d'une ontologie décrivant les QoS	
[AMBT14]	SaaS	+	+	+	
[HS10]	Services cloud	+	+		
[KS11]	Services cloud	+	+		
[TBAT12]	Services cloud	+	+		
Cloud# [LZ11]	Services cloud	+			
BDL [TNL+12]	Services cloud	+		+	
SaaS DL[SZC+07]	SaaS	+			
CSMIC [CSM]	Services cloud			+	
WSMO [wsm]	Services Web	+	+	+	+
LinkedWS [MWB+11]	Services Web				+
Réseaux de service [Car13]	Services sémantiques		+	+	+
Blueprint [NLPvdH12]	Services cloud	+	+	+	+
CoCoon [ZRH+12]	Services IaaS	+	+	+	+
Linked USDL [linb]	Services métier, services logiciels, services infrastructures, et services humains	+	+	+	+

marketplace et l'avons étendu pour décrire les relations potentielles contraintes et possibles d'un service métier.

Nous allons à présent classifier dans le Tableau 2.2, les travaux décrivant les relations d'un service en fonction des types de relations décrites sur la base des critères suivants :

C1 : Décrit les relations établies entre services.

C2 : Décrit les relations potentielles possibles entre services.

C3 : Décrit les relations potentielles contraintes entre services.

Tableau 2.2: Classification des travaux de description de services considérant les relations d'un service en fonction du type de relation décrite

Travaux	**C1**	**C2**	**C3**
WSMO [wsm]	+		
LinkedWS [MWB+11]	+		
Réseaux de service [Car13]	+		
Blueprint [NLPvdH12]			+
CoCoon [ZRH+12]	+	+	
Linked USDL [linb]	+		

Parmi les concepts décrivant les relations d'un service nous retrouvons :

(1) L'association [Car13] et la collaboration [MWB+11] décrivent les relations établies des services ayant été composés par le passé. La connaissance de ces relations n'est pas exhaustive puisqu'elle se construit au fur et à mesure des compositions. De plus, ces concepts ne font pas la distinction entre les services **devant** être composés (contraintes de composition) et ceux **pouvant** l'être (possibilités de composition). Ne pas distinguer les contraintes des possibilités de composition d'un service ne nous permet pas de connaître la composition minimale permettant le bon fonctionnement d'un service métier donné. **Nous avons donc choisi de décrire les contraintes et possibilités de composition de chaque service métier.**

(2) Les rôles de complémentaire et compétiteur dans [Car13]. Le rôle de complémentaire implique qu'il existe une possibilité de composition entre les services. En revanche, cette notion étant basée sur l'évaluation de l'utilisation des services, elle se construit au fur et à mesure des utilisations. Elle ne décrit pas toutes les possibilités de composition d'un service.

(3) Les besoins d'un service [NLPvdH12]. Ces besoins concernent les contraintes d'environnement du service, celles concernant les ressources, et celles concernant la composition d'un service. Les besoins définis dans le blueprint [NLPvdH12] regroupent ces besoins dans un seul concept, ce qui ne permet pas d'automatiser la composition et le déploiement des services. En effet, il n'est pas possible d'identifier automatiquement le type de besoin représenté (contrainte de composition ou de déploiement). **Nous avons fait la distinction dans la description de chaque service entre les concepts décrivant les contraintes de composition, les contraintes d'environnement, et les ressources nécessaires.**

(4) Les offres de services [linb, ZRH+12] combinant plusieurs services. Elles peuvent aussi bien décrire des relations établies (usdl-core:ServiceOffering, cocoon:isAttached, cocoon:hasNetwork) que les relations potentielles possibles (cocoon:isAttachable, cocoon:hasPossibleNetwork). Les offres de services décrivant des relations établies sont très dépendantes des fournisseurs et ne décrivent pas toutes les possibilités de combinaisons de services. Les relations potentielles possibles décrites dans CoCoOn sont spécifiques aux services IaaS liant les services de stockage et de calcul, et les services de réseau et de calcul.

Nous faisons dans ce travail la distinction entre les types de contraintes et définissons les concepts de contraintes de composition, d'environnement, et de ressources. Nous ajoutons à ce travail de distinction la description des possibilités de composi-

tion de chaque service métier, permettant ainsi de connaître toutes les relations potentielles de composition (possibles et contraintes) d'un service métier. Dans un souci d'automatisation de la configuration des applications métier sur les environnements cloud, nous décrivons aussi les paramètres configurables de chaque service métier.

2.4 Elicitation des besoins pour le développement d'applications

La description des besoins lors du développement d'applications diffère en fonction de la méthode de développement utilisée [BIN10]. En effet, dans un développement en cascade, l'élicitation des besoins constitue la première phase du processus de développement. Le livrable de cette dernière constitue les spécifications fonctionnelles de l'application désirée. Dans une méthode agile en revanche, les besoins sont évolutifs. Ils sont affinés au fur et à mesure des itérations par l'implication de l'utilisateur.

Les travaux traitant de l'élicitation des besoins lors du développement d'applications traitent de la découverte et de la définition des besoins de l'utilisateur en termes d'objectifs et de fonctionnalités pour l'application désirée [ZC05] [NE00], [CBCG10]. Ils décrivent des techniques et approches (entretiens, questionnaires, utilisation de médias riches [HPW98], etc) permettant de récolter et de comprendre les besoins des différents acteurs tels que les intervenants métier, les utilisateurs finaux, et les ingénieurs.

Les travaux de la littérature distinguent les besoins fonctionnels des besoins non fonctionnels [Jal05]. Les besoins fonctionnels décrivent les actions qui doivent être effectuées par le système. Les besoins non fonctionnels quant à eux décrivent les propriétés du système [Jal05], tels que les besoins en qualité de services.

Dans ce travail, nous considérons que les besoins sont décrits par l'utilisateur via une interface Web. Nous ne nous intéresserons donc pas aux techniques d'élicitation des besoins, mais plutôt aux types de besoins pris en compte pour le développement d'applications. Nous distinguons dans ce qui suit deux situations dans lesquelles l'élicitation des besoins diffère, à savoir, le développement classique et le développement orienté service.

2.4.1 Pour le développement classique

UML (Unified Modeling Language) [uml] est utilisé pour l'élicitation des besoins pour le développement orienté objet. UML permet de représenter la conception d'un système logiciel. Il regroupe des notations et formats permettant de modéliser le système selon plusieurs vues (ex : architecturale, d'implémentation, de déploiement, etc) grâce aux diagrammes UML. Plusieurs diagrammes peuvent être représentés. Nous citons entre autres, (i) les diagrammes de cas d'utilisation décrivant des scenarii représentant le comportement fonctionnel du système et son interaction avec les acteurs (ex : utilisateurs de

l'application), (ii) les diagrammes de classe décrivant les classes du système et les relations les liant [ZC05], et (iii) les diagrammes de séquence qui décrivent le déroulement des traitements entre les éléments du système et ses acteurs.

Ce langage permet de décrire dans le détail le fonctionnement de l'application à développer, son implémentation, son déploiement, etc. Une bonne connaissance de ces derniers et de la notation UML est donc nécessaire pour modéliser l'application. Les spécifications UML sont ensuite utilisées par les développeurs pour l'implémentation de l'application.

Une technique d'analyse fonctionnelle descendante assez ancienne SAD (Structured Analysis and Design) [DeM79, You88] décrit l'élicitation des besoins orientée fonction. Elle permet entre autres de décrire la décomposition fonctionnelle des composants du système en décrivant les données entrantes et sortantes à l'aide de diagrammes de flux de données "Data Flow Diagrams" (DFD). La représentation des entités du système, leurs attributs, et leurs relations est effectuée grâce aux diagrammes Entité Association [ZC05].

Dans ce travail aussi, une bonne connaissance des besoins du système à développer (en termes des flux de données et des entités constituant le système) et de la notation des diagrammes est nécessaire.

Hong et al. [HCS05] prennent en compte le contexte dans la définition des besoins pour la conception d'applications ubiquitaires. Ils distinguent trois types de contextes :

- Le contexte de calcul comprenant ce qui est relatif à la configuration hardware, comme la taille de l'écran utilisé, le processeur et la bande passante.

- Le contexte de l'utilisateur comprenant les facteurs humains tel que le profil de l'utilisateur.

- Le contexte physique comprenant les informations fournies par l'environnement tels que la localisation, le temps, et la température.

La prise en compte du contexte lors de la définition des besoins d'une application permet de définir la réaction du système en fonction de certains paramètres comme la taille de l'écran utilisé ou le profil de l'utilisateur. Là encore, une bonne connaissance des attentes que l'on a envers la réactivité et l'adaptabilité techniques de l'application à développer est nécessaire.

Les applications cloud ont pour caractéristique d'être multi-tenants. De ce fait, les différents clients d'une même application peuvent utiliser des configurations différentes de cette dernière parce qu'ils ont des besoins spécifiques différents. Zhou et al. [ZYL11] distinguent entre les besoins communs à plusieurs clients et ceux spécifiques à certains clients lors de l'élicitation des besoins d'une application cloud. Ils définissent une technique collaborative de recueil de besoins "Collaborative Requirement Elicitation Technique" (CRETE). Les clients potentiels d'un SaaS peuvent voter[2] pour les besoins com-

[2]Ce vote représente une sélection des besoins existants. Il a pour objectif de déterminer les besoins récurrents.

muns ayant été introduits par d'autres clients, et en exprimer de nouveaux en termes de fonctionnalités de l'application. Les informations relatives à un besoin sont les fonctionnalités, le raffinement d'une fonctionnalité (hiérarchisation sur différents niveaux d'abstraction), et les contraintes décrivant les dépendances entre fonctionnalités (ex : une relation d'exclusion entre deux fonctionnalités). La modélisation collaborative de besoins inclut le vote direct, le vote propagé, et les préférences du client envers un besoin voté "oui" (ex : le nom ou la description du besoin). Un vote direct a lieu lorsque le client vote "oui" ou "non" pour un besoin exprimé par le fournisseur de SaaS ou par d'autres clients. Lorsque le client vote "oui" pour une relation de raffinement, ce vote est propagé pour les deux fonctionnalités impliquées dans la relation. Les besoins communs et spécifiques sont combinés pour permettre au fournisseur de développer un SaaS où les besoins individuels (spécifiques à chaque client) peuvent être satisfaits par une configuration de l'application sans avoir à changer le code source.

Les besoins exprimés dans CRETE permettent aux fournisseurs de SaaS de développer ou d'adapter les logiciels fournis afin de mieux correspondre aux attentes des clients. Ils ne sont pas prévus pour la génération automatique d'applications métier.

2.4.2 Pour le développement d'applications orientées service

L'architecture AOS est basée sur les services et les processus [PJ08]. L'ingénierie des besoins dans les approches orientées service permet de définir des modèles, des notations, des ontologies et des langages pour modéliser l'application désirée sous forme de workflows de services [bpm, wsb, wsca, YZ15, XLQY07], ou encore de définir des langages de requêtes de services [MZBL13].

BPMN (Business Process Model and Notation) [bpm, soa] est un standard permettant de représenter de manière graphique les processus métier d'une organisation. BPMN 2.0 permet de définir trois grands types de diagrammes : les orchestrations, les chorégraphies, et la collaboration. Une orchestration décrit une séquence de flux d'activités dans un processus. La collaboration décrit les interactions entre deux ou plusieurs processus. Une chorégraphie est une vue spécifique d'une collaboration qui décrit les flux de messages entre différentes tâches de différents participants d'une collaboration.

WS-BPEL (Web Service Business Process Execution Language) est un langage basé sur XML et standardisé par OASIS [wsb, RFG09] permettant la description d'un processus métier dans lequel les actions sont effectuées par des services Web. Ces actions peuvent être contenues dans des boucles grâce aux attributs "forEach", "while", "repeatUntil", ou être exécutées en séquence grâce à la balise "sequence". L'invocation d'un service Web se fait par l'attribut "invoke". D'autres attributs permettent entre autres de communiquer avec l'extérieur par la réception d'un message d'un service Web grâce à l'attribut "receive". L'exécution de ce processus est effectuée par un moteur d'orchestration.

WS-CDL (Web Service Choreography Description Language) est lui aussi basé sur XML [wsca, wscb]. Il décrit les collaborations pair à pair multi-parties d'un point de

vue global, c'est à dire, il permet de décrire l'échange de données ainsi que le format d'échange des données entre les différents participants alors que WS-BPEL le fait selon le point de vue d'un seul participant [wscb].

BPMN et WS-BPEL peuvent être combinés pour respectivement décrire et exécuter les processus métier. Pant et al. [PJ08] décrivent un mapping automatique des modèles BPMN vers WS-BPEL pour l'exécution d'un processus suivant l'AOS.

L'utilisation de BPMN, WS-BPEL, et WS-CDL nécessite la connaissance de la notation et du langage utilisés pour la représentation d'un processus métier. D'un autre côté, la modélisation d'une application orientée service sous la forme d'un processus métier nécessite la connaissance des flux de contrôle et de données entre les différents services concernés dans une composition.

Xiang et al. [XLQY07] proposent un mécanisme d'élicitation des besoins en services "Service Requirements Elicitation Mechanism" (SREM). Ce mécanisme est constitué de :

(i) L'ontologie "Service Requirements Modeling Ontology" (SRMO) qui est illustrée dans la Figure 2.2. Dans SRMO, un service peut être un objectif, une tâche, ou une ressource. La classe "Goal" (objectif) permet de décrire une condition ou un état que l'utilisateur souhaite atteindre ou réaliser. La relation entre différents objectifs peut être de type (i) "Sequence" où les objectifs sont exécutés les uns à la suite des autres, (ii) "Unordered" où les objectifs peuvent être exécutés dans n'importe quel ordre, et (iii) "Choice" où un seul objectif peut être exécuté parmi toutes les alternatives. Chaque objectif peut être réalisé par une ou plusieurs tâches "Task". Un attribut qualité "Quality attribute" peut être associé à chaque tâche ou objectif pouvant décrire le coût désiré ou une qualité de service tels que le temps de réponse ou la disponibilité. Une ressource représente une entité physique ou informationnelle associée à une tâche pour la réalisation de l'objectif correspondant, décrivant à titre d'exemple la table des taxes à appliquer permettant l'exécution de la tâche de calcul des taxes de tous les achats d'un client. SRMO considère deux types d'acteurs : le fournisseur de service et le requêteur de service.

(ii) Du processus d'élicitation de besoins en tant que service "Service Requirements Elicitation Process" (SREP). SREP définit des questions à poser au requêteur permettant de générer les besoins de ce dernier suivant les concepts de l'ontologie SRMO (ex : quels sont les sous-composants d'un objectif?).

(iii) Du processus de réconciliation des besoins en service "Service Requirements Reconciliation Process" (SRRP). SRRP permet de fusionner les besoins venant de différents requêteurs visant le même objectif (service). Chaque besoin (tâche, objectif) est pondéré par le nombre de fois que ce dernier a été exprimé par les requêteurs. Cette information est utile pour les fournisseurs de service afin de connaître la pertinence d'un besoin exprimé.

Même si avec SREM, le requêteur n'a nul besoin de connaître l'ontologie de de-

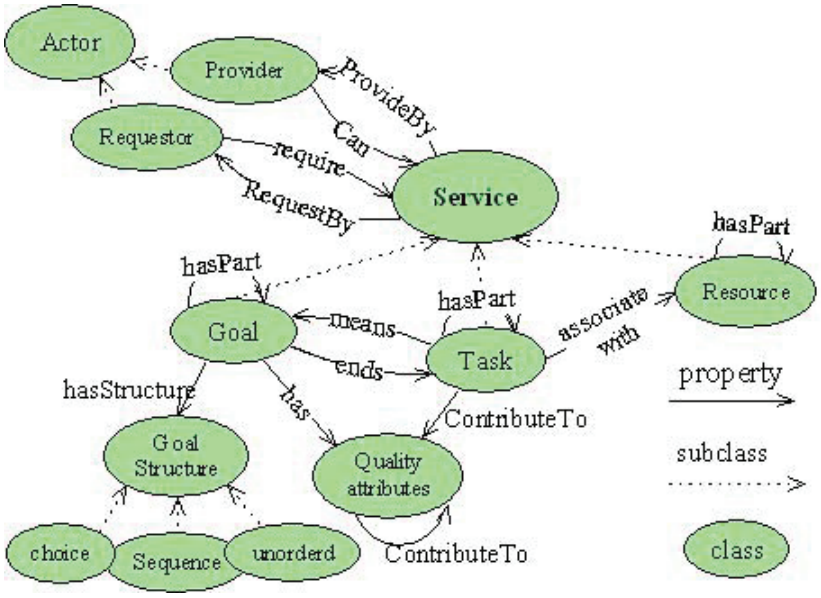


Figure 2.2: Service Requirement Modeling Ontology [XLQY07]

scription des besoins (SRMO) étant donné que ces derniers sont élicités sur la base de questionnaires définis dans SREP, l'utilisateur doit tout de même connaître les différents objectifs, tâches, et ressources nécessaires, ainsi que l'ordre des objectifs à réaliser, à savoir séquentiel, désordre, ou encore en exclusion mutuelle.

Yuan et al. [YZ15] décrivent une approche d'élicitaion interactive de besoins pour le développement de "Product Line Software" (PLS) orienté service personnalisé. Après que les ingénieurs aient décrit les fonctionnalités liées à un domaine, l'approche d'élicitaion interactive des besoins consiste à faire évaluer les besoins par le client via un dialogue Homme/Machine en langage naturel. Cela sert à identifier l'ensemble des besoins qui satisfont les clients parmi ceux définis par les ingénieurs. Les auteurs développent une ontologie pour représenter les fonctionnalités d'un domaine particulier. Le modèle d'ontologie décrit est illustré par la Figure 2.3. Il décrit les concepts suivants :

- Requirement : décrit un besoin qui peut être nécessaire à l'utilisateur. Un besoin est soit fonctionnel (fonctionnalité), ou non fonctionnel (qualité ou SoftGoal).

- Functions : décrit une fonctionnalité qui peut être atomique ou composite.

- Quality : décrit une contrainte imposée à une fonctionnalité. A titre d'exemple pour une fonctionnalité de recherche, deux qualités peuvent être associées, à savoir, une recherche par matching exacte ou large.

- SoftGoal : décrit l'environnement global dans lequel le PLS fonctionne.

- Rank : décrit l'importance d'un besoin. Il définit l'ordre d'évaluation des besoins par le client.

- OtherInfo : utilisé pour décrire les informations sur les besoins qui peuvent être utilisées lors de l'évaluation des besoins par l'utilisateur.

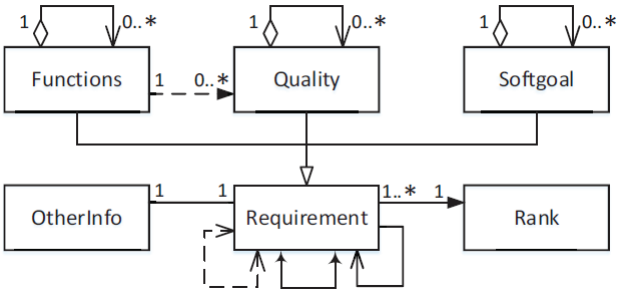


Figure 2.3: Modèle d'ontologie pour la description des besoins pour le développement de PLS [YZ15]

L'ontologie décrit aussi sept relations comme suit :

- Generalize : décrit le lien entre un concept général et ses cas particuliers.
- Decompose : permet de décomposer un besoin composite en besoins plus atomiques.
- Rely : décrit la relation de dépendance entre deux besoins. En effet, X repose "rely" sur Y implique que la sélection de X nécessite la sélection de Y.
- Contradict : permet de décrire l'exclusion entre deux besoins. En effet, X contredit "contradict" Y implique que la sélection de X empêche la sélection de Y et inversement. Cette relation est utile lorsque deux besoins ont le même rôle mais ont des contraintes différentes comme tel est le cas pour le choix entre une recherche par matching exact ou large.
- Associate : permet de décrire l'association d'une fonctionnalité avec ses contraintes de qualité. Une contrainte de qualité ne peut être réalisée si la fonctionnalité associée n'est pas sélectionnée.
- HasRank : permet d'associer un rang à un besoin.
- Invalid : indique une invalidité entre deux besoins.

L'ontologie décrite dans ce travail permet aux ingénieurs de décrire les besoins d'une application PLS orientée service en vue de l'automatisation de l'élicitation du besoin personnalisé pour chaque client. Elle nécessite une bonne connaissance de la décomposition fonctionnelle du besoin, des relations entre les besoins (contradiction, qualité, etc) et de l'ordre d'évaluation du besoin par les clients potentiels.

Mitra et al. [MZBL13] définissent un langage formel déclaratif dénommé "Web Service Request Language" (WSRL) qui contrairement au langage WSDL, se centre sur ce

que l'utilisateur attend du service (atomique ou composite) plutôt que sur ce que permet de faire le service décrit. L'utilisateur définit ses besoins dans le système de requête de services Web (comme décrit dans la requête WSRL du Listing 2.1) en termes de variables d'information associées aux entrées décrites (ex : type de processeur="Intel" (ligne 3 du Listing)); de fonctionnalités (ex : InitiatedDelivery (ligne 6)); de contraintes de qualité (ex : ServiceFee<10$ (ligne 7)); et de valeur de retour pour la satisfaction d'une demande de service (ex : reçu de paiement (ligne 8)).

```
1 request{
2 require<info> Laptop : : ComputerType = " Laptop "
3 and Laptop : : Processor = "Intel"
4 and Laptop : : RequiredRating ="5"
5 and Laptop : : ScreenSize = "15 in "
6 require<fn> Laptop : : initiatedDelivery
7 require<pol> Laptop : : ServiceFee < "\$10"
8 return Laptop : : PurchaseReceipt
9 }
```

Listing 2.1: Requête WSRL pour l'achat d'un ordinateur

A partir de la requête de l'utilisateur, le système WSRS (Web Service Request System) génère des graphes (dirigés) d'états qui décrivent les différents états engendrés par l'invocation des services associés aux fonctionnalités désirées. Un état décrit les variables de fonctionnalités (ex : achat d'ordinateur="true" pour dire que cette fonctionnalité est invoquée) et les variables sur les informations instanciées. Le système de requête de services Web invoque alors dynamiquement des services Web répondant aux besoins de l'utilisateur.

L'utilisation de WSRL requiert de l'utilisateur de connaître la notation de ce langage de requête. WSRL permet d'invoquer les services correspondant aux fonctionnalités décrites dans la requête, il ne permet pas de construire une application métier composite.

2.4.3 Analyse des travaux traitant de l'élicitation des besoins pour le développement d'applications

Les services cloud réduisent fortement les connaissances techniques requises pour leur utilisation par l'automatisation de leur gestion côté fournisseur. Dans l'optique de se conformer à cette caractéristique, l'objectif de notre travail est d'automatiser la génération d'applications cloud orientées service à partir d'un besoin non technique de l'utilisateur exprimé via une interface Web. La minimisation des connaissances passe par le fait que l'utilisateur puisse exprimer son besoin à un haut niveau d'abstraction des détails techniques, et ne nécessite pas de connaître un langage, une notation ou une ontologie pour exprimer son besoin. Nous avons choisi de ne comparer que les travaux liés aux approches orientées service vu que c'est le contexte de notre travail. Nous avons donc choisi les critères suivants pour comparer dans le Tableau 2.3 les travaux cités dans cette section.

C1 : Objectif du travail.

C2 : Niveau d'abstraction de la description du besoin. Cela permet de décrire le niveau de détail exigé pour les besoins décrits.

Tableau 2.3: Comparaison entre les travaux de description des besoins

Travaux	**C1 : Objectif du travail**	**C2 : Niveau d'abstraction**
BPMN [bpm]	Description des processus métier	Les activités désirées, les évènements attendus, les flux de contrôle, les différents participants d'une collaboration, le flux de messages
WS-BPEL [wsb]	Description de l'exécution d'un processus métier orienté service	Flux de contrôle, flux de données
WS-CDL [wsca]	Description des chorégraphies point à point	Flux de contrôle, flux de données
SRMO [XLQY07]	Ontologie de description des besoins	Les fonctionnalités désirées, les relations entre les fonctionnalités, les attributs qualité, les flux de contrôle
[YZ15] pour le développement de PLS orienté service	Permettre l'identification des besoins spécifiques d'un client en particulier	Les fonctionnalités désirées, l'environnement de fonctionnement, les contraintes associées à chaque fonctionnalité, le rang d'un besoin, les relations entre les besoins (contradiction, qualité, etc)
WSRL [MZBL13]	Description déclarative de requête orientée service	Les fonctionnalités désirées, les entrées/sorties, la requête WSRL

Les travaux cités dans le Tableau 2.3 permettent de décrire le besoin des applications orientées service. Néanmoins, certains visent des sous-objectifs comme la personnalisation des applications PLS; la définition d'un langage, une ontologie ou une notation pour la description du besoin etc. Les travaux décrivant des notations et des langages imposent à l'utilisateur de connaître ces derniers afin de pouvoir les utiliser. Ils peuvent être distingués par le niveau d'abstraction lors de la description des besoins (critère C2). En effet, la description d'un flux de contrôle et/ou d'un flux de données nécessite une bonne connaissance du processus métier à développer. Par ailleurs, cela n'est pas propice à l'utilisation des services métier, car ces derniers ne sont pas définis par leurs entrées/sorties (plus haut niveau d'abstraction que les services Web), et leur composition ne consiste pas à invoquer des opérations de services mais plutôt à adapter un service métier afin qu'il puisse coopérer avec un autre.

2.5 Conclusion

Il ressort de notre analyse de l'état de l'art que :

- Les travaux traitant de la description de services ne permettent pas de décrire les relations potentielles possibles et contraintes des services métier de manière exhaustive [MWB+11] [NLPvdH12] [Car13]. Cependant, la connaissance de ces paramètres s'avère primordiale si l'on souhaite automatiser la composition de services métier. En effet, il n'est pas possible d'évaluer la composabilité de ces derniers comme pour les services Web (matching des entrées/sorties) en raison

de leur nature différente. Rappelons que nous avons défini un service métier comme un package applicatif fournissant une fonctionnalité métier, composable avec d'autres services métier, et déployable sur plusieurs infrastructures.

- Les travaux traitant de l'élicitation des besoins pour le développement d'applications orientées service, proposent des langages et des notations permettant de décrire un besoin sous forme de processus métier [bpm] [wsb] ou de requête de services [MZBL13]. Il proposent aussi des ontologies décrivant les concepts de besoins à considérer [YZ15]. Les travaux décrivant des langages et notations exigent de l'utilisateur de connaitre ces derniers afin de pouvoir les utiliser. De plus, la plupart des travaux exigent de l'utilisateur d'avoir une connaissance détaillée de son besoin, notamment du flux de contrôle et de données entre les différents services de l'application.

Dans notre travail, nous proposons un environnement et une méthodologie de provisionnement d'applications cloud orientées service. Ce travail s'inscrit dans la démarche de diminution des connaissances techniques requises de l'utilisateur pour le développement d'applications métier sur le cloud. Il satisfait la caractéristique d'accessibilité au grand public des technologies cloud par l'automatisation du développement orienté service et du déploiement de l'application désirée.

Nous avons choisi de faire en sorte que l'utilisateur puisse exprimer son besoin via une interface Web (pas besoin d'un environnement spécifique) à un haut niveau d'abstraction des détails techniques. Ces besoins incluent les fonctionnalités désirées, les contraintes de qualité de service et les préférences de l'utilisateur. Ils sont ensuite automatiquement traduits par notre système pour les formaliser suivant le vocabulaire RIVAL (RequIrement VocAbuLary) (cf. Chapitre 4).

Afin de permettre une composition automatique des composants de l'application désirée, nous avons choisi d'utiliser et d'étendre le langage Linked USDL pour la description des services de la marketplace. Ce choix a été guidé par le fait que Linked USDL offre une grande couverture des aspects métier, techniques et opérationnels, et permet une extension facile par l'ajout de nouveaux concepts (cf. Section 2.3.3). Nous avons étendu Linked USDL pour décrire les relations qu'un service métier a avec les autres en termes de contraintes et de possibilités de composition. Nous avons aussi intégré la description des ressources nécessaires au déploiement (ex : CPU, mémoire...) ainsi que les paramètres configurables de chaque service métier afin d'automatiser le déploiement et la configuration de ce dernier.

Chapitre 3

Développement et Déploiement des Applications sur les Environnements Cloud

Sommaire

3.1 Introduction

Ce chapitre a pour objectif d'identifier les différents travaux traitant du développement et du déploiement d'applications sur les environnements cloud. Nous avons classifié ces travaux comme suit :

- Environnements de développement et de déploiement d'applications cloud.
- Approches de développement et de déploiement d'applications cloud.
- Architectures pour les applications SaaS multi-tenants [PLL10] (hors de portée de ce travail).

Nous discutons ces différents travaux afin de positionner notre travail par rapport aux solutions existantes.

3.2 Environnements de développement et de déploiement d'applications cloud

SOCCA (Service-Oriented Cloud Computing Architecture) [TSB10] combine AOS et cloud computing afin que différents clouds puissent interopérer lors du développement d'applications orientées service. Les applications SaaS sont construites par l'assemblage de services qui représentent des packages qui peuvent être déployés sur différents clouds. L'utilisateur cherche des workflows de services décrivant l'application souhaitée ou en crée de nouveaux. Les services cloud sont décrits suivant une ontologie cloud décrivant entre autres la manipulation des données et les schémas de communication entre les services cloud (ex : le routage des messages). Ainsi, les ressources infrastructures nécessaires au déploiement des composants du SaaS sont découvertes par un cloud broker. Ce dernier permet entre autres d'évaluer les ressources cloud, de négocier dynamiquement le SLA en fonction des besoins en qualité de service et du budget alloué à l'application, et de déployer les services sur un ou plusieurs clouds.

SOCCA requiert l'intervention de l'utilisateur au niveau de la construction ou de la recherche du workflow de services correspondant à l'application souhaitée.

Sun et al. [SWZ+10] décrivent SOSDC (Service Oriented Software Development Cloud), une plateforme cloud pour le développement en ligne de logiciels orientés service et un environnement d'hébergement dynamique pour ces derniers. SOSDC est une architecture qui couvre les trois niveaux des services cloud. Le niveau SaaS fournit un environnement de développement en ligne. Ce niveau comporte deux modules : service Xchange et MyCloud. Service Xchange permet le support des services Web partagés, où les utilisateurs peuvent publier, découvrir, ou invoquer un service. Ils peuvent aussi soumettre leurs feedbacks sur certains services en termes d'attributs fonctionnels et non

fonctionnels afin de mesurer entre autre la fiabilité des services. MyCloud est un environnement de développement personnel à chaque développeur, dans lequel ce dernier peut effectuer différentes tâches telle que la gestion de souscription à des services Web atomiques. Le niveau PaaS fournit "App Engine", un environnement d'hébergement dynamique pour les logiciels orientés service. Il permet de mettre en place et de gérer l'environnement de déploiement des applications en spécifiant entre autres les appliances logicielles (boitiers applicatifs, ex : serveur d'application). Le niveau IaaS fournit les ressources infrastructures comme le stockage et le réseau.

SOSDC requiert l'intervention du développeur dans toutes les étapes de développement et de déploiement de l'application.

Zhou et al. [ZAG+12] insèrent une nouvelle couche CaaS (Composition as a Service) entre les couches SaaS et PaaS. CaaS fournit aux utilisateurs un middleware basé sur le cloud pour la composition dynamique des services CM4SC (Cloud based Middleware for Dynamic Service Composition). CM4SC permet une découverte automatique, et une composition automatique et dynamique de services Web atomiques. CM4SC est développé comme un environnement de composition de services Web dans lequel les utilisateurs peuvent décrire leurs besoins en termes de tâches. En effet, l'utilisateur sélectionne une tâche parmi une liste de tâches présentes dans le répertoire de CM4SC. Une tâche est associée à un service atomique. Si l'utilisateur souhaite composer plusieurs services, il a la possibilité de choisir une autre tâche dans une seconde liste contenant les tâches correspondant aux services composables avec le premier sélectionné. Deux services sont composables si les sorties du service A correspondent aux entrées du service B. CM4SC transforme les besoins exprimés en termes de tâches par l'utilisateur en modèle de processus comprenant le service composite ainsi que les flux des données de ses services atomiques et exécute cette composition.

CM4SC utilise des services Web qui sont composés par le matching de leurs entrées/sorties et ne génère pas d'applications mais exécute plutôt un processus de tâches réalisées par des services Web.

Une architecture conceptuelle d'une plateforme qui permet d'exécuter des applications SaaS configurables et multi-tenants est proposée dans [KKH11]. Kang et al. utilisent une architecture dirigée par les métadonnées pour satisfaire la multi-tenancy des applications. La plateforme est composée de :

- Une application cliente : qui permet l'accès au SaaS au travers d'un navigateur Web et de router les requêtes/réponses entre l'utilisateur et l'application.

- Un configurateur : qui est une application Web utilisée par les locataires pour configurer les aspects configurables de l'application SaaS, à savoir : la structure organisationnelle (ex : contrôle d'accès), l'interface utilisateur, le modèle de données (ex : ajouter/supprimer des champs de données), le workflow (ex : réarranger l'ordre des activités), la logique métier (ex : créer une nouvelle opération sur la base de données qui satisfait les nouveaux besoins métier).

- Un moteur d'exécution : qui génère les applications multi-tenants spécifiques à un locataire en utilisant le codebase[1] de l'application et les méta-données de configuration.

- Un système de gestion des méta-données : qui fournit à la fois une API pour l'accès partagé à la base de données des méta-données, et des champs supplémentaires que chaque locataire peut personnaliser pour les objets de données qui ne sont pas pris en compte dans l'application SaaS de base.

- Les bases de données et un répertoire : la plateforme contient deux types de bases de données. Une base de données des méta-données pour le stockage des aspects configurés de l'application SaaS, et la base de données de l'application pour la gestion du codebase de l'application et des données générées par l'utilisateur. Le répertoire sert pour le stockage des workflows sous forme de fichier XML.

Ce travail permet la personnalisation d'une application SaaS existante par la configuration. Il ne permet pas la génération d'une application composite à la volée en fonction de la demande de l'utilisateur.

MODACLOUDS [ADNC+12] est un projet européen dont l'objectif est de diminuer le vendor lock-in (enfermement propriétaire) lors du développement d'applications cloud. Il fournit :

- Un environnement de conception permettant (i) de modéliser l'application souhaitée suivant le principe de Développement Dirigé par les Modèles (DDM), (ii) d'évaluer les solutions d'hébergement cloud suivant des caractéristiques non fonctionnelles (gestion des risques, coût, localisation géographique, performance etc), et (iii) de générer semi-automatiquement le code de l'application en développement en fonction du cloud ciblé.

- Un environnement d'exécution permettant le monitoring de l'application en cours d'exécution et de réagir aux fluctuations de performances en envisageant un redéploiement sur une autre solution cloud.

La conception suivant le modèle DDM permet de faire fonctionner les applications développées sur des plates-formes multi-clouds et la migration des applications d'un cloud à un autre. Cependant, cette approche nécessite la modélisation de la totalité de l'application en amont, et remet le choix final de la plateforme cloud au développeur (intervention humaine).

Le travail de [Hat] décrit "Software Engineering as a Service", une plateforme de conception et de déploiement de logiciels accessible sur le cloud. La plateforme est composée de trois modules :

[1] Ensemble du code source

- Le module de provisionnement : qui recommande le dimensionnement (nombre de noeuds, spécifications du serveur) des environnements de développement, de test, de simulation, et de production et déploie ces derniers. Le dimensionnement se base sur différents paramètres tels que la taille de l'application (ex : nombre de cas d'utilisation) et des logiciels à provisionner. Ce module permet aussi de sélectionner et d'installer des packages de développement spécifiques pour l'exécution du projet logiciel.

- Le module d'exécution : qui permet aux utilisateurs d'effectuer des activités de génie logiciel sur le cloud, tels que : la récolte des exigences, la conception de modules, et le codage.

- Le module des processus : ce module fournit les activités de support qui permettent le bon déroulement du processus de développement, tels que (i) les processus de gestion de projet; (ii) les processus qualité comme la vérification du code; (iii) les processus de gestion des intervenants; (iv) les processus de développement comme le contrôle des versions et la sauvegarde du code.

Software Engineering as a Service fournit un environnement de développement d'application sur le cloud en fournissant des outils aidant pour ce faire. Ce travail ne concerne pas la génération automatique d'applications orientées service.

Un environnement de création et provisionnement de services Web composites sur le cloud baptisé E2E cloud est décrit par Kamienski et al. [KSA+14]. Dans E2E cloud, l'utilisateur décrit un flux d'exécution ou bien une idée abstraite (incomplète) du service désiré sous forme d'entrées/sorties et des fonctions désirées. Dans ce deuxième cas, le service abstrait est passé au sous-module de composition automatique qui concatène les services disponibles afin de générer une composition complète qui respecte les besoins de l'utilisateur (entrées/sorties et fonctions). Le générateur de code traduit la composition en code déployable. Un middleware alloue les ressources infrastructures cloud (privé ou public) afin de remplir les objectifs techniques et financiers. Il permet aussi la gestion de l'élasticité et l'équilibrage de charge de l'application une fois déployée.

E2E cloud est un environnement de provisionnement automatique d'applications orientées service. Il repose sur le matching des entrées/sorties des services pour en connaître la composabilité. Cette démarche n'est pas applicable aux services métier considérés dans ce travail.

Tejedor et al. [TEL+11] présentent Service Superscalar (ServiceSs). ServiceSs fournit un modèle de programmation basé sur les tâches qui peuvent être des services Web ou des méthodes. Il fournit aussi un runtime d'exécution pour le développement et l'exécution des applications orientées service dans le cloud. Les développeurs de services créent des orchestrations complexes qui invoquent des services distants ou des méthodes locales par l'écriture de programme java séquentiel sans avoir à utiliser des APIs. Une interface permet la déclaration des services et méthodes en définissant leurs

noms, leurs paramètres d'entrées/sorties, les ports utilisés dans le cas d'un service, les contraintes de ressources dans le cas d'une méthode, etc. Le runtime orchestre automatiquement le déploiement élastique des méthodes dans le cloud, et l'exécution des services et méthodes. Le système de gestion des ressources de ServiceSs ne gère pas les ressources des services puisque ces derniers sont fournis par des entités externes. Il permet cependant de définir plusieurs localisations pour un service, et le nombre d'invocations concurrentes à ce dernier afin de prévenir les surchages. Lors de l'exécution d'une orchestration, les dépendances entre les différentes tâches sont identifiées et schématisées sous forme de graphe de dépendances. Ce dernier détermine le flux de contrôle et le flux de données entre les différentes tâches. ServiceSs est cloud-unaware puisque le développeur n'a pas besoin de spécifier d'informations relatives aux ressources pour le déploiement.

ServiceSs décrit un modèle de programmation pour les applications orientées service. Il requiert du développeur de connaître tous les services Web et méthodes nécessaires au développement de l'application désirée en plus du langage de programmation ServiceSs.

Tsai et al. [THS11] introduisent EasySaaS, un framework de développement de SaaS qui répond à la contrainte de multi-tenancy et qui diminue la charge de développement au niveau locataires. EasySaaS fournit une approche de personnalisation en fonction des besoins des locataires en offrant deux alternatives pour la construction de SaaS. La première permet aux locataires de publier les besoins et les spécifications de l'application, ainsi que les scripts de test, et laisse le soin aux fournisseurs de SaaS de personnaliser leur solution afin de répondre aux exigences des locataires. La seconde alternative permet aux locataires de chercher des services, des composants ou des templates réutilisables satisfaisant leurs besoins. Puis les locataires peuvent faire des personnalisations basées sur les templates, et créer de nouveaux services/composants par une composition.

EasySaaS ne traite pas le provisionnement automatique d'applications orientées service. Il permet au contraire à l'utilisateur de personnaliser des SaaS développés par des fournisseurs, ou de construire des services composites à partir de services existants. Les applications ne sont pas générées à la volée à partir du besoin de l'utilisateur.

3.3 Approches de développement et de déploiement d'applications cloud

Kommalapati et al. décrivent dans [KZ11] le cycle de vie de développement de SaaS. Les auteurs préconisent une approche qui favorise l'évaluation des capacités des plateformes cloud. Si le service SaaS en cours de développement n'est pas le premier dans l'organisation, les phases d'évaluation, de souscription et de fonctionnement seront

moins détaillées, en raison du travail qui aura déjà été fait lors du précédent développement. Les auteurs considèrent six phases dans le cycle de vie de développement d'un SaaS.

1. Projection : les nouvelles opportunités et besoins métier, et les applications qui peuvent bénéficier des caractéristiques du cloud sont identifiées. Une décision est prise entre "acheter ou développer".

2. Évaluation des plateformes : consiste à définir l'architecture technique et conceptuelle de l'application à développer, puis à évaluer les différentes plateformes afin de sélectionner celle qui permet de réaliser cette architecture. Les caractéristiques prises en compte pour l'évaluation sont entre autres : l'économie, les capacités, la sécurité, la fiabilité, la disponibilité, la scalabilité, les performances, et la reprise après sinistre.

3. Planning : établit le plan de projet, de ressources, et de communication et élabore une stratégie d'atténuation des risques.

4. Souscription : une souscription est acquise au niveau PaaS ou IaaS. Cette phase aboutit à la validation des stratégies de récupération et de sauvegarde, de reprise après sinistre, de gestion de souscription, et de support de production. Un SLA et les contrats de prix pour les services gérés (calcul, stockage...) sont négociés.

5. Développement : les spécifications sont traduites en code. Cette phase inclut la mise en place d'un environnement de développement. Le service est développé à travers un processus itératif où le déploiement et le test s'effectuent continuellement tout au long des itérations. Les politiques de sécurité de l'application et le processus de support et de helpdesk sont intégrés.

6. Fonctionnement : consiste à mettre en place les processus de déploiement du service développé avec les meilleures qualités possibles. Des activités d'évaluation des capacités requises, des tests de charge, des processus de sauvegarde et de récupération, de reprise après sinistre, et de continuité d'activité sont mises en place.

Ce cycle de vie de développement de SaaS intègre les changements dus à l'environnement cloud comme la découverte et souscription à une PaaS, la négociation de SLA, etc. Ce travail ne s'inscrit pas dans le cadre de l'automatisation du développement d'applications cloud.

Guha et al. [GAD10] adoptent et actualisent la méthode de développement agile "Extreme Programing" pour le développement d'applications cloud, et proposent Extreme Cloud Programming. Extreme Cloud Programming régit les interactions entre les ingénieurs et le fournisseur cloud sur les différentes phases (scénarios de l'application, planning, design, construction, et test) de développement d'applications. Les auteurs

attribuent à titre d'exemple, la tâche de comptabilité des ressources dans la plateforme cloud au fournisseur cloud. Ils préconisent aussi la collaboration avec le fournisseur cloud pour la conception de l'architecture logicielle, le mapping de l'architecture logicielle vers l'architecture hardware, la conception de l'interface, la conception des types de données, l'estimation du coût et la planification de l'estimation du projet.

Extreme Cloud Programming ne s'inscrit pas dans le cadre de l'automatisation du développement d'applications cloud.

Sledziewski et al. [SBA10] présentent une approche qui utilise un DSL (Domain Specific Language) dans le processus de développement et de déploiement de logiciels sur le cloud. Le but de ce travail est de fournir un framework pour le développement de DSL qui facilitera la modélisation et le déploiement des applications cloud. Cette approche repose sur deux phases fondamentales :

- La première phase concerne le développement d'un DSL pour un domaine donné, de manière à faciliter la modélisation d'une application. Le DSL conçu dans cette phase sera disponible aux concepteurs d'applications sous la forme de SaaS afin de permettre une meilleure accessibilité. Cette phase implique deux étapes :
  * La définition du méta-modèle avec les experts du domaine.
  * Les spécifications des templates permettant la génération automatique de code depuis les modèles.
- Dans la deuxième phase, le langage DSL est utilisé par les concepteurs d'applications pour modéliser les applications. Ces modèles sont traduits automatiquement en code exécutable spécifique à une plateforme cloud cible, puis, ce code est déployé automatiquement sur la plateforme cloud.

Ce travail automatise la génération de code exécutable et le déploiement de l'application conçue en utilisant le DSL. Il nécessite toutefois la connaissance du DSL pour la conception de l'application désirée.

Chou et al. [CO12] proposent un modèle baptisé "orange model" qui considère le cycle de développement d'applications cloud intégrées, en utilisant une approche incrémentale, et itérative. Le principal objectif est d'assurer une intégration sécurisée. La collaboration dans le "orange model" considère trois types de réunions entre l'équipe métier, l'équipe IT, et l'équipe sécurité : la réunion de démarrage, la réunion de revue après chaque phase, et la réunion de mise en production. Les phases suivantes représentent le cycle de développement :

1. Détermination des besoins : consiste à déterminer les objectifs et besoins métier et à les convertir en besoins IT.
2. Planning de projet et de sécurité : consiste à déterminer les besoins en sécurité, présélectionner les SaaS candidats, et proposer un plan de projet.

3. Analyser les candidats et leurs risques : cela passe par des tests de vulnérabilité des SaaS candidats, et l'analyse de l'intégration d'un SaaS au système interne de l'entreprise.

4. Choisir une solution : cette phase passe par l'évaluation des candidats (comparaison) et la sélection du candidat à intégrer, la mise en place de plans d'amélioration de la sécurité, l'identification des parties du système interne qui doivent être évaluées et testées après que l'intégration ait été effectuée, et la négociation et la signature du contrat avec le fournisseur de SaaS.

5. Implémentation et intégration du SaaS au système interne de l'entreprise.

6. Test et évaluation de la sécurité : les vulnérabilités du système intégré sont évaluées par des tests d'intrusion externe (pour simuler un hacker externe) et interne (pour vérifier quelles vulnérabilités pourraient être exploitées quand un attaquant a pénétré le système).

7. Mise en fonctionnement du système et amélioration de la qualité : après la mise en fonctionnement du système intégré, ce dernier est évalué. En fonction de la revue d'évaluation du système, une mise à jour des politiques de sécurité est effectuée.

8. Ajouter une nouvelle application SaaS durant le fonctionnement : cette phase permet d'intégrer un nouveau SaaS au système en fonctionnement répondant à de nouvelles fonctionnalités requises par l'entreprise en suivant les différentes phases d'intégration (phases 1 à 7) décrites dans le "orange model".

9. Arrêt du service : dans le cas où le service doit être arrêté pour être remplacé, un nouveau plan de solution est mis en place, et les risques sont de nouveau évalués pour la solution de remplacement. Après la phase d'implémentation et de tests, les données sont transférées vers le nouveau SaaS à intégrer et l'équipe sécurité s'assure que les données ont bien été supprimées de l'ancien SaaS. Enfin le service est arrêté. Si le service est arrêté sans intention de remplacement, les données sont supprimées de ce SaaS et le service est arrêté.

Le "orange model" est un modèle de développement de SaaS intégrés dont l'objectif est d'assurer une intégration sécurisée. Toutes les phases de ce modèle requièrent l'intervention d'un ou de plusieurs intervenants technique, métier, et sécurité.

Beaucoup de travaux se sont penchés sur la question du développement de SaaS multi-tenants [ACK12], [LKH12], [KKH11], [LZZ+11], [RARV12], [LZL+10], [THS11]. Ces travaux permettent la personnalisation des applications cloud à partir des données de configuration de chaque locataire en fonction de ses besoins. La plupart de ces travaux sont dirigés par les méta-données pour stocker les informations de configuration [ACK12], [LKH12], [KKH11]. Ces méta-données de configuration de chaque

locataire sont ensuite utilisées pour générer et compiler une application personnalisée. La configurabilité des SaaS peut concerner le niveau présentation (interfaces de l'application) [LZZ+11], [KKH11], le modèle de données [KKH11], ou encore le partage d'applications avec les autres locataires [RARV12] (i.e., les utilisateurs peuvent choisir s'ils souhaitent, et avec qui ils souhaitent partager l'application).

Ces travaux répondent à la caractéristique multi-tenants des SaaS. Ils permettent la configuration et la personnalisation de SaaS développés par les fournisseurs. La personnalisation de SaaS est automatique. Ces travaux ne se penchent cependant pas sur le provisionnement d'applications à partir d'un besoin exprimé par l'utilisateur.

3.4 Analyse des travaux de développement et de déploiement des applications sur les environnements cloud

Les travaux cités dans ce chapitre décrivent des approches et environnements permettant le développement et le déploiement d'applications cloud. Ils visent des objectifs divers et variés tels que le respect de l'aspect multi-tenants des applications cloud, la diminution du vendor lock-in, etc.

L'objectif de notre travail est de diminuer les connaissances techniques et l'intervention humaine pour le provisionnement d'applications métier sur le cloud. De ce fait, nous avons choisi d'analyser dans les tableaux 3.1 et 3.2 les travaux cités dans ce chapitre décrivant respectivement des environnements de développement et de déploiement d'applications cloud, et des approches de développement et de déploiement d'applications cloud selon les critères suivants :

C1 : Classification du travail en fonction de son type.

C2 : Les phases dans lesquelles l'intervention de l'utilisateur est nécessaire.

C3 : Phases ou tâches automatisées.

C4 : Type des besoins introduits par l'utilisateur lorsque ce dernier n'a pas à développer l'application désirée par lui même.

La plupart des travaux décrivant des environnements de développement et de déploiement d'applications cloud ne permettent pas l'automatisation de la génération d'applications orientées service, ou de services composites sauf deux travaux sur les 9 étudiés : CM4SC [ZAG+12], et E2E cloud [KSA+14]. Cependant, ces derniers utilisent des services Web qui sont composés par le matching de leurs entrées/sorties. Cette technique d'étude de la composabilité (matching des entrées/sorties) ne peut être appliquée aux services métier. D'un autre côté, CM4SC nécessite la connaissance des opérations à exécuter puisque le choix de ces dernières est effectué sur la base de listes de services

qui se suivent et dont le contenu dépend du service sélectionné dans la liste précédente. Donc une idée précise des services atomiques à instancier est requise.

Tableau 3.1: Comparaison des environnements de développement et de déploiement d'applications cloud

Travaux	**C1 : Type**	**C2 : Intervention de l'utilisateur**	**C3 : Tâches automatisées**	**C4 : Besoins**
SOCCA [TSB10]	Architecture de construction d'applications cloud	Au niveau de la construction de la composition de services	La découverte de ressources cloud	L'utilisateur cherche des workflows de services décrivant l'application souhaitée ou en crée de nouveaux
SOSDC [SWZ+10]	Environnement de développement d'applications cloud	Au niveau du développement et du déploiement de l'application	-	-
CM4SC [ZAG+12]	Environnement de composition automatique et dynamique de services Web	Au niveau de l'expression du besoin	La composition de services Web	Les fonctionnalités représentant les opérations à composer
Architecture conceptuelle d'exécution de SaaS configurables [KKH11]	Environnement d'exécution de SaaS	Le fournisseur de SaaS intervient lors du développement de ce dernier et l'utilisateur de SaaS intervient lors de la configuration de ce dernier	Personnalisation de l'application	Les données de configuration
MODA-CLOUDS [ADNC+12]	Environnement de développement et d'exécution d'applications	Au niveau du développement de l'application	La génération de code est semi-automatique	-
Software Engineering as a Service [Hat]	Environnement de développement d'applications cloud	Au niveau du développement et du déploiement de l'application	Le dimensionnement (nombre de noeuds, spécifications du serveur) des environnements de production et de tests	-
E2E cloud [KSA+14]	Environnement de provisionnement d'applications cloud	Au moment de la description du besoin	La transformation des besoins de l'utilisateur en flux exécutable puis en code déployable	Les fonctionnalités désirées et leurs entrées/sorties
Service Superscalar [TEL+11]	Modèle de programmation	Au moment du développement	Le déploiement de l'application	-
EasySaaS [THS11]	Framework de développement de SaaS	Première alternative : lors de l'élicitation du besoin. Deuxième alternative : lors de la découverte et composition de services, et lors de la personnalisation de l'application	-	Première alternative : spécifications de l'application, scripts de test. Deuxième alternative : services composites, templates personnalisés

Un seul travail parmi ceux proposant des approches de développement et de déploiement d'application cloud automatise la génération des applications cloud, c'est celui du développement basé sur les DSL [SBA10]. Ce travail construit un DSL en amont afin de permettre à des intervenants métier de concevoir et déployer leurs applications; donc la conception de l'application désirée est fortement dépendante de l'existence et de la connaissance du DSL.

Tableau 3.2: Comparaison des approches de développement et de déploiement d'applications cloud

Travaux	**C1 : Type**	**C2 : Intervention de l'utilisateur**	**C3 : Tâches automatisées**	**C4 : Besoins**
Cycle de vie de développement de SaaS [KZ11]	Cycle de vie	Au moment du développement et du déploiement	-	-
Extreme cloud programming [GAD10]	Méthode agile pour le cloud	Au moment du développement et du déploiement	-	-
Développement basé sur les DSL [SBA10]	Approche de développement et de déploiement d'applications cloud	Les développeurs interviennent au moment de la création du DSL et les intervenants métier interviennent au moment de la conception de l'application	La transformation des modèles d'applications en code exécutable spécifique à un cloud cible, puis le déploiement de l'application générée	Les utilisateurs introduisent le modèle conceptuel de l'application par l'utilisation de DSL
Orange model [CO12]	Modèle de développement de SaaS intégrés	Dans toutes les phases du modèle. Orange model nécessite la collaboration de plusieurs intervenants (technique, métier, et sécurité)	-	-
[ACK12], [LKH12], [KKH11][LZZ+11], [RARV12], [THS11], [LZL+10]	Approches de personnalisation des applications cloud	Lors de la configuration du SaaS	Personnalisation de l'application	Données de configuration

3.5 Conclusion

Il ressort de notre analyse de l'état de l'art que seuls trois travaux parmi ceux étudiés permettent l'automatisation du provisionnement (développement et déploiement) d'applications cloud. Deux des trois travaux [ZAG+12] [KSA+14] génèrent des applications en composant automatiquement des services Web par le matching de leurs entrées/sorties. Cette technique de vérification de la composabilité des services n'est pas applicable aux services métier. Le troisième travail [SBA10] permet à des intervenants métier de concevoir leurs applications en utilisant un DSL développé en amont. Le modèle d'application était ensuite automatiquement traduit en code exécutable et déployé sur le cloud. Dans ce travail, la conception de l'application était très dépendante de la connaissance et de la disponibilité du DSL.

Notre travail s'inscrit dans l'optique de mettre en place une méthodologie et un environnement de provisionnement d'applications cloud orientées service. Il automatise la mise en place de l'application composite à partir d'un besoin exprimé par l'utilisateur à un haut niveau d'abstraction des détails techniques de développement (en termes de fonctionnalités, préférences de déploiement et de coût, et contraintes de qualité de service). Il repose d'une part sur la composition automatique de services métier qui

sera possible grâce à la description des relations de ces derniers, et d'autre part sur l'automatisation du déploiement de l'application composite générée. Cette automatisation permet de réduire l'intervention humaine et les connaissances techniques requises de l'utilisateur pour le provisionnement d'applications métier. En effet, l'intervention de l'utilisateur n'a lieu que pour (i) l'expression des besoins, (ii) l'introduction des données de configuration pour personnaliser l'application générée, et (iii) l'utilisation de l'application automatiquement générée et déployée.

Partie II

Contributions

Chapitre 4

Description des services de la marketplace et des besoins de l'utilisateur

Sommaire

4.1 Introduction

Le provisionnement d'applications orientées service dans le cloud, requiert de s'intéresser au cycle de vie de provisionnement dans son intégralité, qui va de l'expression du besoin en terme de fonctionnalités, à la réalisation de l'application, et son déploiement. Pour cela, en première étape nous allons constituer une marketplace de services métier et de services IaaS. Dans ce chapitre nous décrivons les services constituant la marketplace en utilisant le langage Linked USDL. Ce choix a été guidé par le fait que parmi les travaux analysés dans l'état de l'art (cf Chapitre 2), Linked USDL était celui permettant la description d'un grand nombre de services (métier, cloud, Web) alors que les autres langages étaient spécifiques. De plus, Linked USDL offre une grande couverture des aspects métier, techniques et opérationnels des services. Aussi, Linked USDL est basé sur le Linked Data, i.e, il réutilise plusieurs vocabulaires liés, ce qui facilite son extension. Nous réutilisons des concepts des modules usdl-core et usdl-price de Linked USDL pour décrire, entre autres, les fonctionnalités d'un service métier et le coût de déploiement d'un service métier sur une IaaS. Cependant, Linked USDL ne décrit pas les relations potentielles d'un service. Il décrit plutôt les relations établies entre services constituant une offre de service. Nous étendons alors ce langage pour décrire les relations entre un service et un autre afin de connaître la composabilité d'un service métier. Afin d'automatiser le déploiement et la configuration de services métier, nous décrivons les ressources nécessaires au déploiement de chaque service ainsi que ses paramètres configurables. Nous décrivons les paramètres de QoS d'un service donné sur la base de ses précédentes invocations et non pas sur le niveau de qualité exigé comme cela est possible avec le SLA.

Nous décrivons ensuite les besoins de l'utilisateur pour le provisionnement d'applications métier orientées service sur le cloud. Afin de minimiser les connaissances techniques requises de l'utilisateur pour ce provisionnement, nous avons choisi de permettre à l'utilisateur d'exprimer son besoin via un formulaire Web, afin qu'il n'ait pas besoin d'un logiciel en particulier ni de connaître une notation ou un langage particuliers. De plus, grâce aux concepts décrivant les relations d'un service, nous limitons les connaissances requises de l'utilisateur. En effet, ce dernier décrit les fonctionnalités désirées sans avoir à se préoccuper des services associés ni de la composabilité de ces derniers. Le vocabulaire RIVAL (Requirement Vocabulary) que nous définissons formalise les besoins de l'utilisateur en termes de fonctionnalités désirées, de besoins en qualité de services, et de préférences utilisateur.

La Figure 4.1 permet de positionner les descriptions des services de la marketplace et des besoins de l'utilisateur par rapport à MADONA (Methodology for AutomateD prOvisioning of cloud-based service-oriented busiNess Applications), la méthodologie de provisionnement d'applications orientées services que nous proposons. Cette dernière est présentée dans le Chapitre 5.

Ce chapitre est organisé comme suit : d'abord un exemple illustratif est introduit

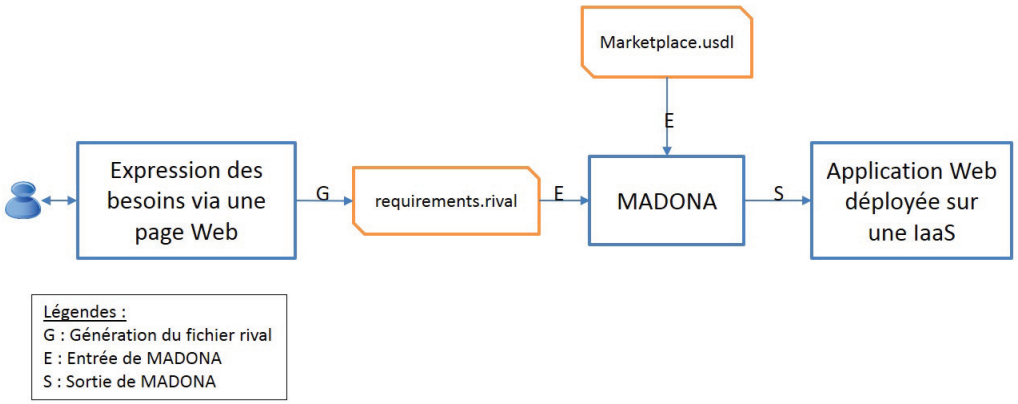


Figure 4.1: Entrées/sorties de MADONA

dans la Section 4.2 permettant d'étayer les phases de MADONA, mais aussi la description des services de la marketplace et des besoins de l'utilisateur. Dans la Section 4.3, les concepts réutilisés de Linked USDL et ceux nouvellement introduits pour la description des services de la marketplace sont décrits. Nous décrivons aussi les services nécessaires à l'illustration de l'exemple illustratif. La Section 4.4 décrit les concepts utilisés pour la description des besoins de l'utilisateur en vue du provisionnement d'applications métier orientées service sur le cloud. Une illustration de la description des besoins de l'utilisateur est aussi décrite pour le scénario introduit en Section 4.2.

4.2 Scénario illustratif

Nous considérons le scénario suivant pour illustrer ce travail. Un propriétaire d'une petite entreprise d'organisation d'événements (que nous appellerons ci-après l'utilisateur) souhaite mettre en place un système de gestion de projet. L'utilisateur a aussi besoin d'un Service de Gestion de Versions (SGV) afin que les intervenants puissent stocker, retrouver, et fusionner les différentes versions des étapes du développement d'un événement. Il souhaite aussi que le gestionnaire de projet ait une fonctionnalité de gestion de plannings. Un service de gestion de projet particulier peut nécessiter pour son bon fonctionnement d'autres services, qui à leur tour peuvent être dépendants d'autres services pour fonctionner. Il peut être composé avec un SGV particulier ou peut n'avoir aucune possibilité de composition avec un SGV. Nous ne nous attendons pas à ce que l'utilisateur connaisse ou spécifie ces détails techniques. Au lieu de cela, l'utilisateur se concentre sur ses besoins à un haut niveau, aussi bien fonctionnels (ex : systèmes de gestion de projet et de gestion de versions), que non fonctionnels (ex : coût de l'application).

Dans une situation classique, l'utilisateur irait (i) chercher sur le Web les différents systèmes de gestion de projets, de gestion de versions, et de gestion de plannings, (ii) analyser les différentes possibilités afin de choisir celle qui correspond le plus à son besoin fonctionnel et non fonctionnel, (iii) effectuer les paramétrages et change-

ments nécessaires afin que les trois composants puissent fonctionner ensemble, (iv) préparer l'environnement d'installation, puis (v) procéder au déploiement. Cette petite entreprise ne possédant pas de Direction des Systèmes d'Information (DSI) et possédant peu de moyens doit trouver une solution pour mettre à la disposition de ses employés ce système de gestion de projet en un temps minimum et avec peu de connaissances techniques. En effet, l'utilisateur ignore les détails techniques de l'application désirée (quelle application choisir, de quels modules est-elle composée, comment la déployer, sur quel type de système d'exploitation, a t-elle besoin d'une base de données, si oui de quel type, etc). Nous souhaitons permettre à l'utilisateur de ne se concentrer que sur ses besoins fonctionnels et non fonctionnels de haut niveau. Les besoins fonctionnels comprennent les fonctions de l'application métier désirée. Les besoins non fonctionnels incluent les préférences de l'utilisateur et les paramètres de qualité de service. L'utilisateur repose sur une bonne expérience précédente de l'utilisation d'"Amazon" [amab] et préfère héberger son application sur cette même infrastructure, en Europe. Il préfère aussi : payer les coûts d'utilisation de l'application avec la monnaie européenne "euro", et que le coût soit inférieur à 30 euros par mois.

4.3 Extension du langage Linked USDL pour la description des services de la marketplace

Les services de la marketplace (services métier et IaaS) sont décrits par leurs fournisseurs. Chaque service métier a un script de déploiement, de personnalisation et un script supplémentaire pour sa mise en relation avec d'autres services métier. Les scripts de mise en relation permettent la configuration d'un service métier afin qu'il soit en mesure de travailler avec un autre. A titre d'exemple, un script liant un moteur de Wiki et une base de données permet de créer les tables de la base de données et communiquer son adresse IP au service l'utilisant (moteur de Wiki).

Nous avons soulevé dans le Chapitre 2, des travaux qui permettaient de décrire les relations établies et potentielles d'un service. Aucun de ces travaux ne permettait de décrire à la fois les contraintes et possibilités de composition de manière exhaustive. Pour pallier ces manquements, nous étendons le langage Linked USDL pour décrire les relations d'un service avec de nouvelles propriétés. La Figure 4.2 illustre le langage Linked USDL étendu. Nous décrivons dans ce qui suit les concepts réutilisés de Linked USDL, et ceux que nous avons défini pour enrichir le modèle.

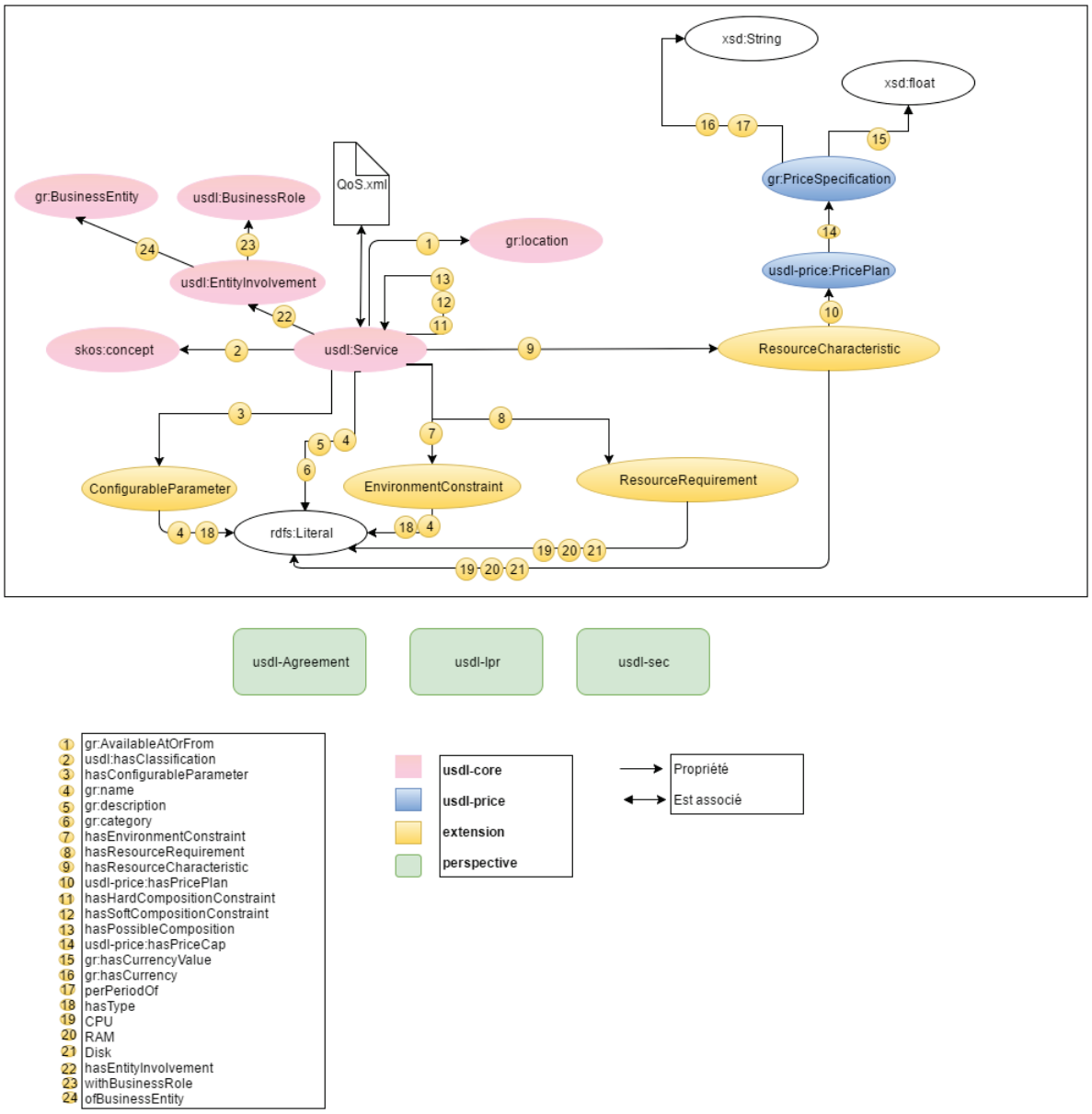


Figure 4.2: Extension du Linked USDL pour la description des services de la marketplace

4.3.1 Les concepts de Linked USDL réutilisés

Nous réutilisons les concepts suivants de Linked USDL pour la description des services de la marketplace. Certains concepts sont utilisés pour décrire aussi bien les services métier que les IaaS, d'autres seront spécifiques aux IaaS.

4.3.1.1 Les concepts communs à la description des services métier et IaaS

- La classe "skos:concept" utilisée dans le module usdl-core pour spécifier le type du service décrit. Dans ce travail, nous considérons deux types de services : les services métier qui sont composés pour former une application métier, et les IaaS qui fournissent un environnement de déploiement pour l'application métier.
- La classe "usdl:Service" utilisée dans le module usdl-core pour décrire un service. Nous ajoutons les propriétés "gr:name" (propriété 4 de la Figure 4.2) et "gr:description" (propriété 5 de la Figure 4.2) qui permettent respectivement de représenter le nom du service ainsi que sa description en langage naturel.
- La classe "usdl:EntityInvolvement" qui permet de décrire les entités métier "gr:BusinessEntity" impliquées dans le service, ainsi que leurs rôles "usdl:BusinessRole". Nous réutilisons cette classe pour associer un fournisseur à un service.

4.3.1.2 Les concepts spécifiques à la description des IaaS

- La classe "gr:location" du vocabulaire Good Relations utilisée dans le module usdl-core permet de décrire la localisation d'une offre de service. Nous réutilisons cette classe pour décrire aussi la localisation d'un service. Les lois régissant les différents pays détenant les centres de données protègent la confidentialité des données de manière différente. A titre d'exemple, le USA Patriot Act autorise le gouvernement américain à accéder aux données numériques des utilisateurs (particuliers et entreprises) dès lors que ces dernières sont stockées aux États Unis. Dans notre travail, la localisation d'un service ne concerne que les IaaS. En effet, les services métier représentent des packages déployables sur différentes infrastructures, leur localisation avant leur déploiement n'a pas d'impact sur la gestion de la confidentialité des données.
- Les classes "gr:PriceSpecification" et "usdl-price:PricePlan" définies dans le module usdl-price pour décrire respectivement le coût associé à une offre de service et ses différentes options d'achat. Nous avons réutilisé ces classes pour décrire aussi le coût d'un service. Le coût que l'on a considéré dans ce travail représente une estimation du coût d'utilisation d'un service métier sur une infrastructure cloud, c'est à dire le coût d'allocation des ressources virtuelles pour le déploiement et l'utilisation de ce service métier (ex : nombre de CPU, RAM, Disque, bande passante, etc). Chaque fournisseur IaaS attribue un coût à chaque option d'achat. Nous décrivons pour une option d'achat son coût maximal (propriété 14 de la Figure 4.2). Nous avons détaillé les spécifications de prix grâce aux propriétés "gr:hasCurrencyValue", "gr:hasCurrency", et "perPeriodOf" qui permettent de décrire respectivement le coût maximal, la monnaie, et la période de facturation.

4.3.2 Enrichissement de Linked USDL avec de nouveaux concepts

Nous avons ajouté des concepts à Linked USDL permettant de décrire les relations de composition d'un service métier, ses contraintes de déploiement, ses paramètres configurables, sa catégorie, les caractéristiques techniques d'un service IaaS, et la description QoS d'un service métier et IaaS. Dans ce qui suit, chaque nouveau concept est motivé, défini, et illustré par un exemple.

4.3.2.1 Les relations de composition

Un service métier S1 peut être lié à un autre service métier S2 créant ainsi une relation de composition. Cette dernière inclut les contraintes et possibilités de composition (Définitions 1 et 2) tel qu'illustré sur la Figure 4.3.

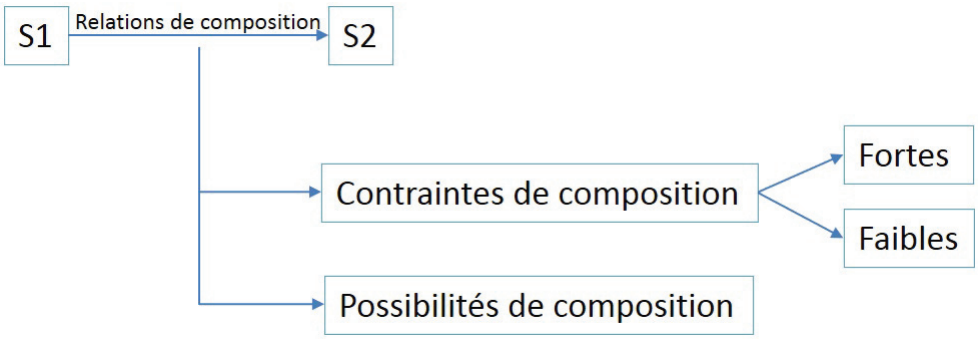


Figure 4.3: Relations d'un service

4.3.2.1.1 Contraintes de composition

Certains services ont besoin d'autres services pour fonctionner correctement.

Définition 1. Les contraintes de composition lient le service métier décrit à un autre. Elles représentent les services métier sans lesquels le service décrit ne peut fonctionner. Les contraintes de composition peuvent être *fortes* ou *faibles*.

- Les contraintes fortes (propriété 11 de la Figure 4.2) imposent des services qui doivent être composés à celui décrit. Par exemple, MediaWiki doit être composé avec une base de données MySQL.
- Les contraintes faibles (propriété 12 de la Figure 4.2) offrent le choix de sélectionner un et un seul service dans une famille de services métier fournissant la même fonctionnalité. Par exemple, Joomla doit être composé avec une base de données MSSQL, PostgreSQL, ou MySQL.

4.3.2.1.2 Possibilités de composition

L'AOS repose sur la composition de services. Lorsqu'on compose des services Web, nous pouvons évaluer le matching des entrées/sorties de deux services à composer pour savoir s'ils sont composables, i.e., si les sorties du service A sont compatibles avec les entrées du service B. Avec les services métier, cette approche n'est pas applicable. Dans les travaux cités dans le Chapitre 2, les possibilités de composition n'étaient pas représentées comme telles. Les auteurs décrivaient plutôt les différentes compositions passées d'un service [MWB+11] [Car13]. Ces dernières ne font pas la distinction entre les contraintes et les possibilités de composition et ne permettent pas une description exhaustive des relations potentielles d'un service. Pour pallier ce problème, nous décrivons pour chaque service métier ses possibilités de composition.

Définition 2. Les possibilités de composition (propriété 13 de la Figure 4.2) lient le service métier décrit à un autre. Un service métier S1 est une possibilité de composition du service S2 si et seulement si S1 peut être composé avec S2 et S2 fonctionne correctement s'il n'est pas composé avec S1. Par exemple, OpenStack fonctionne correctement sans tableau de bord, mais le tableau de bord "Horizon" d'OpenStack représente une possibilité de composition des composants de base d'OpenStack.

4.3.2.2 Les contraintes de déploiement

Les contraintes de déploiement représentent des contraintes liées aux outils logiciels et aux ressources nécessaires au déploiement d'un service métier.

4.3.2.2.1 Contraintes d'environnement

Les services métier représentent des packages déployables et non pas des services déjà installés chez le fournisseur. Chaque service fonctionne dans un environnement qui lui est propre et nécessaire. Cette notion a été décrite dans le travail de Nguyen et al. [NLPvdH12] mais regroupée avec les contraintes de ressources et de composition. A des fins d'automatisation, nous définissons les contraintes d'environnement et de ressources pour chaque service métier.

Définition 3. Les contraintes d'environnement (propriété 7 de la Figure 4.2) d'un service métier représentent les logiciels qui doivent être installés sur la même machine virtuelle que le service métier décrit. Chaque contrainte d'environnement est décrite par son type (ex : serveur Web) et son nom (ex : apache). Nous supposons que les contraintes d'environnement sont automatiquement intégrées dans les scripts de déploiement d'un service métier.

4.3.2.2.2 Contraintes de ressources

Chaque service métier nécessite un minimum de ressources permettant son bon fonctionnement.

Définition 4. Les contraintes de ressources (propriété 8 de la Figure 4.2) d'un service métier représentent les caractéristiques exigées pour la machine virtuelle hébergeant ce service en termes de CPU (propriété 19 de la Figure 4.2), de mémoire (propriété 20 de la Figure 4.2), et de disque (propriété 21 de la Figure 4.2).

4.3.2.3 Les caractéristiques techniques des services IaaS

Afin de permettre la sélection d'un service IaaS répondant aux contraintes de déploiement d'un service métier, nous définissons pour chaque service IaaS (ex : Amazon EC2) les caractéristiques techniques des instances qu'il offre grâce à la propriété 9 de Linked USDL étendu. A ces caractéristiques techniques est associé un plan de coût (propriété 10 de la Figure 4.2) permettant de décrire le coût d'un type d'instance de machine virtuelle pour un fournisseur donné.

4.3.2.4 Les paramètres configurables

Afin d'automatiser la configuration de l'application métier générée par MADONA, nous définissons pour chaque service métier ses paramètres configurables.

Définition 5. Les paramètres configurables (propriété 3 de la Figure 4.2) d'un service métier représentent les paramètres qui peuvent être personnalisés pour l'utilisation du service, tels que, le nom de l'application, le logo etc. Chaque paramètre configurable est décrit par son nom ("gr:name", propriété 4 de la Figure 4.2) et par le type du composant html ("hasType", propriété 18 de la Figure 4.2) à insérer sur le formulaire Web de configuration (ex : une zone de texte pour le nom d'une application, et un bouton parcourir ou une zone de texte pour le logo de cette dernière).

4.3.2.5 La catégorie d'un service métier

Nous définissons pour chaque service métier sa catégorie par l'utilisation de la propriété "gr:category" (propriété 6 de la Figure 4.2). Une catégorie représente la famille de services à laquelle un service métier appartient. A titre d'exemple, le service MediaWiki appartient à la famille "moteur de Wiki". Cette notion sera utilisée dans MADONA pour noter la qualité des services. En effet, la notation de la qualité d'un service se fait en le comparant aux services de la même catégorie.

4.3.2.6 Les paramètres QoS

Nous utilisons dans ce travail une approche différente de celle de Linked USDL pour la description des paramètres de qualité des services de la marketplace. En effet, Linked USDL permet de décrire le SLA (niveau de qualité visé) liant le client et le fournisseur. Nous avons choisi au contraire de confier la tâche de description des paramètres de qualité de service à un service tiers permettant ainsi d'avoir une évaluation objective et représentative des paramètres QoS d'un service donné en fonction de ses invocations passées (niveau de qualité obtenu par le passé). Nous considérons que l'historique d'invocation des services est fourni par un service tiers. Ces services tiers d'évaluation et de comparaison de services prolifèrent sur le Web. Cloud Armor [cloa] et Cloudorado [clod] fournissent de tels services. Le premier fournit un dataset de rangs QoS (disponibilité, temps de réponse, facilité d'utilisation, etc) affectés par des utilisateurs aux services cloud utilisés. Le second fournit une comparaison des fournisseurs cloud en termes de niveau SLA, de prix, et de fonctionnalités. Nous allons donc confier la description des paramètres QoS des services de la marketplace à un service tiers externe (ex : service Web), qui retournera les QoS d'un service (sous forme de fichier XML) à partir de son nom. Dans ce travail, nous considérons quatre paramètres de qualité de services, à savoir : le respect de la confidentialité des données manipulées par le service, la préservation contre la perte des données manipulées par le service, la disponibilité du service, et le temps de réponse du service. Le Listing 4.1 illustre un exemple de fichier XML retourné, décrivant les paramètres QoS d'un service.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <QoS>
3     <Availability>7</Availability>
4     <Response_Time>173.82358</Response_Time>
5     <Data_Privacy>8</Data_Privacy>
6     <Data_Loss>8.902313</Data_Loss>
7 </QoS>
```

Listing 4.1: Fichier XML décrivant les paramètres QoS d'un service

4.3.3 Description des services pour l'exemple illustratif

Nous allons illustrer l'exemple décrit en Section 4.2, où rappelons le, un utilisateur souhaite mettre en place une application de gestion de projet composée avec un service de gestion de versions, et un service de gestion de plannings. Afin de mieux illustrer le langage Linked USDL étendu, nous avons choisi de décrire un service métier et un service IaaS. Le Listing 4.2 illustre la description de ProjectMan, un service métier de gestion de projet. Le Listing 4.3 illustre la description de l'IaaS Amazon Europe qui est l'infrastructure choisie par l'utilisateur pour le déploiement de son application. Les descriptions des services de la marketplace sont stockées dans le fichier marketplace.usdl. Cela permet de faire une recherche simultanée des services répondant aux fonctionnalités primaire et secondaires de l'utilisateur (cf. Section 5.2.1).

ProjectMan a quatre possibilités de composition (lignes 18-29 du Listing 4.2) avec respectivement MyVCS1 et MyVCS2, des services de gestion de versions; GP1 un service de gestion de plannings; et memcached, un service de gestion de mémoire cache. Il a deux contraintes de composition (lignes 11-16) avec respectivement la base de données MySQL et MyCRM, un service CRM qui permet d'avoir une vision sur les relations entre les clients et les employés qui gèrent leurs évènements. ProjectMan requiert pour son installation une machine virtuelle avec les caractéristiques suivantes : 1 CPU, 1 Go de RAM et 10 Go de disque (lignes 31-35). Il peut être personnalisé par son nom et son logo (lignes 37-44).

```
1  <http://mydomain.fr/usdls/projectman> a usdl:service;
2
3  gr:name"ProjectMan";
4
5  gr:description "ProjectMan is a project management engine.";
6
7  gr:category "Project management";
8
9  usdl:hasClassification businessservice;
10
11 usdl:hasHardCompositionConstraint
12 [   a usdl:service ;
13     gr:name "MySQL"],
14
15 [   a usdl:service ;
16     gr:name "MyCRM"];
17
18 usdl:hasPossibleComposition
19 [   a usdl:service;
20     gr:name "MyVCS1"],
21
22 [   a usdl:service;
23     gr:name "MyVCS2"],
24
25 [   a usdl:service;
26     gr:name "GP1"],
27
28 [   a usdl:service;
29     gr:name "memcached"];
30
31 usdl:hasResourceRequirement
32 [  a usdl:ResourceRequirement;
33    usdl:CPU "1";
34    usdl:Memory "1G";
35    usdl:Disk "10G"];
36
37 hasConfigurableParameter
38 [  a usdl:ConfigurableParameter;
39    gr:name "name";
40    usdl:hasType "textarea"],
41
42 [  a usdl:ConfigurableParameter;
43    gr:name "logo";
44    usdl:hasType "textarea".
```

Listing 4.2: Description du service ProjectMan avec le langage Linked USDL étendu

Afin de mieux illustrer les différentes phases de MADONA, nous considérons que dans notre scénario MyCRM à son tour, a besoin de communiquer avec EmployMan (un service de gestion des employés) afin d'identifier les employés impliqués dans un projet. MyCRM, EmployMan, et MyVCS requièrent une base de données MySQL.

L'IaaS d'Amazon Europe décrite dans le Listing 4.3 est un service fourni par Amazon (lignes 5-8). Cette IaaS offre des instances de machines virtuelles qui sont déploy-

ables en Europe (ligne 12). Nous illustrons dans ce Listing la description .usdl d'un seul type d'instance. La description des caractéristiques des ressources de cette instance est faite de la ligne 17 à 30. La description du coût de cette instance est effectuée de la ligne 22 à 29.

Le fichier marketplace.usdl sert d'entrée à la phase de découverte de services métier et à la phase de découverte d'IaaS.

```
1  <http://mydomain.fr/usdls/AmazonEurope> a        usdl:service ;
2
3         gr:name "Amazon Europe" ;
4
5         usdl:hasEntityInvolvement
6         [ a EntityInvolvement;
7           usdl:ofBusinessEntity Amazon;
8           usdl:withBusinessRole Provider ];
9
10        usdl:hasClassification IaaS;
11
12        gr:availableAtOrFrom Europe;
13
14        gr:description "Amazon Elastic Compute Cloud (Amazon EC2) provides
15        resizable compute capacity in the cloud." ;
16
17       usdl:hasResourceCharacteristic
18       [a ResourceCharacteristic;
19           CPU "1";
20           RAM "1Go";
21           Disk "10Go";
22           usdl-price:hasPricePlan
23           [ a usdl-price:PricePlan;
24               usdl-price:hasPriceCap
25               [ a usdl-price:PriceSpecification
26                    gr:hasCurrency "euro";
27                    gr:hasCurrencyValue "5";
28                    perPeriodOf "month"],
29          ];
30       ].
```

Listing 4.3: Description d'Amazon Europe avec le langage Linked USDL étendu

4.4 Description des besoins de l'utilisateur pour le développement d'applications orientées service sur le cloud

La description du besoin implique la récolte, la transformation et le traitement des besoins de l'utilisateur envers l'application métier souhaitée. Étant donné que notre objectif global est de minimiser les connaissances techniques requises pour le développement et déploiement d'applications métier sur le cloud, nous avons identifié dans cette optique deux objectifs majeurs.

Objectif 1 : Identifier les exigences minimales requises pouvant permettre la sélection et la composition de services métier satisfaisant la requête de l'utilisateur.

Objectif 2 : Ne pas ignorer les besoins non techniques qui peuvent s'avérer importants pour l'utilisateur, et qui peuvent nous permettre de sélectionner une composition plutôt qu'une autre.

4.4.1 Concepts du vocabulaire RIVAL

Nous définissons le vocabulaire RequIrement VocAbuLary (RIVAL) pour formaliser les besoins fonctionnels et non fonctionnels de l'utilisateur (exprimés via un formulaire Web) par l'utilisation de vocabulaires liés. La Figure 4.4 illustre les classes de RIVAL qui décrivent les concepts du vocabulaire, et les propriétés qui décrivent les relations entre les classes. Afin de réduire les connaissances techniques requises pour le provisionnement d'applications cloud, aucun besoin technique n'est décrit par l'utilisateur.

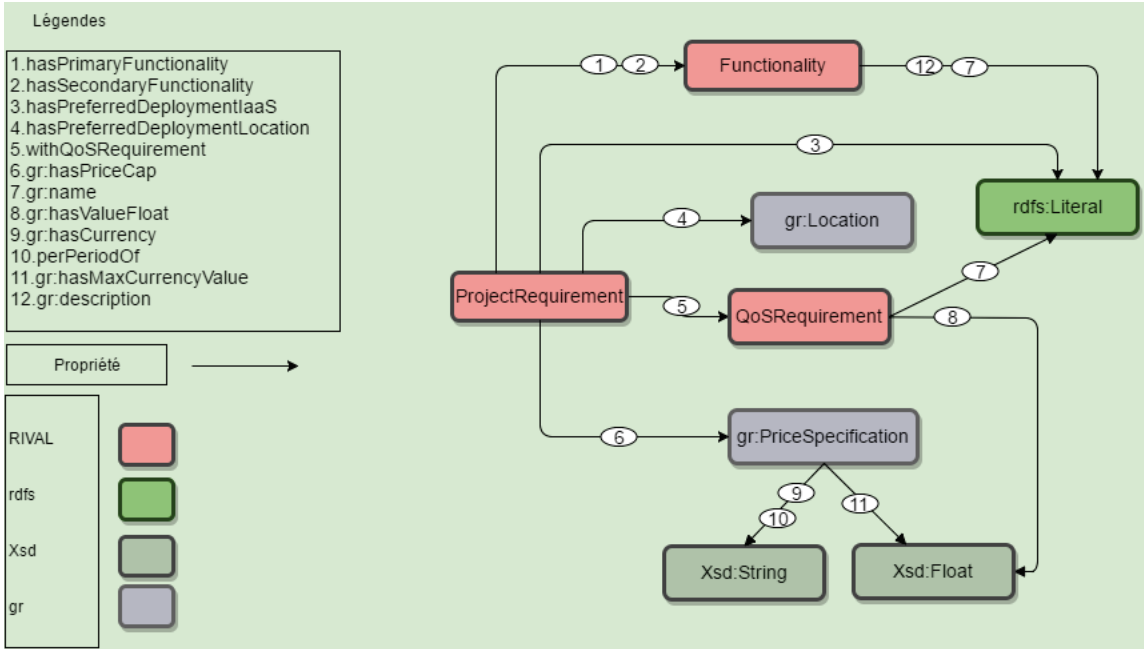


Figure 4.4: RIVAL - Vocabulaire de description des besoins minimaux, fonctionnels et non fonctionnels, de l'utilisateur

Les besoins fonctionnels répondent au premier objectif fixé. Ils représentent les fonctionnalités globales que l'application métier doit accomplir. Ces dernières sont décrites soit par un mot clé, soit par le nom des services métier les satisfaisant. La description des fonctionnalités fait abstraction des détails techniques telles que les contraintes de déploiement ou de composition. Nous introduisons une distinction entre fonctionnalités primaires et secondaires (Définitions 6 et 7). Ces dernières guident respectivement la sélection des services primaires et de leurs possibilités de composition (services secondaires).

Définition 6. Les *fonctionnalités primaires* (propriété 1 de la Figure 4.4) décrivent les fonctionnalités globales souhaitées pour l'application métier désirée par l'utilisateur. Elles sont indépendantes les unes des autres, et peuvent être liées à des fonctionnalités secondaires.

Définition 7. Les *fonctionnalités secondaires* (propriété 2 de la Figure 4.4) décrivent des fonctionnalités liées à une fonctionnalité primaire, et qui permettent d'enrichir le fonctionnement de cette dernière.

Une fonctionnalité est fournie par un service métier. La fonctionnalité de l'application désirée est considérée comme primaire, ex : gestion de projet dans notre scénario. Toute fonctionnalité supplémentaire à ce gestionnaire de projet est considérée comme secondaire, ex : nécessiter un gestionnaire de version avec le gestionnaire de projet pour stocker les différentes versions des étapes du développement d'un événement. Seule une fonctionnalité primaire est permise par projet. Plusieurs fonctionnalités secondaires peuvent lui être associées.

Les besoins non fonctionnels répondent au second objectif fixé, et couvrent les besoins en termes de qualité de service ainsi que les préférences de l'utilisateur vis à vis de l'application métier désirée. Il est difficile pour un utilisateur d'estimer les seuils de tolérance acceptables pour les paramètres de qualité de service telles que la disponibilité ou l'intégrité des données. En effet, les utilisateurs visent toujours une qualité maximale. Pour ces raisons, nous utilisons des poids (Définition 8) que l'utilisateur affecte aux paramètres de qualité de service. Les besoins en qualité de service pour l'application désirée sont donc décrits dans RIVAL par leurs noms (propriété 7 de la Figure 4.4) ainsi que leurs poids qui sont assignés par l'utilisateur (propriété 8 de la Figure 4.4).

Définition 8. Le poids affecté à chaque paramètre de qualité de service décrit la priorité que l'utilisateur attribue à ce dernier. Nous avons choisi de laisser la possibilité à l'utilisateur de distribuer 10 points entre les paramètres QoS considérés, de façon à ce que la somme de toutes les valeurs attribuées soit égale à 10 (les valeurs peuvent être des entiers ou des décimaux). Le choix de borner la somme des valeurs assignées s'explique par le fait que l'utilisateur voulant toujours le maximum de qualité, ira choisir un 10 pour chaque paramètre si nous devions borner le poids de chaque paramètre au lieu de borner la somme. De plus, le choix de la somme égale à 10 est dû à la simplicité, à notre sens, de partager 10 points plutôt qu'un pourcentage. Ces poids seront utilisés pour noter la qualité des services découverts (cf. Section 5.2.5).

La description de ce besoin de qualité de service est facultative. Dans le cas où l'utilisateur ne détermine pas ses priorités pour les paramètres de qualité de services, le même poids sera automatiquement affecté par défaut à chacun de ces paramètres (2.5 pour chaque paramètre étant donné que nous considérons quatre paramètres de qualité de service). Le Tableau 4.1 montre un exemple d'affectation des poids de qualité de services, où l'utilisateur accorde plus d'importance au respect de la confidentialité et à la préservation contre la perte de ses données, par les services utilisés, qu'à la disponibilité ou le temps de réponse de ces derniers.

Tableau 4.1: Poids assignés aux paramètres de qualité de service.

Paramètres de qualité de service	Confidentialité des données	Préservation de la perte des données	Disponibilité	Temps de réponse
Poids assignés	4	4	1	1

Les préférences de l'utilisateur sont liées aux détails de déploiement et de paiement, à savoir :

- La localisation de déploiement (propriété 4 de la Figure 4.4) : en fonction de la sensibilité de ses données, l'utilisateur peut choisir, s'il le souhaite, le continent où son application est déployée. Ce besoin est facultatif.

- Le nom de l'IaaS hébergeant l'application désirée (propriété 3 de la Figure 4.4). Cette préférence est basée sur une précédente expérience d'utilisation d'une IaaS pour l'hébergement de l'application. Ce besoin est facultatif.

- Les détails de paiement sont décrits par la classe "PriceSpecification" du vocabulaire "Good Relations" [goob], à laquelle est associée une monnaie (propriété 9 de la Figure 4.4), un coût maximal (propriété 11 de la Figure 4.4) et une période de facturation (propriété 10 de la Figure 4.4).

4.4.2 Processus de création des fichiers .rival

Les besoins décrits par l'utilisateur sont convertis suivant le vocabulaire RIVAL et stockés dans le fichier requirements.rival. Ce dernier représente des triplets RDF [rdf] (Sujet, Prédicat, Objet) décrivant les besoins de l'utilisateur. La traduction des besoins de l'utilisateur depuis un formulaire Web vers un fichier .rival se fait comme illustré sur la Figure 4.5. D'abord, nous extrayons les informations depuis le formulaire Web. Un fichier .rival vierge est créé et rempli par les préfixes des vocabulaires utilisés. Les préfixes permettent d'éviter d'utiliser l'URL du vocabulaire en instanciant ses classes et ses propriétés (ex : le préfixe "gr" remplace l'URL du vocabulaire GoodRelations "http://purl.org/goodrelations/v1#"). Puis la construction des triplets RDF a lieu. Elle se fait par l'ajout des propriétés décrivant le projet ainsi que leurs valeurs. Ces dernières peuvent être des données ou faire référence à d'autres ressources (sujets). Nous considérons le projet comme le premier sujet à décrire.

Le Listing 4.4 illustre un fichier .rival généré pour le scénario du gestionnaire de projet décrit précédemment. Les lignes 3-12 décrivent respectivement les fonctionnalités primaire et secondaires désirées. Les préférences de l'utilisateur sont décrites dans les lignes 14-21, incluant le fournisseur et la localisation de déploiement préférés (lignes 14-15), et les détails de paiement (lignes 17-21). Les lignes 23-38 décrivent les besoins en terme de qualité de service. Ce fichier .rival représente une entrée pour la phase de découverte de services métier et la phase de découverte et sélection d'IaaS dans MADONA.

Les besoins de l'utilisateur peuvent générer des conflits entre eux. Un conflit a lieu lorsque les besoins génèrent des incompatibilités entre des attributs logiciels communs [EG04], ou lorsque l'exécution d'une activité empêche l'exécution d'une autre [HHT02]. Cela peut être dû à une incohérence dans les spécifications dans le cas de multiples intervenants [Eas94]. Dans notre travail, si des conflits peuvent exister entre deux services métier, ils ne seront pas reflétés sur l'application générée. En effet, la génération de plans de composition consiste à composer les services métier qui **doivent** et ceux qui **peuvent** être composés (contraintes et possibilités de composition).

```
<http://mydomain.fr/rival/project_management_scenario> a rival:Project

        rival:hasPrimaryFunctionality
        [ a Functionality;
          gr:description "project management"];

        rival:hasSecondaryFunctionality
        [ a Functionality;
          gr:description "version control system"],

        [ a Functionality;
          gr:description "planning management"];

        rival:hasPreferredDeploymentIaaS "Amazon";
        rival:hasPreferredDeploymentLocation Europe;

        gr:hasPriceCap
        [ a gr:PriceSpecification;
          gr:hasCurrency "euro";
          gr:hasMaxCurrencyValue "30";
          gr:perPeriodOf "month"];

        rival:withQoSRequirement
        [ a rival:QoSRequirement;
          gr:name "Data_Privacy";
          gr:hasValueFloat  "4"],

        [ a rival:QoSRequirement;
          gr:name "Response_Time";
          gr:hasValueFloat  "1"],

        [ a rival:QoSRequirement;
          gr:name "Data_loss";
          gr:hasValueFloat  "4" ],

        [ a rival:QoSRequirement;
          gr:name "Availability";
          gr:hasValueFloat  "1"].
```

Listing 4.4: Fichier .rival généré à partir de la requête de l'utilisateur, décrite à travers un formulaire Web, pour le scénario utilisé

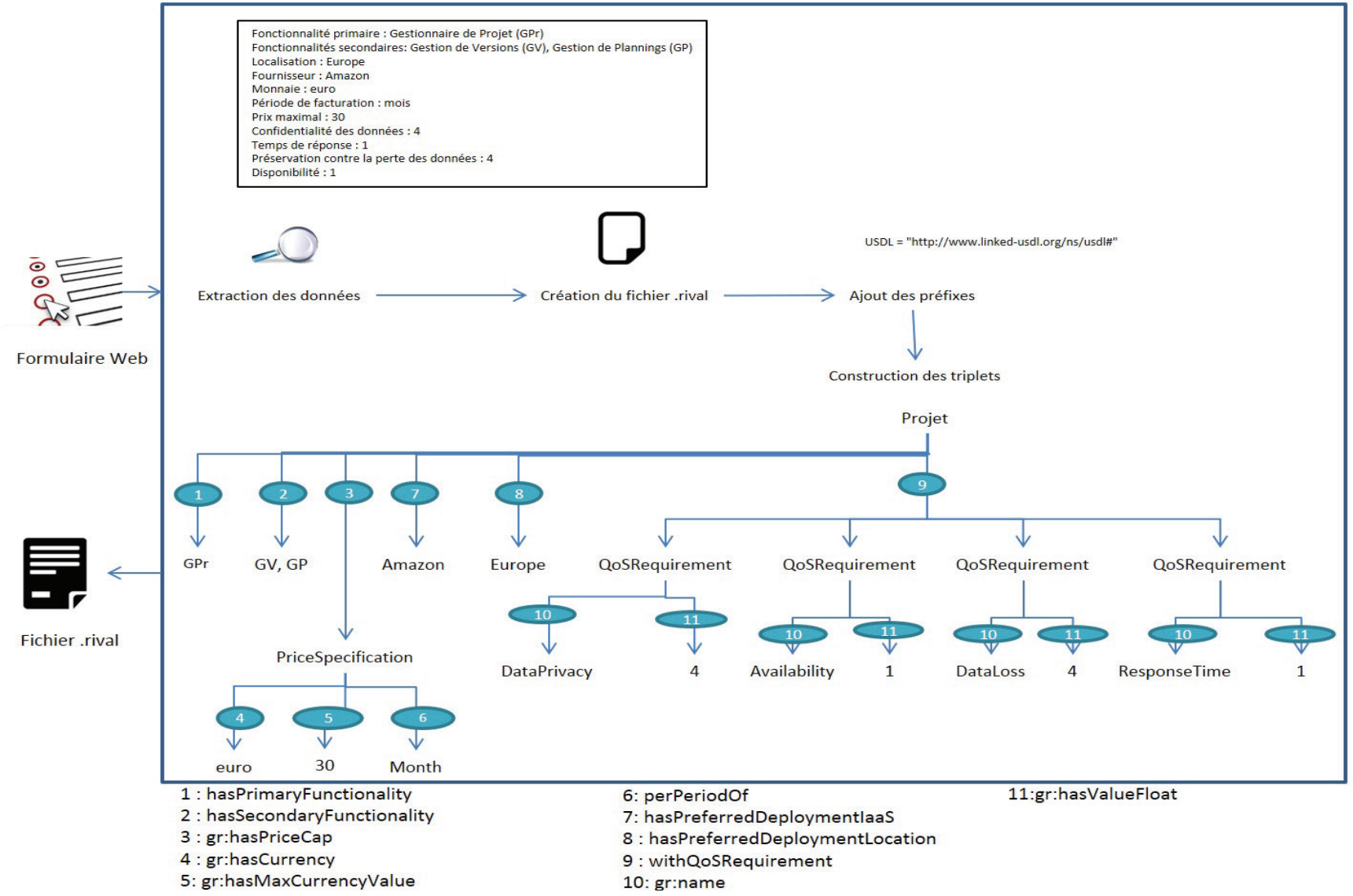


Figure 4.5: Processus de création des fichiers .rival

4.5 Conclusion

Dans la première partie de ce chapitre, nous avons défini le langage Linked USDL étendu pour la description des services de la marketplace (services métier et IaaS). Plusieurs concepts ont été repris du Linked USDL, principalement des modules usdl-core et usdl-price. De nouveaux concepts ont été ajoutés à ceux là pour décrire les relations qu'un service métier doit et peut avoir avec les autres afin de connaître entre autres la composabilité d'un service. Nous avons aussi décrit pour chaque service métier ses contraintes de déploiement et ses paramètres configurables afin d'automatiser respectivement le déploiement et la configuration souhaitée par l'utilisateur pour un service métier donné. Nous avons décrit les caractéristiques techniques d'un service IaaS afin de permettre la sélection de ressources répondant aux contraintes de déploiement de chaque service métier. Nous avons confié la description de la qualité des services de la marketplace à un service tiers afin d'avoir une représentation objective et représentative des états de la qualité du service.

Dans la deuxième partie de ce chapitre, nous avons défini le vocabulaire RIVAL pour formaliser la requête de l'utilisateur, qui est décrite à travers un formulaire Web. Dans ce vocabulaire, nous avons défini les besoins minimaux (fonctionnels et non fonctionnels) permettant une bonne sélection et composition de services métier, et avons introduit la notion de fonctionnalités primaire et secondaire. Les besoins non fonctionnels incluent les préférences de déploiement et de coût de l'utilisateur, et ses besoins de qualité de service. Les besoins pris en compte dans RIVAL n'incluent aucun besoin technique, et sont exprimés à un haut niveau d'abstraction. A titre d'exemple, les besoins en qualité de services sont exprimés en termes de poids symbolisant l'importance que l'utilisateur affecte à ce paramètre au lieu d'être exprimés en valeurs précises pour chaque paramètre de qualité de service.

Le chapitre suivant décrit MADONA pour le provisionnement (développement et déploiement) automatique d'applications sur le cloud. MADONA s'appuie sur le langage Linked USDL étendu pour la description des services de la marketplace et sur RIVAL pour la description des besoins de l'utilisateur.

Chapitre 5

MADONA - Méthodologie orientée service pour le provisionnement automatique d'applications cloud

Sommaire

5.1 Introduction

Dans ce chapitre, nous décrivons MADONA, notre méthodologie de provisionnement automatique (composition et déploiement de services métier) d'applications métier orientées service dans les environnements cloud. MADONA tire avantage du cloud computing et de l'AOS pour réduire les connaissances techniques et l'implication de l'utilisateur nécessaires au provisionnement de ces applications. L'intervention humaine ainsi que les connaissances techniques sont réduites puisque les phases de MADONA sont complètement automatisées. En effet, l'utilisateur intervient uniquement lors de l'élicitation de son besoin et l'introduction de ses données de configuration, et lorsque l'application est déployée et prête à être utilisée. MADONA permet la découverte et la composition automatiques de services métier. Elle prend en entrée les besoins fonctionnels et non fonctionnels exprimés par l'utilisateur via un formulaire Web suivant le vocabulaire RIVAL (cf. Chapitre 4). Elle compose ensuite les services métier de manière automatique constituant ainsi une application abstraite qui sera déployée sur une infrastructure IaaS. Lorsque le processus de découverte ne retourne pas de services répondant aux besoins de l'utilisateur, MADONA permet l'intégration de nouveaux services métier à la marketplace. MADONA utilise un orchestrateur de services pour la gestion du déploiement, de la composition, et de la configuration de services métier. L'orchestrateur que nous avons choisi pour notre implémentation qui sera décrite dans le Chapitre 6 est Juju. Par conséquent, tous les exemples de scripts de déploiement et de configuration illustrés dans ce chapitre sont pour l'orchestrateur Juju.

MADONA est composée des phases suivantes :

(1) Découverte de services métier.

(2) Intégration de nouveaux services.

(3) Génération des plans de composition de services.

(4) Découverte et sélection d'IaaS.

(5) Notation des services et sélection des plans de composition.

(6) Configuration et personnalisation de l'application métier.

(7) Déploiement automatique de l'application métier.

Les phases de découverte de services métier et d'IaaS reposent sur les descriptions des services de la marketplace avec le langage Linked USDL étendu (cf. Chapitre 4).

Le reste de ce chapitre est organisé comme suit. La Section 5.2 décrit et illustre les différentes phases de MADONA. La Section 5.3 conclut ce chapitre.

5.2 Les phases de MADONA

La Figure 5.1 illustre les différentes phases de MADONA et le flux de données entre ces phases.

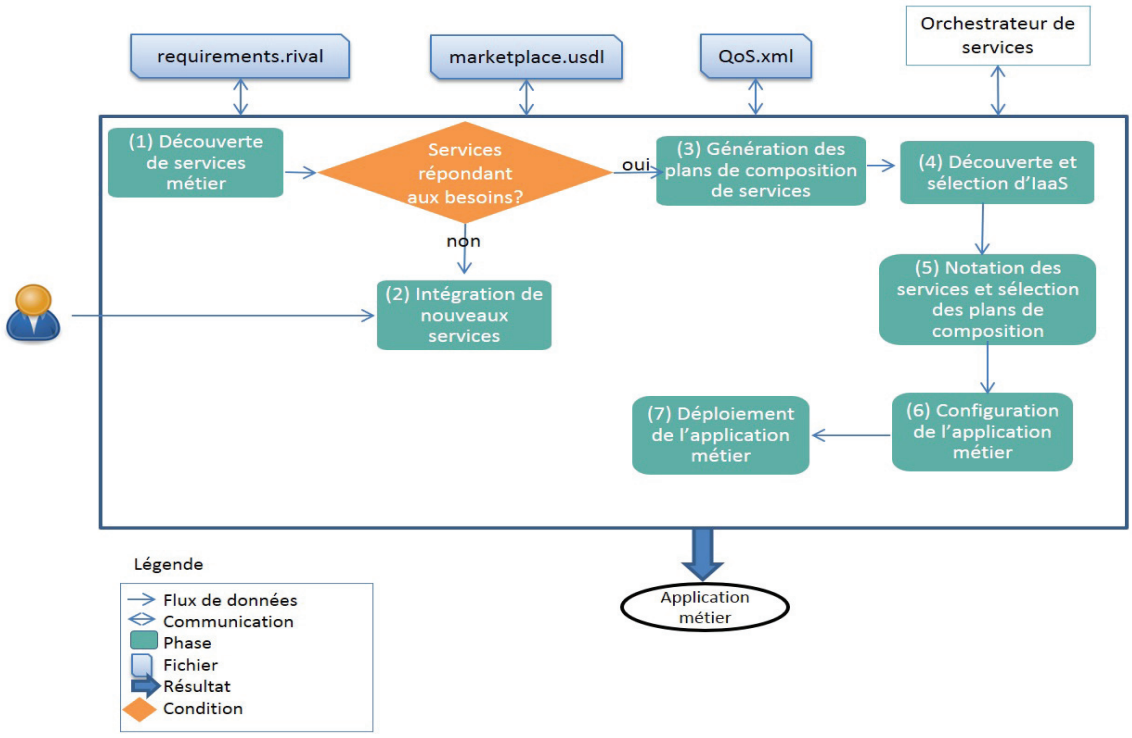


Figure 5.1: MADONA - Méthodologie de provisionnement automatique d'applications métier orientées service sur le cloud

MADONA communique avec :

- Le fichier requirements.rival lors des phases 1, 4, et 5 de la Figure 5.1 pour récupérer respectivement les besoins de l'utilisateur en termes de fonctionnalités désirées, les préférences de déploiement et de coût, et les contraintes de qualité de service représentées par des poids affectés par l'utilisateur à chaque paramètre considéré.

- Le fichier marketplace.usdl lors des phases 1, 2, 4, 5, et 7 de la Figure 5.1 pour (i) découvrir des services métier répondant aux fonctionnalités désirées de l'utilisateur, (ii) stocker la description des services intégrés, (iii) découvrir et sélectionner une IaaS de déploiement, (iv) noter les services et les plans de composition, et (v) récupérer les contraintes de déploiement d'un service métier.

- Les fichiers QoS XML décrivant les paramètres de qualité de service des services de la marketplace. La communication entre MADONA et les fichiers QoS XML

se fait lors de la notation des services et des plans de composition (phase 5 de la Figure 5.1).

- Un orchestrateur de service lors de la configuration (phase 6 de la Figure 5.1) et du déploiement (phase 7) des services de l'application générée.

Les différentes phases de MADONA et le flux de données entre ces dernières sont illustrés dans les sous-sections suivantes.

5.2.1 Découverte de services métier

Cette phase consiste à rechercher des services métier répondant aux fonctionnalités désirées par l'utilisateur en tenant compte des possibilités de composition des services découverts. Dans ce qui suit, nous appellerons services primaire et secondaire, les services métier répondant respectivement aux fonctionnalités primaire et secondaire souhaitées.

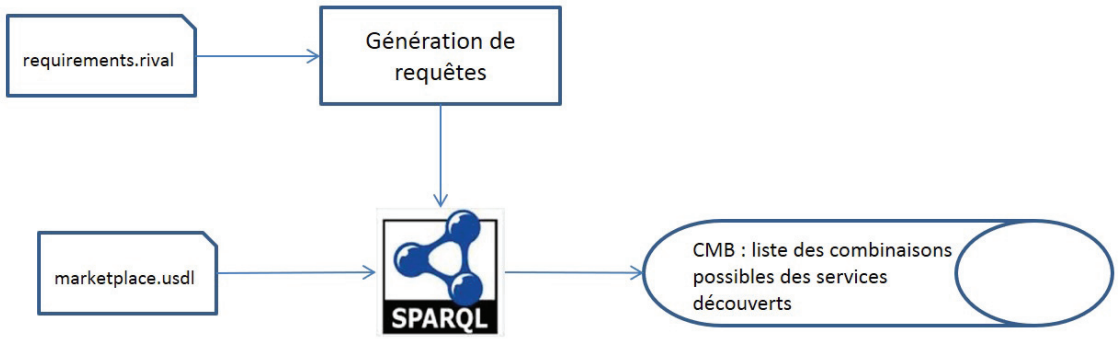


Figure 5.2: Processus de découverte de services métier

Le processus de découverte de services métier est illustré par la Figure 5.2. Il consiste à faire correspondre les besoins fonctionnels de l'utilisateur (décrits dans le fichier requirements.rival) avec les descriptions des services de la marketplace. Pour cela, nous construisons une requête SPARQL Protocol and RDF Query Language (SPARQL) à partir des informations contenues dans le fichier requirements.rival généré lors de l'expression des besoins. Cette requête est lancée sur le fichier de description des services de la marketplace. Elle récupère les services métier répondant aux fonctionnalités primaire et secondaires en respectant les préférences de l'utilisateur en termes de services métier. En effet, l'utilisateur a la possibilité d'introduire un mot clé pour rechercher le service souhaité ou le nom du service en question.

Le Listing 5.1 illustre l'algorithme de construction de la requête SPARQL qui sert à sélectionner les services qui répondent aux fonctionnalités primaire et secondaires désirées par l'utilisateur. Les variables keeUp et keepUp2 de l'algorithme permettent de générer les variables de la requête SPARQL (ex : x, y) en fonction des besoins de

l'utilisateur. Cette requête est construite par l'ajout des préfixes des vocabulaires utilisés dans la requête (lignes 5-6). Ensuite les caractéristiques sur le nom ou la description de la fonctionnalité primaire (lignes 12 à 17) en fonction des préférences de l'utilisateur sont ajoutées à la requête (recherche par nom ou par mot clé). Pour chaque fonctionnalité secondaire désirée, les lignes de la requête permettant de récupérer les services métier qui répondent à sa description, et qui font partie des possibilités de composition du service primaire sont générées (lignes 20-46).

```
1  In: String primary_functionality_description, primary_functionality_name;
2      ArrayList<String> secondary_functionalities;
3  Out: String queryString;
4
5  queryString = "PREFIX usdl: <http://www.linked-usdl.org/ns/usdl#>"+
6                "PREFIX gr:<http://www.heppnetz.de/ontologies/goodrelations/v1#>" +
7                "SELECT ?b ?e ?i WHERE {";
8
9  char keepUp='a';
10 char keepUp2;
11
12 queryString = queryString + "?x gr:description " +" ?"+keepUp+
13   "FILTER regex(?"+keepUp+","+primary_functionality_description+",i).";
14 keepUp=keepUp+1;
15
16 queryString = queryString + "?x gr:name " +"?"+keepUp+
17     "FILTER regex(?"+keepUp+","+primary_functionality_name+",i).";
18
19
20 If secondary_functionalities.length>=1
21 {
22     For (int i=0; i<secondary_functionalities.length; i++)
23     {
24        keepUp=keepUp+1;
25
26        keepUp2=keeUp+1;
27
28        queryString=queryString+"?"+keepUp+" gr:description ?"+keepUp2
29        +"FILTER regex(?"+keepUp2+","+secondary_functionalities.get(i)
30        +",i).";
31
32        keepUp2=keepUp2+1;
33
34        queryString=queryString+"?"+keepUp+" gr:name ?"+keepUp2+".";
35
36        keepUp=keepUp2;
37
38        keepUp2=keepUp2+1;
39
40        queryString=queryString+"?x usdl:hasPossibleComposition ?"+keepUp2+".";
41
42        queryString=queryString+"?"+keepUp2+"gr:name ?"+keeUp+".";
43
44        keepUp=keepUp2;
45     }
46 }
47 queryString=queryString+"}";
48 return queryString;
```

Listing 5.1: Algorithme de génération de la requête SPARQL

Le Listing 5.2 illustre la requête SPARQL générée pour le scénario du gestionnaire de projet précédemment décrit (cf. Section 4.2). Dans cette requête nous souhaitons récupérer les services métier ayant pour fonctionnalité un gestionnaire de projet (ligne 5), et qui ont parmi leurs possibilités de composition un service métier assurant la gestion de versions (lignes 8-11) répondant à la première fonctionnalité secondaire de l'utilisateur, et un autre service métier assurant la gestion de

plannings (lignes 13-16) répondant ainsi à la deuxième fonctionnalité secondaire de l'utilisateur. Nous supposons que les services métier apparaissant dans la description d'un service en tant que contraintes ou possibilités de composition existent dans la marketplace.

Le résultat de la requête représente les différentes combinaisons possibles des variables "b", "e", et "i" qui représentent les services primaires et secondaires. Nous stockons ces différentes combinaisons dans la liste *CMB*.

```
1  PREFIX usdl: <http://www.linked-usdl.org/ns/usdl#>
2  PREFIX gr:<http://www.heppnetz.de/ontologies/goodrelations/v1#>
3
4  SELECT ?b ?e ?i WHERE {
5  ?x gr:description  ?a FILTER regex(?a,"project management","i" ).
6  ?x gr:name  ?b FILTER regex(?b,"","i" ).
7
8  ?c gr:description ?d FILTER regex(?d,"version control system","i").
9  ?c gr:name ?e.
10 ?x usdl:hasPossibleComposition ?f.
11 ?f gr:name ?e.
12
13 ?g gr:description ?h FILTER regex(?h,"Planning management system","i").
14 ?g gr:name ?i.
15 ?x usdl:hasPossibleComposition ?j.
16 ?j gr:name ?i.}
```

Listing 5.2: Requête SPARQL générée pour le scénario de gestion de projet

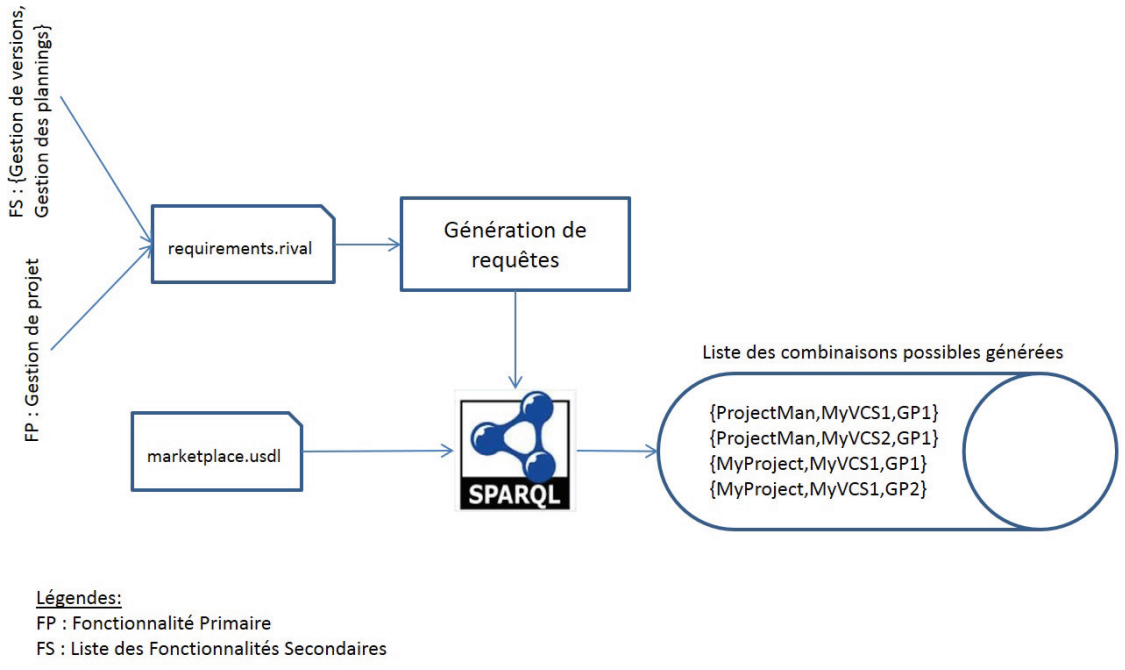


Figure 5.3: Services découverts pour le scénario utilisé

La Figure 5.3 illustre les services découverts pour le scénario que nous utilisons. Deux services répondant à la fonctionnalité de gestion de projet et qui peuvent être composés avec un service de gestion de versions et un service de gestion de plannings

sont découverts. ProjectMan a comme possibilité de composition les services MyVCS1 et MyVCS2 pour la gestion de versions et GP1 pour la gestion de plannings. MyProject peut être composé avec les services MyVCS1 pour la gestion de versions, et GP1 et GP2 pour la gestion de plannings. Quatre combinaisons de services sont générées.

La requête SPARQL pour la découverte de services métier peut toutefois ne pas être concluante dans deux cas de figure :

- Aucun service de la marketplace ne correspond à au moins une fonctionnalité désirée.

- Toutes les fonctionnalités désirées sont couvertes par au moins un service de la marketplace mais aucune composition de ces services n'est possible.

Dans ce cas, l'utilisateur est notifié par le résultat non concluant de la requête, et ce dernier est guidé pour intégrer, s'il le souhaite, à la marketplace de nouveaux services métier développés par des tiers.

5.2.2 Intégration de nouveaux services métier à la marketplace

L'intégration des nouveaux services métier se produit lorsque le processus de découverte ne satisfait pas entièrement la requête de l'utilisateur ou lorsqu'un utilisateur ou un fournisseur de services souhaite enrichir la marketplace. Nous appellerons dans cette section utilisateur, toute personne souhaitant intégrer de nouveaux services à la marketplace.

Cette phase est optionnelle et consiste à guider l'utilisateur dans la démarche d'intégration. Les nouveaux services sont intégrés par leurs descriptions et scripts de déploiement, de composition, et de configuration. Ils sont automatiquement pris en compte pour les compositions et déploiements futurs. Les services intégrés sont développés par des tiers. Ce sont des packages prêts à être déployés sur différentes infrastructures. Ils ne sont pas spécialement développés pour l'utilisateur qui souhaite les intégrer à la marketplace. Il s'agit d'une importation d'un service métier depuis une autre marketplace.

L'intégration des nouveaux services métier nécessite de l'utilisateur d'effectuer deux actions comme illustré sur la Figure 5.4 :

1. Téléchargement des scripts de déploiement, de composition et de configuration : ces scripts sont écrits suivant l'outil d'orchestration ciblé. Ils permettent respectivement (i) d'installer l'environnement nécessaire au déploiement d'un service métier et de déployer ce dernier, (ii) de composer le service nouvellement intégré à d'autres services qui peuvent représenter ses contraintes ou possibilités de composition, et (iii) de personnaliser le service avec des informations de l'utilisateur tel que le logo de son entreprise et le nom de l'application. Ces scripts sont écrits par le fournisseur de services. L'action de télécharger des scripts de déploiement,

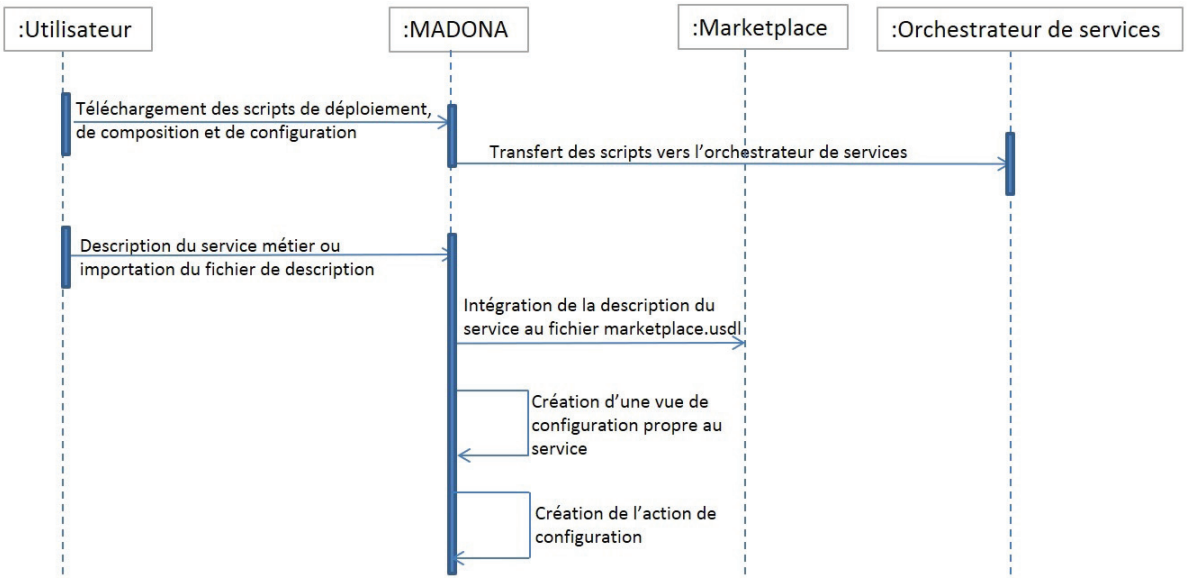


Figure 5.4: Intégration de nouveaux services métier à la marketplace

de composition et de configuration est effectuée par l'utilisateur ou le fournisseur de services. Suite à cette action, MADONA transfère l'archive des scripts vers le serveur de l'outil d'orchestration utilisé.

2. Description des services métier : l'utilisateur a la possibilité d'importer le fichier de description du service, ou introduire sa description via un formulaire Web. La description des services métier génère deux actions effectuées par MADONA comme suit :

   - Intégration de la description du service au fichier marketplace.usdl.
   - Rendre le service métier configurable. Cela passe tout d'abord par la création d'une vue de configuration pour le service nouvellement intégré. Une vue représente un fichier HTML décrivant le contenu d'une page Web (un formulaire Web de configuration dans ce cas). Cette vue est créée automatiquement à partir des paramètres configurables du service et sera utilisée lorsque le service sera sélectionné dans les provisionnements futurs. Elle permettra à l'utilisateur d'introduire ses données de configuration. Une donnée de configuration est la valeur que l'utilisateur associe à un paramètre configurable. Tel qu'illustré dans la Figure 5.5, la vue est créée comme suit : d'abord un fichier portant le nom du service suivi de la chaine de caractère "Conf" est créé dans le répertoire associé aux vues du système implémentant MADONA. Ce fichier est enrichi d'un message qui sera affiché à l'utilisateur lors de la configuration du service lui permettant ainsi de savoir quel est le service qu'il va personnaliser. Pour chaque paramètre configurable du service, nous définissons le code HTML associé permettant d'afficher sur le for-

mulaire de configuration le nom du paramètre configurable et de lui associer le composant Web adéquat (ex : bouton parcourir). Nous ajoutons à la fin du formulaire, le bouton de soumission qui lancera l'action de configuration. Cette dernière est implémentée par une méthode générée automatiquement et intégrée au code source de l'implémentation du système de MADONA. Elle permet de sauvegarder les données de configuration, et de générer et d'exécuter le script de configuration du service (cf. Section 6.4).

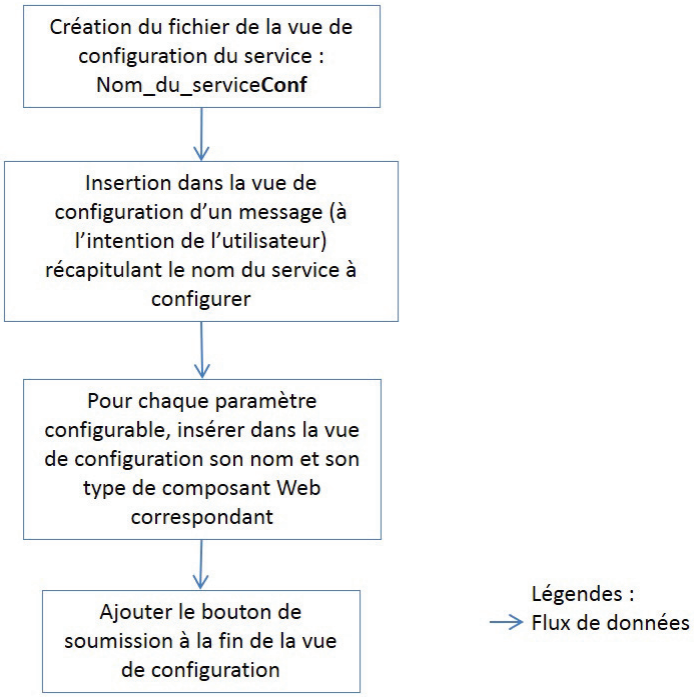


Figure 5.5: Processus de création de la vue de configuration

5.2.3 Génération des plans de composition de services

A partir des résultats de la découverte de services métier, plusieurs plans de composition de services (Définition 9) sont générés. Chaque plan décrit un moyen de répondre à la requête de l'utilisateur.

Définition 9. Un plan de composition de services (P_j) (Figure 5.6) est une application abstraite qui compose des services métier et leur assigne une IaaS pour leur déploiement. Il est composé d'une partie fonctionnelle et d'une partie de déploiement. La partie fonctionnelle consiste en une liste de relations de composition. Une relation de composition lie un service à ses contraintes et/ou ses possibilités de composition. Dans ce travail, nous ne considérons que les contraintes de composition fortes. La partie déploiement d'un plan de composition de services représente l'IaaS sur laquelle il peut être déployé.

Pour simplifier, nous utiliserons dans ce qui suit relation et plan de composition, pour désigner respectivement une relation de composition et un plan de composition de services.

Trois métadonnées s'ajoutent à la description d'un plan de composition :

- ***NoteQoS(P_j)*** représente la note du plan de composition par rapport aux paramètres de qualité de service. Nous avons choisi de noter les plans de composition générés afin d'en sélectionner le meilleur. Nous ferons référence à cette variable dans le texte en tant que "note QoS".

- ***Coût(P_j)*** représente l'estimation du coût d'utilisation de l'application correspondant au plan de composition sur une période donnée. Le coût d'un plan de composition est une estimation du coût des ressources infrastructures (ex : machines virtuelles) hébergeant ce dernier. Chaque service métier est déployé sur une machine virtuelle répondant à ses contraintes de déploiement (CPU, RAM, disque) indiquées dans sa description .usdl. Chaque type de machine virtuelle a un coût qui dépend du fournisseur IaaS qui le déploie. Le coût d'un plan de composition (Équation 5.1) est alors calculé comme la somme des coûts de déploiement de chaque service métier (S_i) d'un plan de composition (P_j) en fonction du fournisseur IaaS automatiquement sélectionné (cf . Section 5.2.4).

$$Cout(P_j) = \sum_{i=0}^{k-1} cout(S_i, Infra) \tag{5.1}$$

  Où k est le nombre de services métier impliqués dans le plan de composition, et *Infra* représente l'IaaS sélectionnée.

- ***Id*** représente l'indice associé à un plan de composition dans la liste des plans générés.

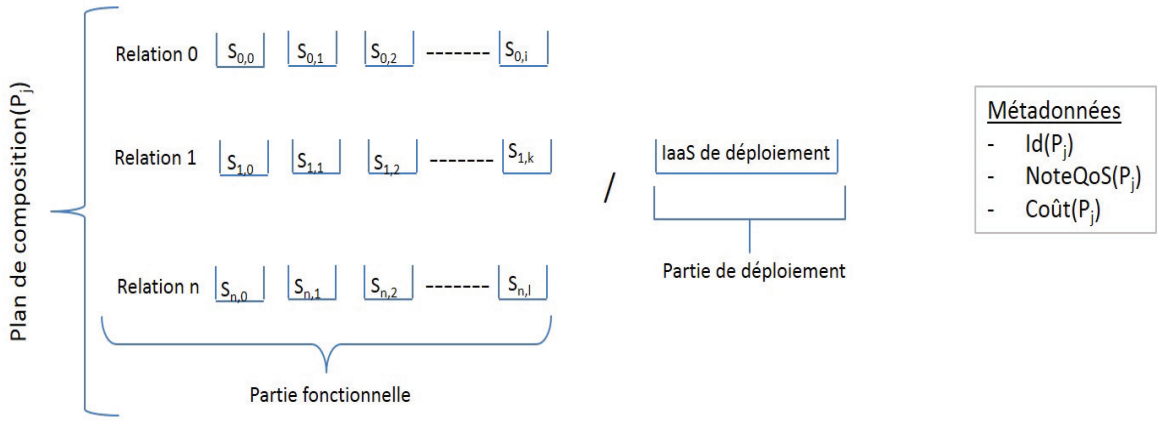


Figure 5.6: Plan de composition

La partie fonctionnelle des plans de composition est générée comme suit (Listing 5.3) : pour chaque combinaison possible de services primaires et secondaires *cmb* issue des résultats de la découverte *CMB*, une relation est initialisée par le premier élément de *cmb* (représentant un service primaire) (ligne 13), suivi de ses contraintes de composition (ligne 14), puis des autres éléments de *cmb* (lignes 16-19) représentant les possibilités de composition du service primaire qui répondent aux n fonctionnalités secondaires désirées. Un plan de composition est alors initialisé avec cette relation. Pour chaque service (à partir du deuxième service) de chaque relation du plan de composition, nous vérifions s'il a des contraintes de composition au travers d'une nouvelle requête SPARQL. Si le service a des contraintes de composition, une nouvelle relation liant le service et ses contraintes de composition est créée et ajoutée au plan de composition. Cette vérification est un processus itératif qui vérifie les contraintes de composition des relations nouvellement ajoutées, au fur et à mesure de la complétion du plan de composition (lignes 27-51).

```
Input : CMB (Discovery Results);
CP: List of composition plans;
cp: A composition plan (List of relations);
Relation: List of services involved in a relation;
cmb: List of services;
for (int i=0; i<CMB.size; i++)
{
    cmb=CMB.get(i);
    for (int j=0; j<cmb.size; j++)
    {
        if (j==0)
        {
            Relation.add(cmb.get(j))
            Relation.add(getConstraint(cmb.get(j))
        }
        else
        {
            Relation.add(cmb.get(j)
        }
    }
    cp.add(Relation);
    cp=verifConstraints(cp);
    CP.add(cp);
    Relation=new(list);
}

composition_plan verifConstraints(composition_plan cp)
{
 int l=0;
 nb_relation=1;
 while(l< nb_relation)
 {
    for(int k=1; k<cp.get(l).size; k++)
        {
            if(getConstraints(cp.get(l).get(k)) not null)
            {
                Relation=new(list);
                Relation.add(cp.get(l).get(k));
                for(int m=0; m<getConstraints(cp.get(l).get(k)).size; m++)
                {
                    Relation.add(getConstraints(cp.get(l).get(k)).get(m));
                }
                cp.add(Relation);
                nb_relation=nb_relation+1;
            }
        }
        l=l+1;

 }
 return(cp);
}
```

Listing 5.3: Algorithme de génération des plans de composition

La Figure 5.7 illustre le processus de génération des plans de composition pour le scénario du gestionnaire de projet. A partir de la liste *CMB* des combinaisons possibles générée lors de la phase de découverte de services métier, les premières relations des plans de composition sont générées. La première relation de chaque plan de composition inclut les contraintes de composition du premier élément de chaque *cmb* de *CMB*. De nouvelles relations sont ajoutées au fur et à mesure pour composer les services et leurs contraintes de composition. La partie fonctionnelle du premier plan de composition comporte quatre relations. La première est composée de ProjectMan, un service primaire; MySQL et MyCRM, ses contraintes de composition; et MyVCS1 et GP1 des services fournissant respectivement la gestion de versions et de plannings et qui représentent des possibilités de composition de ProjectMan. L'association de MyVCS1 et GP1 représente une combinaison possible des services secondaires associés à Project-

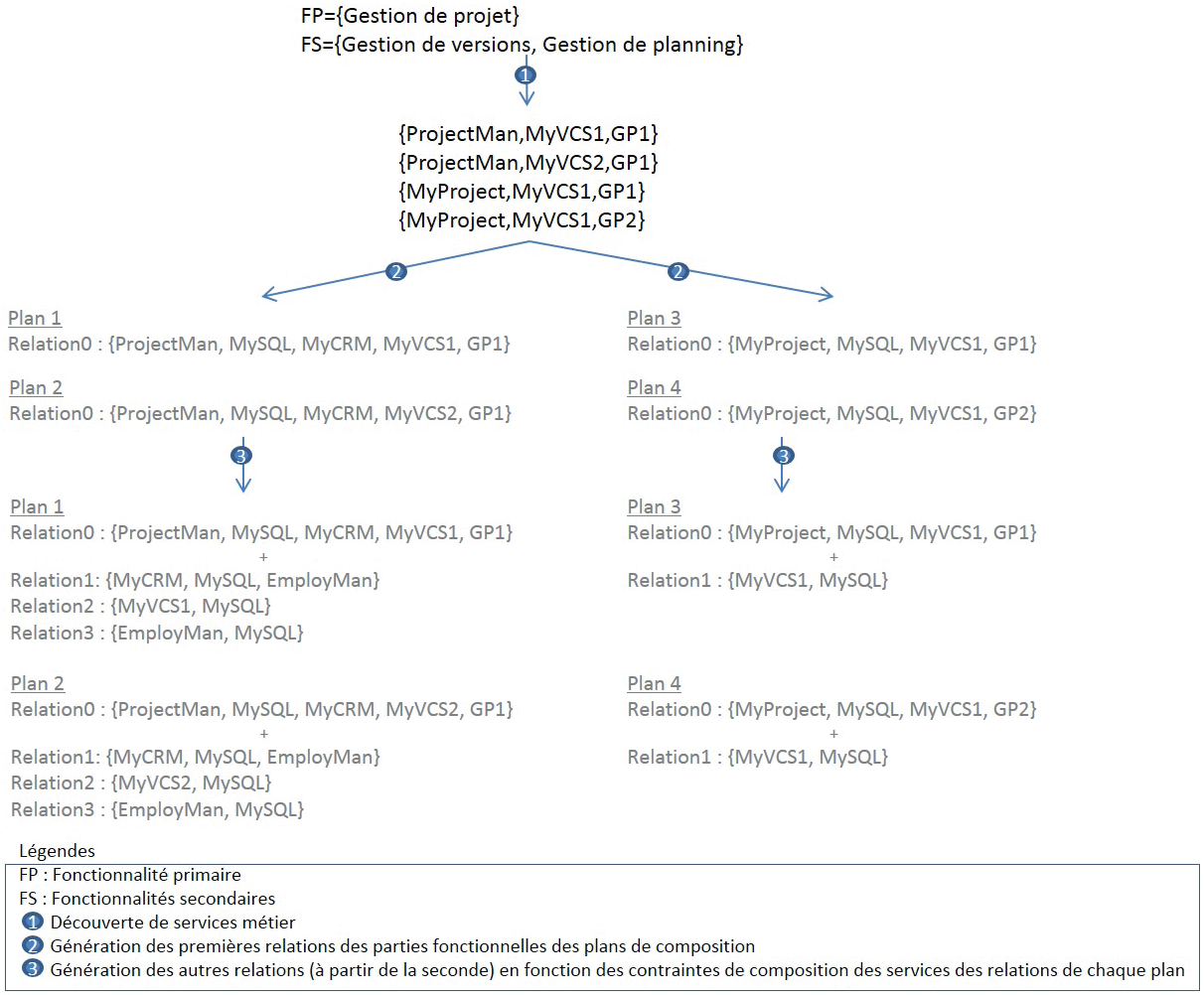


Figure 5.7: Processus de génération des plans de composition pour le scénario du gestionnaire de projet

Man. Les trois autres relations lient respectivement MyCRM, MyVCS1, et EmployMan avec leurs contraintes de composition.

La Figure 5.8 illustre les plans de composition générés pour le scénario du gestionnaire de projet. Rappelons qu'un plan de composition contient une partie fonctionnelle et une partie déploiement. Les plans générés contiennent la partie fonctionnelle composant les services découverts répondant aux besoins fonctionnels de l'utilisateur, à laquelle sera ajoutée la partie déploiement comportant l'infrastructure de déploiement sélectionnée en phase de découverte d'IaaS. Sur cette figure, les valeurs des notes QoS et des coûts des plans de composition ne sont pas encore introduites (NoteQoS=? et Coût=?). En effet, les notes QoS ainsi que les coûts des plans de composition générés sont calculés respectivement pendant les phases de notation des services et des plans de composition (cf. Section 5.2.5), et de découverte et sélection d'IaaS (cf. Section 5.2.4).

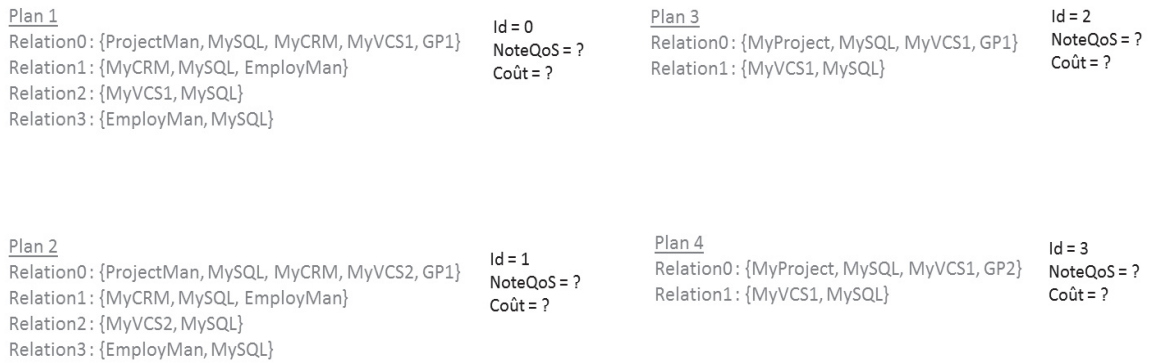


Figure 5.8: Plans de composition générés pour le scénario du gestionnaire de projet

5.2.4 Découverte et sélection d'IaaS

Le processus de découverte d'IaaS sélectionne une IaaS pour le déploiement des plans de composition générés. Cette sélection est obtenue en faisant correspondre les préférences de l'utilisateur (la localisation préférée, le fournisseur de déploiement préféré, et les détails de facturation) avec les descriptions .usdl des services IaaS de la marketplace.

La découverte et sélection d'IaaS suivent le processus suivant (illustré par la Figure 5.9) : l'étape 1 consiste à lancer une requête SPARQL (Listing 5.4) sur la marketplace à la recherche d'IaaS (ligne 2 du Listing 5.4) répondant à la localisation (ligne 3) et au fournisseur (lignes 4 à 7) désirés par l'utilisateur. Dans l'étape 2, les IaaS découvertes sont notées suivant leur qualité de service et les contraintes de QoS de l'utilisateur (cf. Section 5.2.5). L'IaaS ayant la plus haute note QoS est sélectionnée (étape 3). Le prix de chaque plan de composition précédemment généré est estimé suivant l'IaaS sélectionnée (étape 4). Cette estimation se fait sur la base du coût des types d'instance requis pour chaque service métier impliqué dans le plan de composition. Si le coût d'un plan de composition dépasse le coût maximum introduit par l'utilisateur, le plan de composition est exclu. Si aucun plan de composition ne reste, l'IaaS ayant la prochaine meilleure note QoS est sélectionnée (étape 3). Sinon, le fichier de configuration de l'outil d'orchestration est automatiquement mis à jour pour indiquer que l'environnement de déploiement sera l'IaaS sélectionnée (étape 5). Les notes QoS des parties fonctionnelles des plans de composition restants (non exclus) sont calculées (étape 6). L'IaaS sélectionnée constitue la partie déploiement des plans de composition.

```
1 SELECT  ?x ?c WHERE {
2 ?x usdl:hasClassification IaaS.
3 ?x gr:availableAtOrFrom  Europe.
4 ?x usdl:hasEntityInvolvement ?a.
5 ?a usdl:ofBusinessEntity ?b.
6 ?b gr:name ?c FILTER regex (?c, "Amazon","i").
7 ?a usdl:withBusinessRole Provider. }
```

Listing 5.4: Requête SPARQL générée pour la découverte d'IaaS

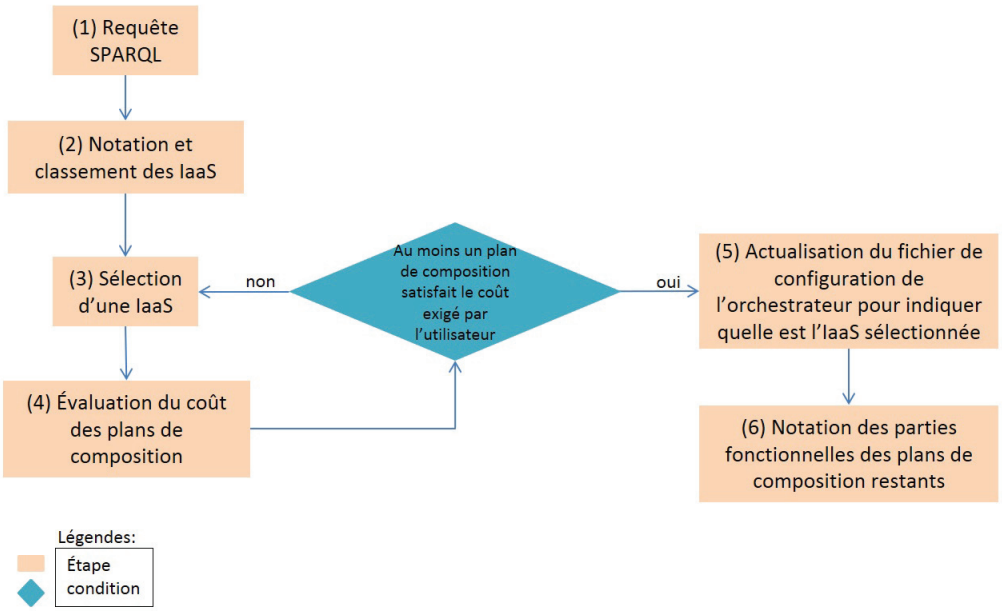


Figure 5.9: Processus de découverte et sélection d'une IaaS

Pour illustrer le processus de sélection d'IaaS, nous supposerons que dans notre scénario l'utilisateur n'ait pas choisi une IaaS préférée pour le déploiement de l'application générée, et que la découverte d'IaaS a retourné trois IaaS répondant aux préférences de localisation de l'utilisateur. La Figure 5.10 illustre le processus de sélection d'IaaS pour ce nouveau scénario. Les coûts des plans de composition sont évalués pour l'IaaS1 (l'IaaS ayant la plus grande note QoS). Rappelons que le coût d'un plan de composition est le coût des ressources (machines virtuelles) hébergeant les services de ce plan. Tous les plans de composition sont exclus puisque leur coût dépasse celui indiqué par l'utilisateur. Dans ce cas, les coûts des plans de composition sont évalués pour l'IaaS2 (l'IaaS ayant la seconde meilleure note QoS). Un seul plan de composition est exclu, nous calculons pour les autres plans leurs notes QoS. Les coûts des plans de composition ne vont pas être évalués pour l'IaaS3, étant donné que la sélection de l'IaaS2 remplit les conditions de l'utilisateur (coût et localisation). La Figure 5.11 illustre les plans de composition complétés par leur partie déploiement et qui vont être évalués par rapport à leur qualité de service.

5.2.5 Notation des services et sélection des plans de composition

Les services IaaS découverts à l'étape précédente (Section 5.2.4) ainsi que les services métier impliqués dans les parties fonctionnelles d'un plan de composition sont notés. Pour rappel, cette notation se fait selon les exigences de qualité de service de l'utilisateur et grâce à l'historique d'invocation des services de la marketplace. Ce dernier est fourni par un tiers impartial (ex : service Web) et donne les paramètres de qualité de service décrivant un service donné de la marketplace en fonction de ses invocations précédentes.

Figure 5.10: Illustration de la sélection d'IaaS

Soit S_i un service et Q_j un paramètre de qualité de service. Le calcul de la note QoS d'un service (service métier et IaaS) pour un paramètre QoS donné se fait en fonction du type de ce dernier (Équation 5.2). Deux scénarii sont possibles :

$$NoteQoS(S_i, Q_j) = \begin{cases} NoteQoS_{upper} \\ NoteQoS_{lower} \end{cases} \tag{5.2}$$

Cas 1 : plus grande est la valeur du paramètre de qualité de service, meilleur est le service, par exemple, la disponibilité du service. Dans ce cas, la note QoS associée à ce paramètre pour un service donné est calculée comme suit (Équation 5.3) :

$$NoteQoS_{upper} = \frac{Val(S_i, Q_j)}{Max(Q_j)} * Coefficient \tag{5.3}$$

Où :

- *Val* est la valeur du paramètre de qualité de service pour un service (service métier ou IaaS) donné.
- *Max* est la valeur maximale du paramètre de qualité de service parmi tous les services fournissant la même fonctionnalité (i.e., les services appartenant à la même

Figure 5.11: Plans de composition complétés par leur partie déploiement

catégorie décrite par la propriété gr:category).

- *Coefficient* est le poids précédemment assigné au paramètre de qualité de service par l'utilisateur.

Cas 2 : plus petite est la valeur du paramètre de qualité de service, meilleur est le service, par exemple, le temps de réponse. Dans ce cas, la note QoS associée au paramètre de qualité de service pour un service donné est calculée comme suit (Équation 5.4) :

$$NoteQoS_{lower} = (1 - \frac{Val(S_i, Q_j)}{Max(Q_j)}) * Coefficient \tag{5.4}$$

Soit n le nombre de paramètres QoS considérés. La note QoS globale d'un service S_i est calculée comme la somme des notes QoS de ce service pour chacun des paramètres de qualité de service considéré (Équation 5.5).

$$NoteQoS(S_i) = \sum_{j=0}^{n-1} NoteQoS(S_i, Q_j) \tag{5.5}$$

Le Tableau 5.1 décrit les paramètres de qualité de service liés à deux services métier répondant à la fonctionnalité de gestion de projet, en fonction de leurs précédentes invocations.

Tableau 5.1: Paramètres de qualité de service des services métier répondant à la fonctionnalité de gestion de projet en fonction de leurs invocations précédentes

Paramètres de qualité de service	ProjectMan	MyProject
Confidentialité des données (%)	80	50
Préservation de la perte des données(%)	89.7	3.9
Disponibilité (%)	90	90
Temps de réponse (ms)	224	495

Suivant la méthode de calcul de la note QoS décrite ci-dessus, nous calculons la note associée aux services de gestion de projet. $NoteQoS_{PM}$ (Équation 5.6) et

NoteQoS$_{MP}$ (Équation 5.7) représentent respectivement, la note QoS des services ProjectMan et MyProject.

$$NoteQoS_{PM} = (\frac{80}{80} * 4) + (\frac{89.7}{89.7} * 4) + (\frac{90}{90} * 1) + ((1 - \frac{224}{495}) * 1) \tag{5.6}$$
$$NoteQoS_{PM} = 9.54$$

$$NoteQoS_{MP} = (\frac{50}{80} * 4) + (\frac{3.9}{89.7} * 4) + (\frac{90}{90} * 1) + ((1 - \frac{495}{495}) * 1) \tag{5.7}$$
$$NoteQoS_{MP} = 3.67$$

La note QoS associée à chaque plan de composition est calculée comme la moyenne des notes QoS des m services impliqués dans le plan de composition (équation 5.8).

$$NoteQoS(P_j) = \frac{\sum_{i=0}^{m-1} NoteQoS(S_i)}{m} \tag{5.8}$$

La sélection du plan de composition et de l'IaaS ayant la plus haute note QoS se fait suivant l'algorithme illustré dans le Listing 5.5. Les plans de composition générés et les IaaS découvertes sont stockés chacun dans une liste et leurs notes QoS associées dans une seconde liste. L'algorithme de sélection prend en entrée la liste des notes QoS des plans de composition ou des IaaS et retourne l'indice de la note QoS la plus haute. Cet indice correspond aussi à l'indice du plan de composition ou de l'IaaS associés dans respectivement la liste des plans de compositions générés et des IaaS découvertes.

La Figure 5.12 illustre les plans de composition gardés après la sélection de l'IaaS de déploiement, avec leurs notes QoS. La valeur sur chaque service du premier plan de composition représente la note QoS du service. La note QoS du plan de composition est la moyenne des notes QoS des services impliqués dans ce plan. Le plan ayant la plus haute note QoS (celui de ProjectMan dans notre scénario) est sélectionné pour être déployé.

```
1  Input: NoteQoS float[]
2  Output: the index of selected composition plan
3  int Selection(float [] NotesQoS)
4  {
5    int j=0;
6
7    float max=NotesQoS[0];
8
9    for(int i=1;i<NotesQoS.length;i++)
10   {
11     if (max<NotesQoS[i])
12     {
13       max=NotesQoS[i];
14       j=i;
15     }
16   }
17   return j;
18 }
```

Listing 5.5: Algorithme de sélection des plans de composition et des IaaS ayant la plus haute note QoS

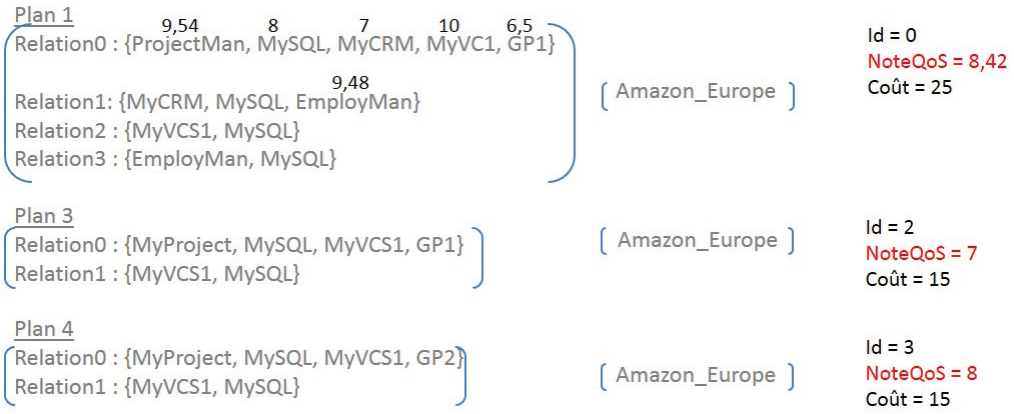


Figure 5.12: Notes QoS des plans de composition

5.2.6 Configuration et personnalisation de l'application

Certains services métier offrent des options de personnalisation. Pour le plan sélectionné, des formulaires Web de configuration sont affichés à l'utilisateur afin qu'il puisse personnaliser les services impliqués dans l'application générée. La configuration ne concerne que les services métier ayant des paramètres configurables. Les paramètres configurables peuvent concerner la personnalisation du design de l'application, par exemple, l'utilisateur peut intégrer le nom et le logo de son organisation ; ou l'administration de l'application, comme introduire le login et mot de passe de l'administrateur. Ces informations sont ensuite automatiquement traduites en script de configuration suivant l'orchestrateur utilisé.

Dans notre scénario, le plan de composition sélectionné est celui de ProjectMan. Seul ProjectMan offre des options de personnalisation. Automatiquement, le formulaire Web de configuration de ProjectMan, illustré ici par la Figure 5.13, s'affichera à l'utilisateur lui permettant de personnaliser le nom et le logo de l'application générée. Le nom et le logo représentent les paramètres configurables de ProjectMan. Une fois que l'utilisateur aura introduit ses données de configuration, un script de configuration est automatiquement généré en fonction des paramètres configurables des services du plan de composition sélectionné et des données de configuration de l'utilisateur. Le Listing 5.6 illustre l'algorithme de génération des scripts de configuration. Le Listing 5.7 illustre le script spécifique à l'orchestrateur "Juju" [juj] permettant de définir le nom de l'application du gestionnaire de projet (ligne 2), ainsi que son logo (ligne 3).

```
Input : List of String: conf (Configurable parameters),
        List of String: value (Configuration data),
        String name (Service name)

String script="#!/bin/bash";
for (int i =0; i<conf.size(); i++)
{
   script =script+"juju set "+name+" "+ conf.get(i)+ "="+ value.get(i);
}

Creation of the configuration file

Script writing on the configuration file
```

Listing 5.6: Algorithme de génération des scripts de configuration

```
#!/bin/bash
Juju set ProjectMan name=MyProjectMan
Juju set ProjectMan logo=C:\path\monlogo
```

Listing 5.7: Script de configuration de ProjectMan

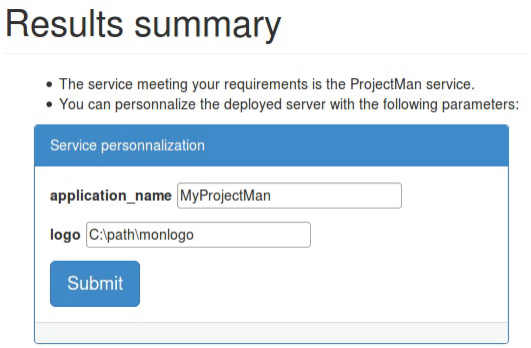


Figure 5.13: Formulaire Web pour la configuration de l'application

5.2.7 Déploiement automatique de l'application métier

La phase de déploiement consiste à déployer l'application générée à partir du plan de composition sélectionné. La Figure 5.14 illustre le déploiement du plan sélectionné. Le déploiement se fait par l'exécution d'un script afin de déployer des machines virtuelles (une par service métier) sur l'infrastructure IaaS sélectionnée. Les relations liant deux services et donc deux machines virtuelles sont ajoutées au script de déploiement déployant ainsi les services métier et considérant les relations entre ces derniers.

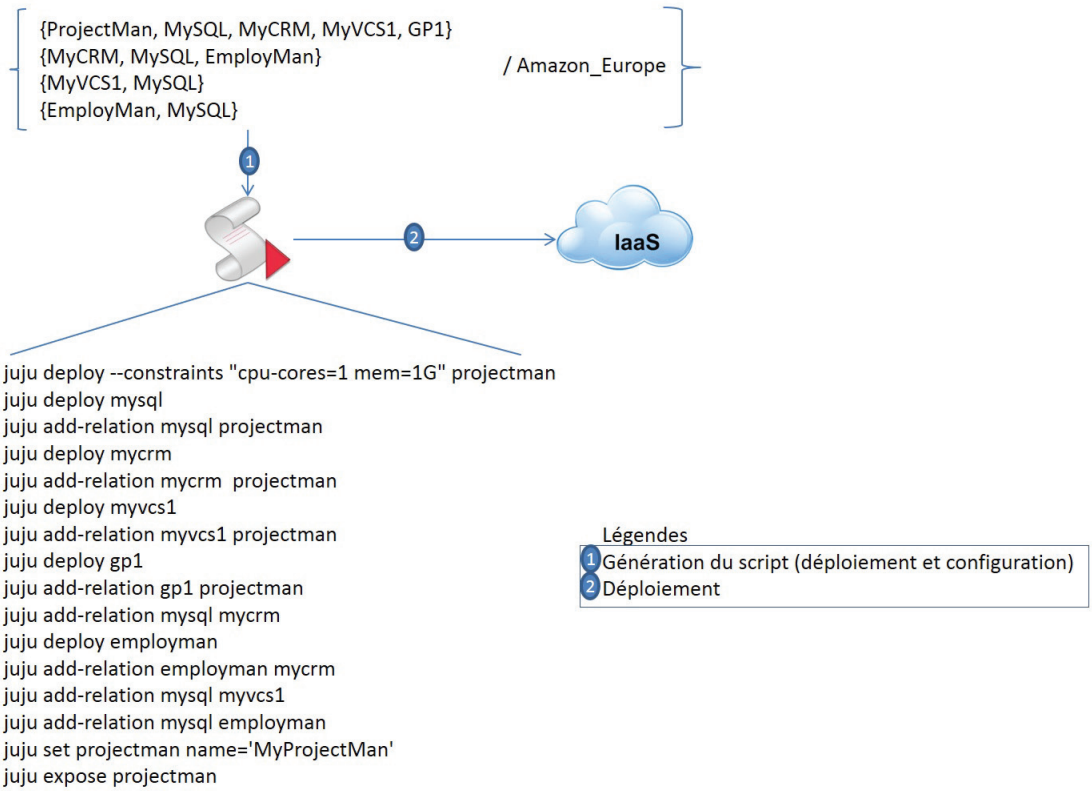


Figure 5.14: Déploiement de l'application métier

Le script de déploiement correspond à des lignes de commande dédiées à un outil d'orchestration de services ("Juju" dans notre implémentation) permettant la manipulation des services métier du plan de composition. En effet, l'outil d'orchestration nous permet de déployer et de composer des services en totale abstraction des détails techniques, à savoir (i) la gestion des dépendances entre les services, (ii) le déploiement des services sélectionnés, et (iii) la scalabilité des services déployés.

Le Listing 5.8 illustre les étapes de génération des scripts de déploiement. Ces derniers sont générés à partir du plan sélectionné comme suit : pour chaque relation du plan sélectionné, les services impliqués sont déployés en indiquant leurs contraintes de déploiement (lignes 10-12). Les services d'une relation (à partir du deuxième service) sont liés au premier service de cette dernière (lignes 13-16). Les méthodes "getCPU" et "getMemory" permettent de récupérer respectivement le nombre de CPU et la taille de la mémoire nécessaires au déploiement d'un service donné. Enfin, le premier service de la première relation est exposé (ligne 20) permettant ainsi à l'utilisateur d'accéder à l'application.

```
Input : Plan (represents the selected plan)
Output : script (deployment script)

for (int i =0, i<Plan.size, i++)
{
 for (int j =0; j<Plan.get(i).size; j++)
 {
  if (Plan.get(i).get(j) has not been deployed yet)
  {
   script=script+"Juju deploy --constraints
   cpu-cores="+getCPU(Plan.get(i).get(j))+
   "mem="+getMemory(Plan.get(i).get(j))+" "+Plan.get(i).get(j);
   if(j>1)
   {
    script=script+"juju add-relation "+Plan.get(i).get(j)+" "+Plan.get(i).get(0);
   }
  }
 }
}
script=script+"juju expose "+Plan.get(0).get(0);
```

Listing 5.8: Algorithme de génération des scripts de déploiement

Le Listing 5.9 illustre le script de déploiement généré automatiquement pour déployer ProjectMan avec l'orchestrateur "Juju". Nous prenons en compte, pour chaque service métier, les ressources minimales nécessaires à son déploiement (lignes 1,2,4,6,8,11).

```
1  juju deploy --constraints "cpu-cores=1 mem=2G" projectman
2  juju deploy --constraints "cpu-cores=1 mem=1G" mysql
3  juju add-relation mysql projectman
4  juju deploy --constraints "cpu-cores=1 mem=2G" mycrm
5  juju add-relation mycrm  projectman
6  juju deploy --constraints "cpu-cores=1 mem=1G" myvcs1
7  juju add-relation myvcs1 projectman
8  juju deploy --constraints "cpu-cores=1 mem=1G" gp1
9  juju add-relation gp1 projectman
10 juju add-relation mysql mycrm
11 juju deploy --constraints "cpu-cores=1 mem=2G" employman
12 juju add-relation employman mycrm
13 juju add-relation mysql myvcs1
14 juju add-relation mysql employman
15 juju expose projectman
```

Listing 5.9: Le script de déploiement généré pour le plan 1 de la Figure 5.12 avec l'orchestrateur "Juju"

5.3 Conclusion

Dans ce chapitre, nous avons décrit MADONA, notre méthodologie de provisionnement (composition et déploiement de services métier) d'applications métier orientées service sur les environnements cloud, qui réduit les connaissances techniques requises de l'utilisateur pour ce provisionnement. Le principal objectif de notre travail est de démocratiser le provisionnement d'applications en automatisant son cycle de vie. MADONA couvre le cycle de vie de provisionnement depuis la découverte de services au déploiement d'une application métier. Elle prend en entrée le fichier .rival décrivant les besoins de l'utilisateur, et le fichier .usdl décrivant les services de la marketplace. En fonction des besoins fonctionnels de l'utilisateur et des services métier disponibles,

MADONA génère plusieurs plans de composition. Ces derniers sont notés en fonction de la qualité des services métier impliqués dans ces plans et des contraintes de qualité imposées par l'utilisateur. Le plan de composition ayant la plus haute note QoS est sélectionné pour être déployé sur une IaaS automatiquement sélectionnée. La sélection d'IaaS dépend des préférences de l'utilisateur et de la qualité des IaaS disponibles. Les services métier impliqués dans le plan de composition sélectionné sont personnalisés par l'utilisateur. MADONA utilise un orchestrateur de services pour le déploiement, la configuration et la composition de services métier. Elle gère les contraintes et possibilités de composition de chaque service métier impliqué dans un plan de composition de manière automatique et dynamique.

Lorsqu'aucun service disponible ne répond aux besoins fonctionnels de l'utilisateur, ce dernier est guidé pour intégrer, s'il le souhaite, de nouveaux services développés par des tiers, à la marketplace. L'intégration de nouveaux services peut se faire aussi par un fournisseur de services qui souhaite publier son service dans la marketplace. Tout nouveau service intégré est automatiquement découvert, déployé et composé lors des prochains provisionnements.

Dans le chapitre suivant, nous décrivons la mise en œuvre de notre approche méthodologique par la construction de l'environnement AAPaaS (Automatic Application Provisioning as a Service) afin de montrer la faisabilité de notre approche.

Chapitre 6

Automatic Application Provisioning as a Service : Mise en œuvre du prototype

Sommaire

6.1 Introduction

Ce chapitre présente la mise en œuvre de l'environnement d'Automatic Application Provisioning as a Service (AAPaaS). AAPaaS est un environnement qui permet le développement orienté service et le déploiement automatiques d'applications cloud, où l'utilisateur introduit ses besoins depuis un formulaire Web, et où une application est générée, déployée et mise à sa disposition. Ce chapitre est organisé comme suit. La Section 6.2 décrit les technologies utilisées pour le développement du prototype d'AAPaaS. Les Sections 6.3 et 6.4 décrivent respectivement le déploiement et l'implémentation d'APaaS. La Section 6.5 conclut ce chapitre.

6.2 Technologies utilisées

Nous avons implémenté AAPaaS comme une application Web. Nous avons pour cela choisi le framework Grails [gra] qui permet le développement d'applications suivant le patron Modèle, Vue, Contrôleur (MVC). Une vidéo du prototype est disponible à l'adresse "liris.cnrs.fr/hind.benfenatki/demo.mp4".

Afin de créer et requêter (i) le fichier de description des besoins de l'utilisateur suivant le vocabulaire RIVAL (requirements.rival), et (ii) le fichier de description des services de la marketplace suivant le langage Linked USDL étendu (marketplace.usdl), nous avons eu recours à l'utilisation du framework à code source ouvert Jena [jen]. Ce dernier permet entre autres la création et la lecture de sources de données RDF et l'exécution des requêtes SPARQL sur ces données.

Afin d'automatiser le déploiement des applications générées par AAPaaS sur des environnements cloud, la question du choix de l'outil de déploiement et l'IaaS sur laquelle effectuer le déploiement a été posée. En effet, beaucoup d'efforts ont été consacrés à faire des formats, APIs, et langages pour la description des besoins de l'utilisateur en termes de ressources et infrastructures sur lesquelles sont déployées les applications cloud. Certains travaux permettent de décrire le besoin au niveau ressources cloud (ex : CPU, type d'instance, service d'authentification, etc), d'autres en revanche, permettent de décrire la structure de l'application à déployer (niveau composant) :

- Le niveau ressources : nous retrouvons dans cette catégorie les travaux décrivant des API et langages permettant l'instanciation de ressources. Ces API et langages peuvent être spécifiques à un fournisseur donné (ex : Amazon Web Services (AWS) CloudFormation [amaa]), ou encore être interopérables permettant ainsi de requêter des ressources depuis différents fournisseurs (ex : mOSAIC [mos]).

- Le niveau composant : nous retrouvons dans cette catégorie TOSCA (Topology and Orchestration Specification for Cloud Applications) [tos] et Juju [juj]. TOSCA,

un standard d'OASIS [oas], décrit une spécification de la topologie d'une application cloud illustrant la structure de l'application, ses composants, et les relations entre les composants. TOSCA permet aussi de décrire des plans de gestion pour le déploiement, la mise en fonctionnement, et la configuration automatiques de l'application. Juju [juj] est un outil à code source ouvert d'orchestration de services qui permet de déployer des services sur des environnements cloud en utilisant des charms. Les charms décrivent des fichiers de configuration YAML Ain't Markup Language (YAML) ainsi que des hooks. Les hooks sont des scripts qui permettent l'installation, le démarrage, l'arrêt, et la configuration d'un service et sa mise en relation avec un autre service. Juju permet de faire appel à ces hooks avec des scripts de haut niveau (proches du langage naturel, ex : juju deploy MySQL). Les services orchestrés par Juju peuvent être déployés sur plusieurs clouds tels que Amazon EC2 [amab], HP Cloud Services [hp], Microsoft Windows Azure [win], OpenStack [ope], etc. Ce qui différencie TOSCA de Juju c'est le niveau d'abstraction lors du déploiement d'applications. En effet, avec Juju, le détail de déploiement, de configuration, et de mise en relation entre services est décrit dans les hooks. L'utilisateur final fait appel à ces hooks par des scripts de haut niveau. De ce fait, la complexité de déploiement, de configuration, et de mise en relation entre services est fortement réduite.

Notre objectif étant d'automatiser le déploiement des applications abstraites générées par AAPaaS, nous avons fait le choix de travailler au niveau composant. En effet, un niveau d'abstraction élevé permet de se concentrer sur la structure de l'application (en termes de composants et de relations entre composants) plutôt que sur les détails techniques de déploiement. Pour l'implémentation d'AAPaaS, nous avons choisi Juju pour la gestion du déploiement, de la configuration, et de la mise en relation des services métier d'un plan de composition. Ce choix a été guidé par les raisons suivantes :

- Outil à code source ouvert et gratuit.

- Documentation disponible et bien illustrée (exemples, vidéos...).

- Prise en main très rapide.

- Une marketplace de services métier assez riche lui est associée. Elle nous a permis de tester nos scenarii.

- Juju permet le déploiement de services métier à l'aide de scripts de haut niveau (proche du langage naturel) qui font appel à des hooks (scripts shell, ou scripts créés en puppet [pup], chef[che], etc). Cela implique, une automatisation du déploiement beaucoup plus simple. En effet, les scripts de haut niveau permettent une description logique des services que l'on souhaite déployer (ex : Juju deploy MySQL) sans avoir à se soucier des détails techniques de déploiement.

- Juju permet le déploiement sur plusieurs infrastructures cloud (privées et publiques), et de faire des simulations sur une machine Ubuntu par l'utilisation de Linux Containers.

6.3 Déploiement d'AAPaaS

Nous avons déployé le prototype d'AAPaaS sur deux machines virtuelles sous Ubuntu 14.04 avec 4 Go de mémoire et 80 Go de disque avec une allocation dynamique. La Figure 6.1 illustre le déploiement de ce prototype. Nous avons installé le serveur de l'application AAPaaS sur une machine virtuelle et l'environnement Juju sur la seconde. Les deux machines virtuelles communiquent en utilisant Open-SSH.

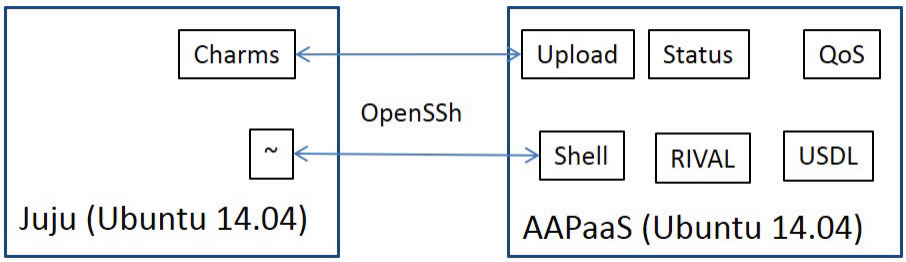


Figure 6.1: Déploiement d'AAPaaS

Chacune de ces deux machines virtuelles (serveur d'application AAPaaS et serveur Juju) contient des répertoires permettant de stocker les fichiers nécessaires à la découverte, à la notation, et au déploiement des services métier du plan de composition sélectionné. Le serveur d'application AAPaaS contient des répertoires qui ont été créés lors de l'implémentation du prototype et sont ensuite consultés et modifiés (si besoin) automatiquement (depuis le code). Ces répertoires sont les suivants :

- USDL : permet le stockage des descriptions des services métier de la marketplace. Toutes les descriptions de services sont consignées dans un seul fichier au format Turtle [tur], afin de permettre une recherche rapide et simultanée des services primaires et secondaires.

- QoS : permet le stockage des fichiers contenant les paramètres de qualité de service. Pour chaque service de la marketplace, un fichier XML existe portant le nom du service, et décrivant les valeurs de chaque paramètre de qualité de service[1]. Le Listing 4.1 (cf. Chapitre 4 page 64) illustre le fichier QoS au format XML d'un service métier.

[1]Comme exposé dans le Chapitre 4, dans ce travail nous considérons quatre paramètres de QoS : le temps de réponse, la disponibilité, la perte de données, et la confidentialité des données

- Rival : permet le stockage des fichiers .rival au format Turtle.

- Status : permet le stockage (i) du script permettant la récupération du statut des services déployés, et (ii) du fichier contenant le statut de déploiement des services déployés (fichier log). Le statut d'un service représente son état de déploiement. Nous considérons deux types de statuts : "en cours de déploiement" si le déploiement n'est pas terminé, l'adresse IP du service si le déploiement de ce dernier est terminé.

- Shell : permet le stockage des scripts de déploiement et de configuration générés. Il stocke aussi les scripts permettant l'envoi et l'exécution des scripts de déploiement et de configuration sur le serveur Juju. A titre d'exemple, le Listing 5.9 (cf, Chapitre 5 page 93) illustre le script de déploiement de l'application de gestion de projet. Le Listing 6.1 illustre le script permettant l'envoi (ligne 2) du script de déploiement (execute.sh) vers le serveur Juju et son exécution (ligne 3).

```
1 \#!/bin/bash
2 scp -i /root/.ssh/hind service/shell/execute.sh hind@192.168.255.5:~;
3 ssh -i /root/.ssh/hind hind@192.168.255.5 './execute.sh';
```

Listing 6.1: Script permettant l'envoi du script de déploiement vers le serveur Juju, et son exécution

- Upload : permet le stockage temporaire des charms (hooks et fichiers de configuration) des nouveaux services métier à intégrer dans la marketplace de services. Ces charms sont supprimés juste après leur transfert vers le serveur Juju.

Le serveur Juju quant à lui utilise le répertoire "Charms" pour stocker les charms envoyés depuis le serveur d'AAPaaS. Ces charms sont ceux des services nouvellement intégrés. Les scripts de déploiement et de configuration sont stockés à la racine du serveur Juju avant d'être exécutés.

6.4 Implémentation d'Automatic Application Provisioning as a Service

La Figure 6.2 illustre l'architecture du prototype d'AAPaaS. L'élicitation des besoins, la découverte de services métier, l'intégration de nouveaux services à la marketplace, la génération et notation de plans de composition, et le déploiement automatique sont mis en œuvre.

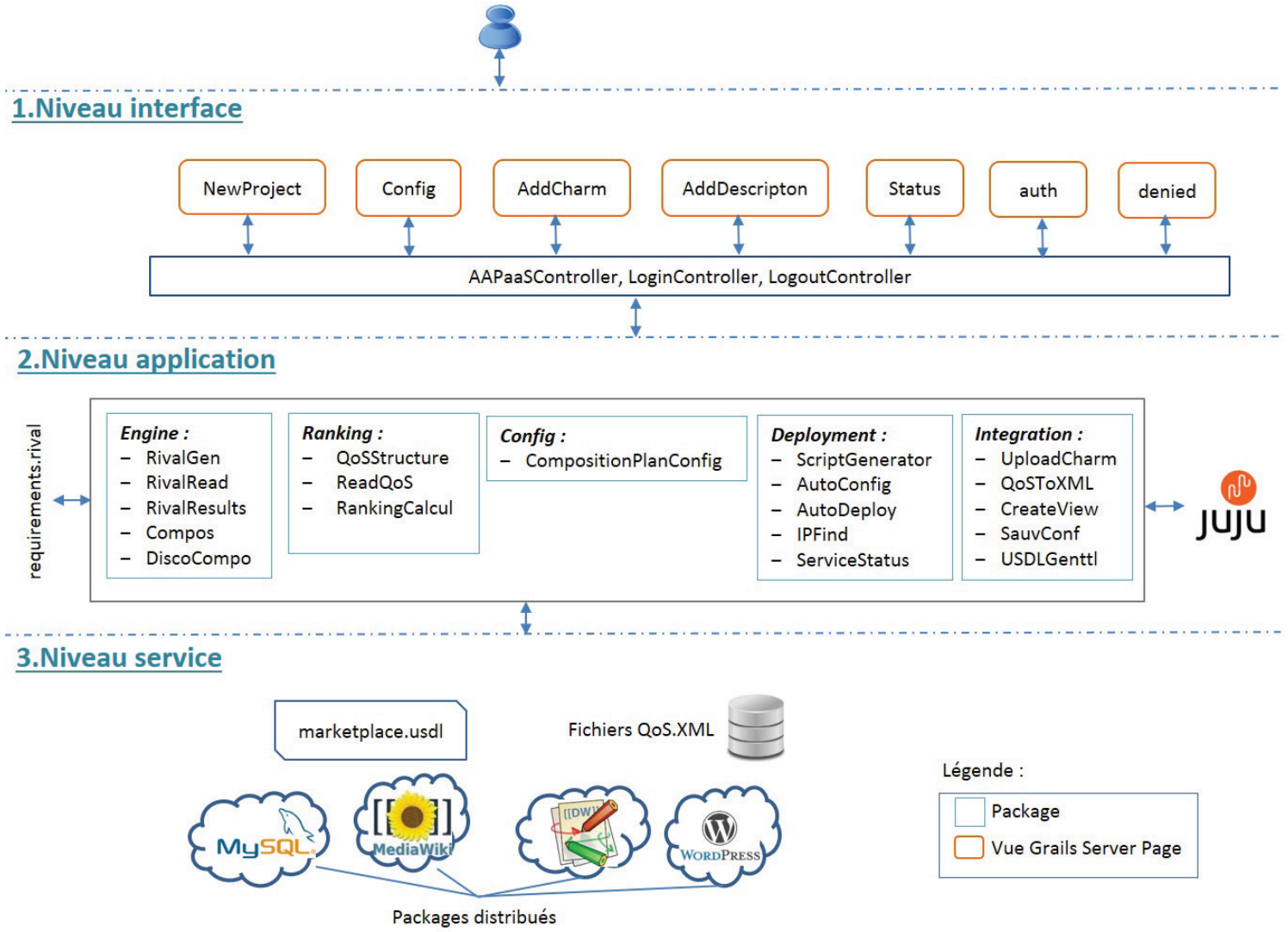


Figure 6.2: Architecture d'AAPaaS

Le prototype fonctionne comme suit : à partir des besoins de l'utilisateur, plusieurs plans de composition sont générés et notés. Le plan ayant la plus haute note QoS est sélectionné. Un script est généré et envoyé au serveur Juju pour le déploiement de ce plan. Nous rappelons que certains services métier fournissent certaines options de configuration (ex : le nom de l'application). Dans ce cas, après avoir sélectionné le plan de composition à déployer, une nouvelle page Web s'ouvrira fournissant une interface pour choisir ces options. Un nouveau script pour configurer le service est créé et envoyé au serveur Juju. Une fois que le processus est terminé, l'utilisateur est renvoyé vers la page Web affichant le statut de l'application générée. Le statut de chaque service est analysé pour tenir l'utilisateur au courant de l'état de déploiement de son application. Ce statut a deux valeurs possibles "en cours de déploiement" ou le lien (adresse IP) vers l'application si cette dernière est déjà déployée. Un rafraîchissement automatique toutes les trente secondes de la page Web permet le renvoi et l'exécution d'un script pour l'obtention du statut des services déployés. Dans le cas où aucun service de la marketplace ne répond aux besoins fonctionnels de l'utilisateur, ce dernier est dirigé vers les interfaces Web "AddCharm" et "AddDescription" qui permettent respectivement d'intégrer de nouveaux charms à la marketplace et de décrire le service nouvellement intégré.

AAPaaS est composé de trois niveaux comme suit :

1. **Le niveau interface** (Niveau 1 de la Figure 6.2) permet la communication avec l'utilisateur. Il est composé de contrôleurs et de vues représentant les interfaces Web de MADONA. Les contrôleurs sont au nombre de trois et ont les rôles suivants :

   - Deux contrôleurs (LoginController et LogoutController) se chargent de la gestion de la connexion et déconnexion des utilisateurs. Ces deux contrôleurs sont créés automatiquement par l'ajout du composant "Spring Security Core" de Grails.
   - Le troisième contrôleur (AAPaaSController) gère le passage d'une vue à une autre et l'appel des classes appropriées à une situation donnée. Il route les données en entrée (introduites par l'utilisateur via des interfaces Web) entre les différentes classes Java d'AAPaaS.

   Ce niveau est composé de cinq types de vues (hors page d'accueil et page de connexion). Les vues représentent des "Groovy Server Pages" (GSP). Les vues du niveau interface sont les suivantes :

   - La vue "NewProject" (illustrée par la Figure 6.3) décrit un formulaire Web permettant d'introduire les besoins pour un nouveau projet.
   - La vue "AddCharm" (illustrée par la Figure 6.4) permet de télécharger de nouveaux charms Juju.

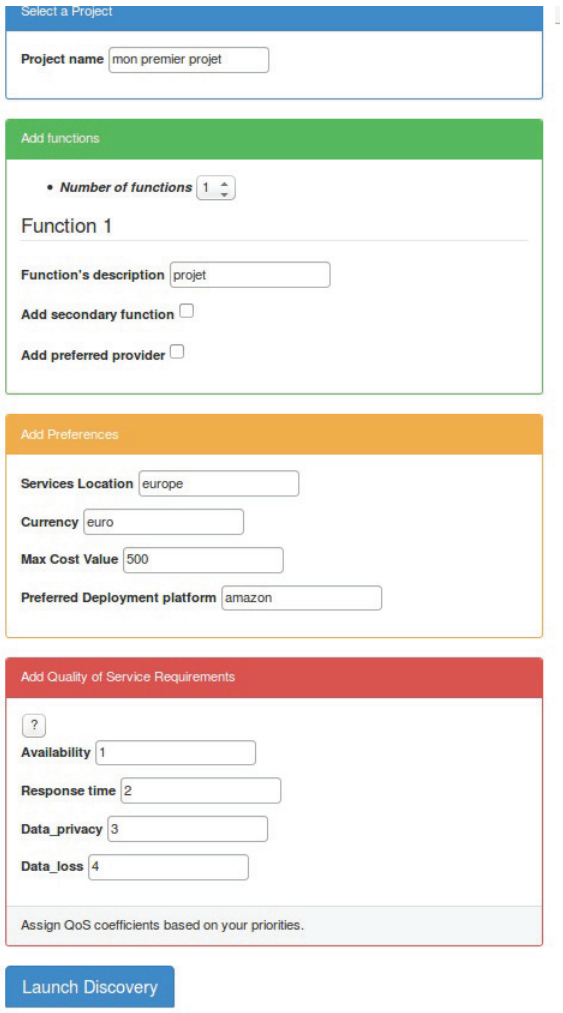


Figure 6.3: La vue "NewProject"

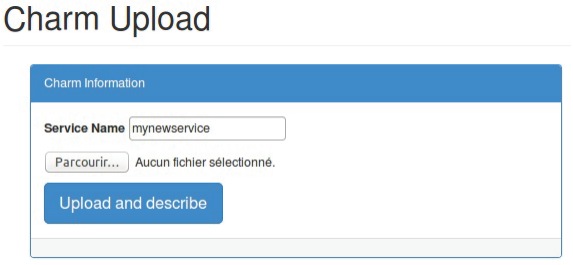


Figure 6.4: La vue "AddCharm"

- La vue "AddDescription" (illustrée par la Figure 6.5) permet de décrire les services métier intégrés.

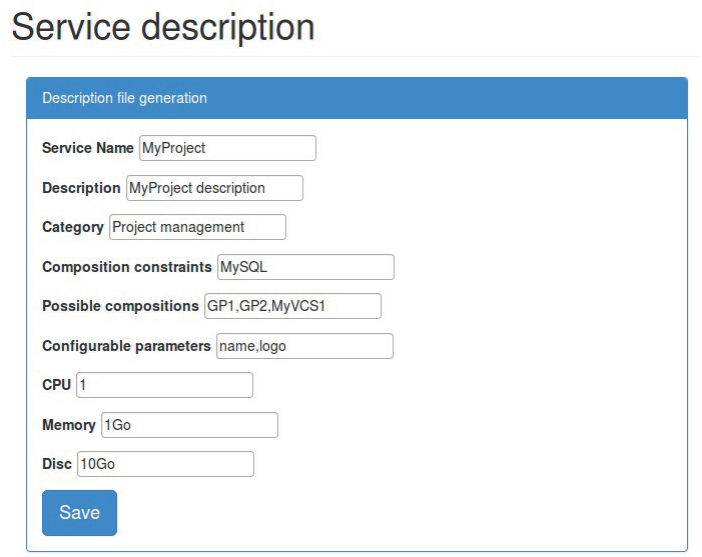


Figure 6.5: La vue "AddDescription"

- La vue "Status" (illustrée par la Figure 6.6) permet d'afficher le statut des services métier déployés.

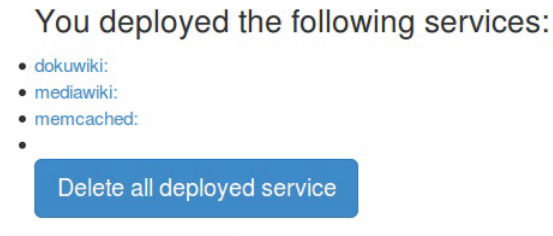


Figure 6.6: La vue "Status"

- A chaque service métier de la marketplace est associée une vue de configuration. Cette dernière est automatiquement créée lors de l'intégration du service à la marketplace, en fonction des paramètres configurables du service. Elle permet d'introduire les données de configuration de l'utilisateur. La Figure 5.13 (cf. Chapitre 5 page 91) illustre la vue de configuration générée pour le service ProjectMan.

2. **Le niveau application** (Niveau 2 de la Figure 6.2) permet la génération et le déploiement de plans de composition. Il est composé de classes Java, distribuées sur cinq packages (Engine, Ranking, Config, Deployment, Integration) comme suit :

   (a) ***Engine :*** ce package est responsable de générer les plans de composition à partir des besoins de l'utilisateur. Il contient cinq classes :

   - "RivalGen" permet de générer un fichier pour stocker les besoins de l'utilisateur exprimés sur le formulaire Web de la vue "NewProject" suivant

le vocabulaire RIVAL. Ce fichier est stocké sur le serveur de l'application dans un souci de réutilisation ultérieure par l'utilisateur. Le constructeur de cette classe crée un fichier requirements.rival et initialise un modèle avec les préfixes des vocabulaires utilisés pour la description des besoins de l'utilisateur (ex : le vocabulaire Good Relations). Le modèle est une instance de la classe "model" de l'API Jena qui décrit une structure de donnée qui stocke une collection de triplets RDF sous forme de graphe. Nous avons ajouté plusieurs méthodes dans la classe "RivalGen" pour permettre le remplissage du fichier requirements.rival avec les besoins de l'utilisateur. Le Listing 6.2 illustre la méthode "location" permettant de remplir le fichier RIVAL avec la localisation préférée de l'utilisateur. La ligne 3 permet la création de la propriété du vocabulaire RIVAL : "hasPreferredDeploymentLocation". La ligne 4 lie la ressource "project" avec la localisation préférée de l'utilisateur via la propriété "hasPreferredDeploymentLocation".

```
1 public void location(String location)
2 {
3   Property location1 = model.createProperty(RIVAL,"
        hasPreferredDeploymentLocation");
4   project.addProperty(location1,location);
5 }
```

Listing 6.2: Méthode pour l'introduction de la localisation préférée de l'utilisateur dans le fichier RIVAL

- La classe "RivalRead" permet d'extraire les informations décrites sur un fichier requirements.rival préalablement généré et stocké. Ces informations sont utilisées pour la génération de requêtes SPARQL lors de la découverte des services métier.
- La classe "RivalResults" définit un objet Java pour la description de la structure des résultats de la lecture d'un fichier requirements.rival. A titre d'exemple, la description de la fonctionnalité (primaire et secondaire) désirée est représentée par une chaine de caractère, le poids associé à chaque paramètre QoS est un décimal.
- La classe "Compos" définit un objet Java pour la description de la structure d'un plan de composition. Dans notre implémentation, nous avons représenté les plans de composition par une liste de relations, où chaque relation représente une chaine de caractères qui débute par un double slash "//". Les services métier impliqués dans chaque relation sont séparés par un "/".
- La classe "DiscoCompo" permet grâce à la méthode "MainDiscovery" (i) de générer une requête SPARQL en fonction des besoins de l'utilisateur (appel à la méthode "request", et aux méthodes de la classe "RivalRead"

pour la récupération des besoins de l'utilisateur), (ii) de lancer la requête SPARQL sur les descriptions USDL étendu des services de la marketplace, (iii) de composer les services métier découverts en fonction des contraintes et possibilités de composition de chacun et des besoins fonctionnels de l'utilisateur (méthodes "searchSparqlQuery", "verifConstraints", et "getConstraints"), et (iv) de faire appel à la classe "RankingCalcul" du package "Ranking" afin de sélectionner le plan de composition ayant la plus haute note QoS. La méthode "getConstraints" (Listing 6.3) permet d'exécuter une requête SPARQL (ligne 12) sur le fichier marketplace.usdl afin de récupérer les contraintes de composition d'un service donné à travers la variable "constraint".

```
1  ArrayList<String> getConstraints(String service)
2  {
3    ArrayList<String> constraint;
4    String query= "PREFIX usdl: <http://www.linked-usdl.org/ns/usdl\#>\n" +
5    "PREFIX gr:<http://www.heppnetz.de/ontologies/goodrelations/v1#>\n" +
6              "SELECT ?c WHERE {\n" +
7              "?x gr:name ?a FILTER regex(?a,\""+service+ "\",\"i\").\n"+
8              "?x usdl:hasHardCompositionConstraints ?b.\n"+
9              "?b gr:name ?c.}\n";
10   Query queryy = QueryFactory.create(query);
11   QueryExecution qe = QueryExecutionFactory.create(queryy, model);
12   ResultSet results = qe.execSelect();
13   while (results.hasNext())
14   {
15     QuerySolution qs=results.next();
16     try
17     {
18       constraint.add(qs.get("c"));
19     }
20     catch(NullPointerException e)
21     {
22       System.out.println("error while getting constraints for a service");
23     }
24   }
25   return constraint;
26 }
```

Listing 6.3: Méthode getConstraints permettant de récupérer les contraintes de composition d'un service

(b) ***Ranking :*** ce package s'occupe du calcul de la note QoS d'un service et d'un plan de composition. Il est composé de trois classes :

- La classe "QoSStructure" représente un objet Java qui décrit la structure des paramètres de qualité de services. Dans ce travail, nous avons considéré quatre paramètres de QoS, et la valeur de chaque paramètre est représentée par un décimal.
- La classe "readQoS" permet de parcourir le fichier QoS au format XML d'un service donné et de récupérer pour chaque paramètre de qualité de service, sa valeur.
- La classe "RankingCalcul" calcule la note QoS d'un service et d'un plan de composition, et récupère le plan avec la meilleure note. Le constructeur de cette classe prend en entrée la liste des plans de composition et

les besoins de qualité de services de l'utilisateur, i.e., les poids affectés à chaque paramètre QoS lors de l'élicitation des besoins. Pour chaque plan de composition, le constructeur récupère les services impliqués (méthode "ServPourRank"), et fait appel à la méthode "CompoRanking" qui calcule la note QoS de chaque plan de composition. Ces notes QoS sont stockées dans une liste et sont classées pour récupérer l'indice du meilleur plan de composition grâce à la méthode "OrderRanking". La méthode "RankService" calcule la note QoS d'un service à partir de son nom. La méthode "maxValues" récupère pour un service donné, les valeurs maximales de chaque paramètre de qualité de service entre les services de même catégorie, ex : MySQL, PostgreSQL sont de la catégorie base de données. La méthode "getDescription" récupère la catégorie d'un service à partir de son nom.

(c) ***Config :*** ce package contient la classe "CompositionPlanConfig" qui permet de générer le script Juju de configuration d'un service métier à partir des données de configuration entrées par l'utilisateur via le formulaire Web de configuration. Le constructeur de cette classe prend en entrée le nom du service à configurer, ses paramètres configurables, ainsi que les données de configuration pour générer le script de configuration, qui sera stocké dans le répertoire /shell de MADONA.

(d) ***Deployment :*** ce package est responsable du déploiement de l'application correspondant au plan de composition sélectionné. Il est constitué de cinq classes :

- La classe "ScriptGenerator" permet de générer le script Juju de déploiement de l'application sélectionnée suivant l'algorithme décrit dans le Listing 5.8 (cf. Chapitre 5 page 93). Le constructeur de cette classe s'assure de l'existence (ou le crée à défaut) du fichier execute.sh dans le répertoire /shell de MADONA. Ce fichier contiendra le script de déploiement du plan de composition sélectionné.
- Les classes "AutoConfig" et "AutoDeploy" permettent d'envoyer au serveur Juju via Open-SSH et d'exécuter respectivement les scripts de configuration et de déploiement. L'utilisateur est alors renvoyé à la vue "Status" où l'application déployée est mise à sa disposition avec le statut "en cours de déploiement".
- La classe "IPFind" permet de lancer un script de récupération du statut des services déployés sur le serveur Juju et de renvoyer l'adresse IP de ces derniers afin que l'utilisateur puisse accéder à l'application une fois déployée. Ce script est exécuté à chaque actualisation automatique de la vue "Status". Une actualisation a lieu toutes les trente secondes. Ce statut est analysé afin de récupérer le nom des services et leur statut : "en

cours de déploiement" si le service n'est pas encore prêt à être utilisé, son adresse IP sinon. Ce résultat sera ensuite affiché sur la vue "Status". Le constructeur de cette classe permet d'exécuter le script "test.sh" du Listing 6.4 stocké dans le répertoire Status de MADONA. Ce script permet de récupérer le statut des services déployés avec Juju dans le fichier status.txt (ligne 2) et de rappatrier ce fichier dans le répertoire "Status" de MADONA (ligne 3) avant de l'analyser.

```
1 \#!/bin/bash
2 ssh -i /root/.ssh/hind hind@192.168.255.5 'juju status > status.txt';
3 scp -i /root/.ssh/hind hind@192.168.255.5:~/status.txt Status/;
```

Listing 6.4: Script de récupération du statut de Juju

La méthode "serviceList" permet de récupérer le statut de chaque service depuis le contenu du fichier status.txt, et associe à chaque nom de service son statut.

- La classe ServiceStatus définit l'objet Java du statut d'un service. Cet objet est composé du nom du service et de son URL.

(e) ***Integration :*** pour permettre l'intégration de nouveaux services métier à la marketplace, nous avons implémenté cinq classes. Ces classes sont les suivantes :

- La classe "UploadCharm" permet d'exécuter le script charm.sh, stocké dans le répertoire shell. Ce script permet de transférer une archive contenant les charms d'un nouveau service métier, depuis le serveur de l'application vers le serveur Juju.
- La classe "QoSToXML" génère les valeurs des paramètres de qualité de service pour un service donné et les stocke dans le répertoire QoS, dans un fichier XML ayant pour nom le nom du service concerné (*nom-du-service*-QoS.xml). Pour tester notre prototype d'une manière contrôlée, ces valeurs sont générées de façon aléatoire dans un intervalle prédéfini pour chaque paramètre de qualité de service. Les fichiers QoS XML sont utilisés dans la phase de notation des plans de composition.
- La classe "CreateView" permet de générer la vue de configuration du service nouvellement intégré, sur la base d'un modèle que nous avons prédéfini. Ce modèle de vue est personnalisé automatiquement en fonction du nom du service et de ses paramètres configurables.
- La classe "SauvConf" permet de modifier le code source du contrôleur de l'application "AAPaaSController" afin qu'il intègre l'action de configuration pour le service nouvellement introduit. Le constructeur de cette classe prend en entrée les paramètres configurables du service à intégrer ainsi que son nom. Il parcourt le fichier de code du contrôleur à

la recherche de la chaine de caractère indiquant l'endroit où le code de l'action de configuration doit être placé. Nous avons utilisé pour cela la chaine suivante : //placer la sauvegarde ici. Cette chaine est remplacée par le code de la méthode de configuration du service intégré. Ce code est généré grâce à la méthode "textdeconf" (Listing 6.5) qui prend en entrée le nom du service et ses paramètres configurables. Le Listing 6.6 illustre le code groovy généré et inséré dans le code du contrôleur pour le service "NewService", nouvellement intégré, avec deux paramètres configurables : nom et logo. Il permet (i) d'afficher la vue "Status", (ii) pour chaque paramètre configurable, de récupérer et sauvegarder les données de configuration introduites par l'utilisateur (lignes 6-11). "params.logo" permet de récupérer la valeur introduite dans un formulaire Web dont le composant Web (ex : zone de texte) porte le nom "logo", (iii) d'instancier la classe CompositionPlanConfig du package Config permettant de générer le script de configuration (ligne 12), et la classe AutoConfig pour envoyer et exécuter ce script sur le serveur Juju (ligne 13). La ligne 15 permet de reconnaître le lieu d'insertion du prochain code de l'action de configuration pour les services à intégrer. Dans les Listings 6.5 et 6.6, les variables "conf" et "valeurs" représentent respectivement les paramètres configurables et les données de configuration du service à intégrer.

```
1  public String textdeconf(String name, ArrayList<String> conf)
2  {
3    String retour;
4    retour = "def conf"+name+"()\r\n";
5    retour = retour+"{ \r\n";
6    retour = retour+"def conf = []\r\n";
7    retour = retour+"def valeurs = []\r\n";
8    retour=retour+"redirect(action:\"Status\")\r\n";
9    for (int i=0; i<conf.size(); i++)
10   {
11     if (conf.get(i) != null&& ! conf.get(i).isEmpty())
12     {
13       retour=retour+"def "+conf.get(i) +"=params."+conf.get(i)+"\r\n";
14       retour=retour+"conf.add(\""+conf.get(i)+"\")\r\n";
15       retour=retour+"valeurs.add("+conf.get(i)+")\r\n";
16     }
17   }
18   retour=retour+"CompositionPlanConfig config= new CompositionPlanConfig(conf,
         valeurs, \""+name+"\");\r\n";
19   retour=retour+"AutoConfig auto = new AutoConfig();\r\n";
20   retour=retour+"}\r\n";
21   retour=retour+"//placer la sauvegarde ici";
22   return retour;
23 }
```

Listing 6.5: Méthode "textdeconf" permettant la génération du code de l'action de configuration du service intégré

- La classe "GenUSDLttl" permet de générer et de stocker des descriptions suivant le langage Linked USDL étendu, au format Turtle, à partir d'une description donnée introduite via le formulaire Web de la vue "AddDe-

scription". Plusieurs méthodes ont été ajoutées dans cette classe permettant de remplir la description du service par son nom, ses contraintes et possibilités de composition, sa catégorie, etc.

```
def confNewService()
{
  def conf = []
  def valeurs = []
  redirect(action:"Status")
  def name=params.name
  conf.add("name")
  valeurs.add(name )
  def logo=params.logo
  conf.add("logo")
  valeurs.add(logo)
  CompositionPlanConfig config= new CompositionPlanConfig(conf, valeurs, "
      NewService");
  AutoConfig auto = new AutoConfig();
}
//placer la sauvegarde ici
```

Listing 6.6: Code (groovy) généré pour l'action de configuration d'un nouveau service intégré à la marketplace

3. **Le niveau service** (Niveau 3 de la Figure 6.2) représente la marketplace de services. Il est composé des descriptions USDL étendu des services de la marketplace, des fichiers QoS au format XML, et des packages distribués des services métier. Les éléments de la marketplace sont utilisés lors de la découverte, de la composition, de la notation, de la configuration, et du déploiement de services.

6.5 Conclusion

Dans ce chapitre, nous avons défini l'architecture d'AAPaaS, un environnement de provisionnement automatique et dynamique d'applications cloud orientées service. L'environnement d'AAPaaS a été développé comme étant une application Web qui permet à l'utilisateur d'introduire ses besoins fonctionnels et non fonctionnels puis à utiliser l'application déployée. Nous avons choisi Grails comme framework de développement d'applications suivant le patron Modèle, Vue, Contrôleur.

AAPaaS repose sur l'orchestrateur Juju pour la gestion du déploiement, de la configuration, et de la composition de services métier. Il génère les scripts de déploiement et de configuration, et les exécute sur le serveur Juju via Open-SSH.

L'architecture d'AAPaaS contient trois niveaux :

- Le niveau interface permet le dialogue entre le système d'AAPaaS et l'utilisateur. Il décrit les différentes interfaces Web développées, et gère le flux de données qui transite entre le système et l'utilisateur.

- Le niveau application est le cœur du système d'AAPaaS. Il décrit les différentes classes développées.

- Le niveau service décrit la marketplace de services. Cette dernière est constituée des services métier, du fichier de description de services, et des fichiers XML de description des paramètres QoS des services.

Dans le chapitre suivant, nous évaluons notre travail de manière qualitative et quantitative, en le comparant à d'autres solutions.

Chapitre 7

Expérimentations

Sommaire

7.1 Introduction

Ce chapitre évalue notre travail qualitativement et quantitativement. Automatic Application Provisioning as a Service (AAPaaS) est comparé avec :

- Le déploiement utilisant Juju.
- Le déploiement local sur une machine Ubuntu.
- Le déploiement sur la plateforme d'hébergement et de provisionnement d'applications cloud Bitnami. La plateforme Bitnami permet de déployer des applications cloud d'une manière simple et automatisée. L'utilisateur doit sélectionner l'application appropriée, la localisation pour le déploiement, le système d'exploitation, le type de serveur, la taille du disque, les options d'application tels que le login et le mot de passe de cette dernière, les options de développement pour inclure l'installation de serveurs Web ou système de gestion de version, et les propriétés d'application tels que la langue ou le pseudo. Ces entrées ont des valeurs par défaut pour permettre à l'utilisateur de déployer l'application désirée plus simplement.

La comparaison se fait sur la base des critères suivants :

- La réduction des connaissances techniques requises de l'utilisateur pour le provisionnement d'applications cloud. Cette dernière est évaluée par le nombre de services et lignes de script à connaître suivant le scénario utilisé.
- Les performances en termes de temps de provisionnement.
- L'intérêt de l'introduction des contraintes et possibilités de composition de services dans la description des services métier.
- L'enrichissabilité de la marketplace de services.

Afin d'évaluer AAPaaS, nous avons utilisé trois scenarii. Les premier et deuxième scenarii servent à évaluer le provisionnement des applications MediaWiki et WordPress. Le choix de ces dernières est dû à la disponibilité de leurs services dans la marketplace.

Le troisième scénario consiste à provisionner une application dont les fonctionnalités ne sont satisfaites par aucun service métier des markertplaces des deux plateformes Bitnami [bit] et AAPaaS, et simulons la réaction de ces deux plateformes face à cette situation.

Le reste de ce chapitre est organisé comme suit. Les évaluations qualitative et quantitative sont décrites respectivement en Sections 7.2 et 7.3. La Section 7.4 conclut ce chapitre.

7.2 Évaluation qualitative

Nous comparons dans le Tableau 7.1 les quatre approches sur la base de leurs prérequis pour le provisionnement d'applications, leur processus de découverte, leur gestion des dépendances entre services, et leur gestion du déploiement.

Tableau 7.1: Comparaison des quatre approches

Critères	Déploiement local	Avec Juju	Avec Bitnami	Avec AAPaaS
Pré-requis	- Serveur Web - Packages des différents services métier	Serveur Juju	Navigateur Web	Navigateur Web
Processus de découverte	Manuel	L'utilisateur sélectionne le service à déployer parmi ceux disponibles	L'utilisateur sélectionne l'application à déployer parmi celles disponibles	Automatique, basé sur les fonctionnalités désirées ou sur le nom des services
Gestion des dépendances entre services	Manuelle	Manuelle	Automatique et statique	Automatique et dynamique
Gestion du déploiement	Manuelle	Nécessite des scripts de déploiement	Automatique	Automatique

Il ressort de notre analyse que le prototype d'AAPaaS est proche de Bitnami dans le fait que le processus de provisionnement est complètement automatique dans les deux approches et ne requière aucune pré-installation. Cependant, ces deux systèmes diffèrent essentiellement dans :

- La sélection des services (Tableau 7.2).
- Les connaissances techniques nécessaires au provisionnement d'applications cloud (Figure 7.6 et Tableau 7.2).
- Le cycle de vie de provisionnement comme illustré dans le Tableau 7.2.

Depuis notre analyse résumée dans le Tableau 7.2, il ressort que :

- Le processus de découverte en utilisant AAPaaS réduit les connaissances techniques parce que les utilisateurs sont invités à fournir les informations sur les fonctionnalités nécessaires à la place du nom de l'application. Cela est illustré dans les Figures 7.1 et 7.2 qui décrivent respectivement le provisionnement de l'application MediaWiki (composée de trois services métier) du point de vue utilisateur, et de la gestion des services inclus dans l'application générée. Avec Bitnami, c'est l'utilisateur qui choisit les services à déployer. De plus, avec Bitnami MySQL est présélectionné pour l'installation sur toutes les VM.

Tableau 7.2: Bitnami *versus* Juju *versus* AAPaaS

Phase	Bitnami	Juju	AAPaaS
Elicitation des besoins	L'utilisateur choisit explicitement l'application à déployer depuis un ensemble d'applications disponibles	L'utilisateur choisit explicitement les services métier à déployer et composer depuis un ensemble de services disponibles	L'utilisateur décrit ses besoins fonctionnels (fonctionnalités primaire et secondaires) en termes de mots clés ou de noms des services désirés
Découverte de services métier	Pas de découverte puisque l'utilisateur choisit l'application à déployer	Pas de découverte puisque l'utilisateur choisit les services à déployer et composer	Les services sont découverts en fonction des fonctionnalités et/ou des services désirés (recherche par nom ou par mot clé). Les contraintes et possibilités de composition des services découverts sont aussi prises en considération
Intégration de nouveaux services	Est possible seulement pour le fournisseur. L'utilisateur a toutefois la possibilité d'importer ses VMs depuis Amazon EC2	L'utilisateur a la possibilité d'intégrer de nouveaux services à la marketplace en intégrant les scripts de déploiement, de configuration et de composition	L'utilisateur a la possibilité d'intégrer de nouveaux services à la marketplace en intégrant la description du nouveau service et les scripts de déploiement, de configuration et de composition. Les services nouvellement intégrés sont automatiquement pris en compte dans les provisionnements futurs grâce à leurs descriptions
Génération des plans de composition	N'existe pas. Les possibilités de composition ne sont pas prises en compte. L'utilisateur sélectionne l'application à déployer	N'existe pas. Les contraintes de composition sont connues mais ne sont pas prises en compte automatiquement. L'utilisateur sélectionne les services à déployer et composer	Les contraintes et possibilités de composition sont connues (depuis la description des services) et prises en compte automatiquement pour la génération des plans de composition
Découverte d'IaaS	N'existe pas. L'utilisateur doit choisir l'IaaS de déploiement. Il doit connaître (ou se renseigner sur) les prix d'une VM donnée sur une IaaS donnée	N'existe pas. L'utilisateur doit choisir l'IaaS de déploiement. Il doit connaître (ou se renseigner sur) les prix d'une VM donnée sur une IaaS donnée	Une IaaS est sélectionnée automatiquement en fonction des préférences de l'utilisateur (localisation et fournisseur préférés), des besoins en QoS, et du coût à ne pas dépasser
Notation et sélection des plans de composition	Pas pris en compte	Pas pris en compte	Les plans de composition sont notés pour sélectionner le meilleur (en termes de QoS)
Configuration de l'application métier	A travers une interface Web et spécifique à chaque application	Au travers de scripts de configuration et spécifique à chaque service	A travers une interface Web et spécifique à chaque service
Déploiement de l'application métier	Un type de VM doit être sélectionné pour le déploiement de l'application. Une valeur par défaut existe : l'utilisateur doit connaître la configuration minimale permettant le bon fonctionnement de l'application	Un type de VM doit être sélectionné pour le déploiement des services. Il doit être mentionné dans le fichier de configuration Juju. Une valeur par défaut existe : l'utilisateur doit connaître la configuration minimale permettant le bon fonctionnement de chaque service de l'application. L'utilisateur crée le script de déploiement	Les ressources minimales requises (CPU, mémoire, et disque) sont connues pour chaque service permettant de le déployer dans une VM ayant des ressources suffisantes. Cela est pris en compte automatiquement. Le script de déploiement est automatiquement généré et exécuté

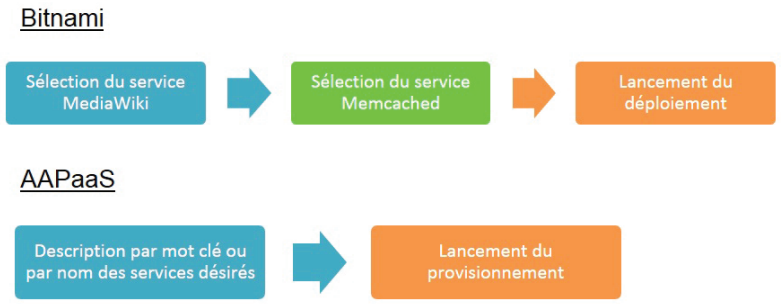


Figure 7.1: Comparaison entre Bitnami et AAPaaS pour la sélection des services de l'application MediaWiki

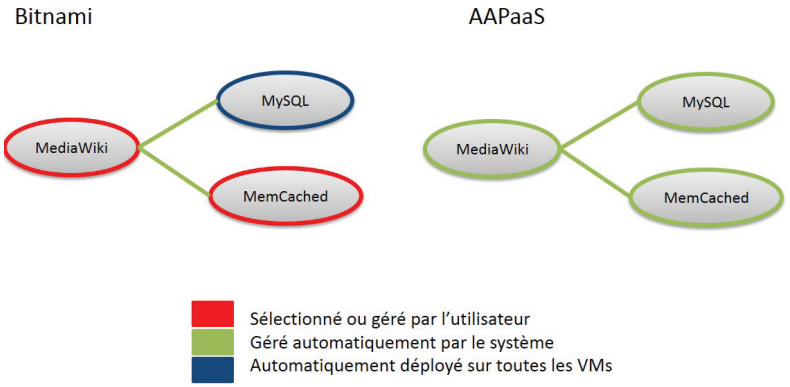


Figure 7.2: Comparaison entre Bitnami et AAPaaS pour la gestion des dépendances des services impliqués dans l'application MediaWiki

- Le processus de génération de plans de composition réduit les connaissances techniques nécessaires à l'utilisateur étant donné que les contraintes et possibilités de composition sont prises en compte de manière automatique dans la génération d'un plan de composition. L'utilisateur ne se préoccupe donc que par la description des fonctionnalités désirées.

- La gestion des contraintes et des possibilités de composition avec AAPaaS se fait de façon dynamique (pour chaque nouveau projet) rendant le processus de provisionnement plus générique et la marketplace enrichissable. La Figure 7.3 décrit le cas où les services de la marketplace ne répondent pas aux besoins de l'utilisateur. Alors qu'AAPaaS permet à l'utilisateur d'enrichir la marketplace en ajoutant des services externes, Bitnami permet simplement aux utilisateurs d'importer leurs instances EC2. Avec AAPaaS, les nouveaux services sont automatiquement intégrés dans la marketplace et utilisés (découverts, composés et déployés) dans des provisionnements futurs.

- Avec Bitnami, l'utilisateur doit connaitre les ressources nécessaires au déploiement

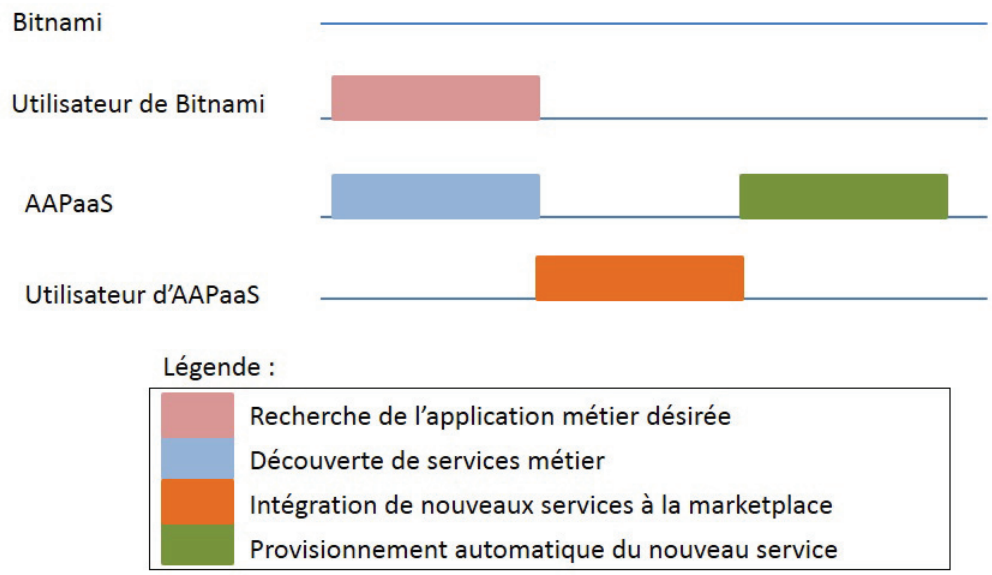


Figure 7.3: Comparaison entre Bitnami et AAPaaS lorsqu'aucun service de la marketplace ne répond aux attentes de l'utilisateur

de son application même si des valeurs par défaut existent. Alors que pour AAPaaS, cette information est connue pour chaque service métier, et prise automatiquement en compte lors de la phase de déploiement.

7.3 Évaluation quantitative

Nous avons évalué les performances, notamment le temps de provisionnement d'applications, du prototype d'AAPaaS (Section 7.3.1). Nous comparons ce dernier avec les autres approches en fonction des connaissances techniques à avoir pour le provisionnement d'applications, en termes de lignes de script à écrire et de nombre de services métier à connaître (Section 7.3.2). Dans la section 7.3.3, nous évaluons le nombre de services invoqués en fonction du nombre de fonctionnalités désirées et du nombre de contraintes de composition de services considérées. Dans la section 7.3.4, nous comparons le nombre de plans de composition générés en fonction du nombre de services découverts et des possibilités de composition de services considérées.

7.3.1 Évaluation des performances du prototype d'AAPaaS

La Figure 7.4 illustre le temps de provisionnement de MediaWiki et WordPress avec Bitnami, Juju, et AAPaaS. Le but d'évaluer le temps de provisionnement avec Bitnami et AAPaaS est de montrer qu'AAPaaS provisionne les applications métier en un temps satisfaisant en comparaison avec une solution industrielle. Les applications MediaWiki

et WordPress sont composées de trois services métier, chacune. Le choix d'évaluer le temps de provisionnement avec les applications MediaWiki et WordPress est dû au fait que les services impliqués dans ces applications sont disponibles dans Bitnami et Juju. Les temps de découverte et de déploiement sont difficiles à évaluer pour Juju et le déploiement local en raison de la difficulté d'estimation du temps d'un travail manuel. En effet, le processus de découverte varie en fonction de comment il est fait, des critères de sélection pris en compte, etc. Le temps de déploiement varie en fonction de la façon dont l'utilisateur est familier avec ce type d'installation. Nous avons toutefois évalué le temps de provisionnement avec Juju en partant du principe que l'utilisateur connaissait les services à déployer et comment les déployer.

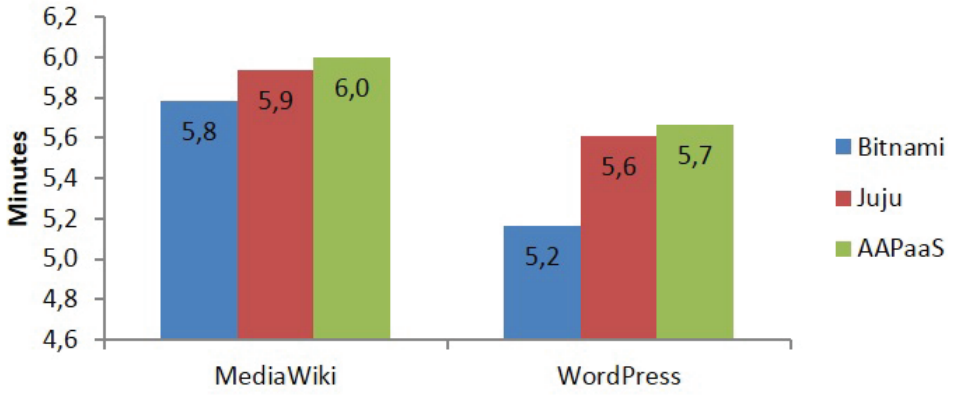


Figure 7.4: Temps de provisionnement de MediaWiki et WordPress avec Bitnami, Juju, et AAPaaS

Le provisionnement de MediaWiki et WordPress utilisant AAPaaS consomme plus de temps qu'avec Bitnami. Une augmentation de 13 secondes (+3%) est observée lors du provisionnement de MediaWiki. Cela dit, le processus de provisionnement est différent entre AAPaaS et Bitnami. En effet, AAPaaS génère et note plusieurs plans de composition répondant aux exigences de l'utilisateur, puis déploie celui avec la plus haute note QoS, tandis que Bitnami permet à l'utilisateur de sélectionner l'application nécessaire et procède à son déploiement. Dans Bitnami, la composition des services est statique, i.e., elle ne compose pas les services à la volée en fonction des besoins de l'utilisateur.

Le provisionnement de MediaWiki et WordPress avec AAPaaS consomme plus de temps qu'avec Juju. Une augmentation de 3 secondes (+1%) est observée lors du provisionnement de MediaWiki. Cependant, AAPaaS automatise les phases avant le déploiement et est basé sur Juju.

Nous avons choisi d'évaluer le temps de provisionnement au lieu d'évaluer la consommation des ressources pour analyser les performances d'AAPaaS pour les raisons suivantes :

- Bitnami est une solution industrielle. Il est donc difficile de comparer Bitnami et AAPaaS parce que nous ne disposons pas d'informations relatives à l'allocation et

la consommation des ressources pour le provisionnement avec Bitnami.

- Optimiser le coût de calcul au travers du temps de provisionnement et de la consommation des ressources est hors de la portée de ce travail.

Malgré un temps de provisionnement plus grand en utilisant AAPaaS, le prototype permet de composer les services métier à la volée et automatiquement, répondant aux besoins de l'utilisateur. AAPaaS réduit les connaissances techniques requises pour provisionner toute application cloud orientée service (Figures 7.6, 7.7). En effet, chaque phase d'AAPaaS est automatisée, et les besoins de l'utilisateur sont exprimés à un haut niveau d'abstraction des détails techniques (en termes de fonctionnalités, de besoins de qualité de service, et des préférences de déploiement et de coût). Inversement, Bitnami déploie des applications existantes dans la marketplace, et Juju requière l'intervention de l'utilisateur dans la sélection des services requis et dans la génération du script de déploiement.

La Figure 7.5 illustre le temps d'exécution des phases de MADONA suivant trois scenarii variant le nombre (i) de fonctionnalités désirées, (ii) de plans de composition générés, et (iii) des services et relations dans chaque plan de composition. Nous rappelons qu'une relation lie un service à ses contraintes et/ou possibilités de composition. Les trois scenarii sont :

- Scénario 1 : 1 fonctionnalité désirée, 2 plans de composition générés. Chaque plan de composition comporte deux services métier et une relation.

- Scénario 2 : 2 fonctionnalités désirées, 2 plans de composition générés. Chaque plan de composition comporte 4 services métier et 2 relations.

- Scénario 3 : 3 fonctionnalités désirées, 4 plans de composition générés. 2 plans de composition comportent chacun 4 services métier et 3 relations, et 2 plans de composition comportent 6 services métier et 5 relations chacun.

Les phases consommant le plus de temps d'exécution sont celles manipulant des fichiers. En effet, nous observons une augmentation remarquable dans le temps d'exécution de la phase de notation des plans de composition au fur et à mesure que la taille des résultats augmente (en termes de nombre de plans de composition générés, de relations, et de services métier impliqués). Cela est dû au fait que la notation interroge un fichier QoS XML par service métier impliqué dans le plan de composition. La variation de la taille des résultats des trois scenarii n'affecte pas significativement le temps de création du fichier RIVAL, de génération et d'exécution des scripts de configuration et de déploiement, et de découverte de services métier puisque ces phases manipulent le même nombre de fichiers quelque soit la taille des résultats.

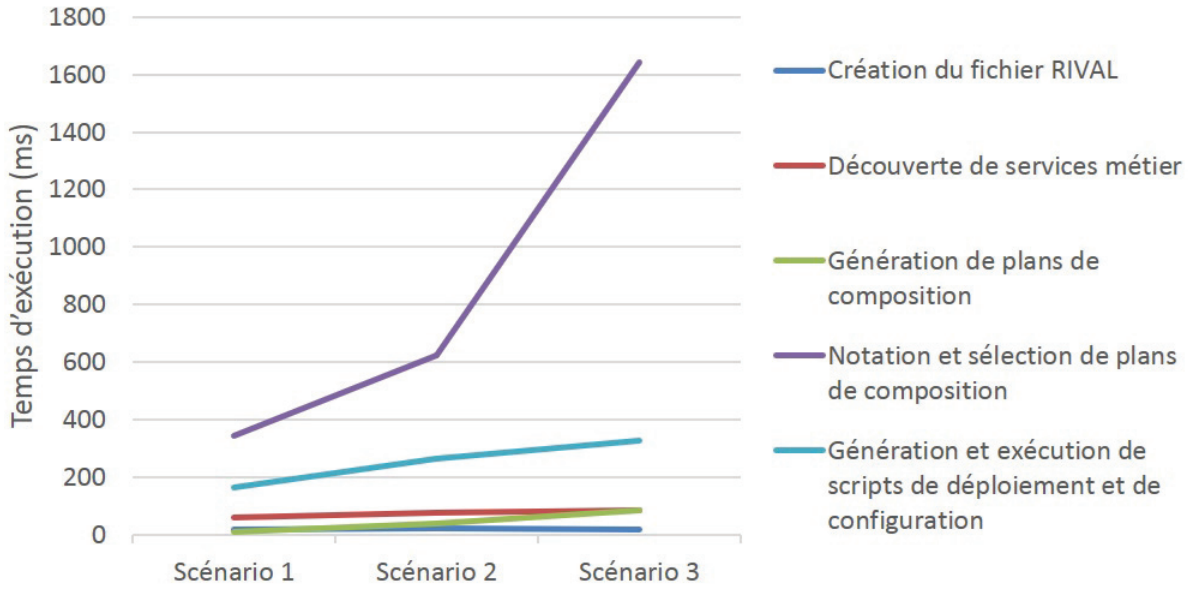


Figure 7.5: Temps d'exécution des phases de MADONA suivant trois scenarii

7.3.2 Les connaissances techniques à avoir pour le provisionnement

La Figue 7.6 illustre le nombre de services que l'utilisateur doit connaître pour le provisionnement des applications MediaWiki et WordPress. Ces dernières sont composées, chacune, de trois services métier. Pour le déploiement local, avec Juju, et avec TOSCA, l'utilisateur doit connaître tous les services impliqués dans le plan de composition. Avec Bitnami, l'utilisateur n'a pas à se soucier des contraintes de composition.

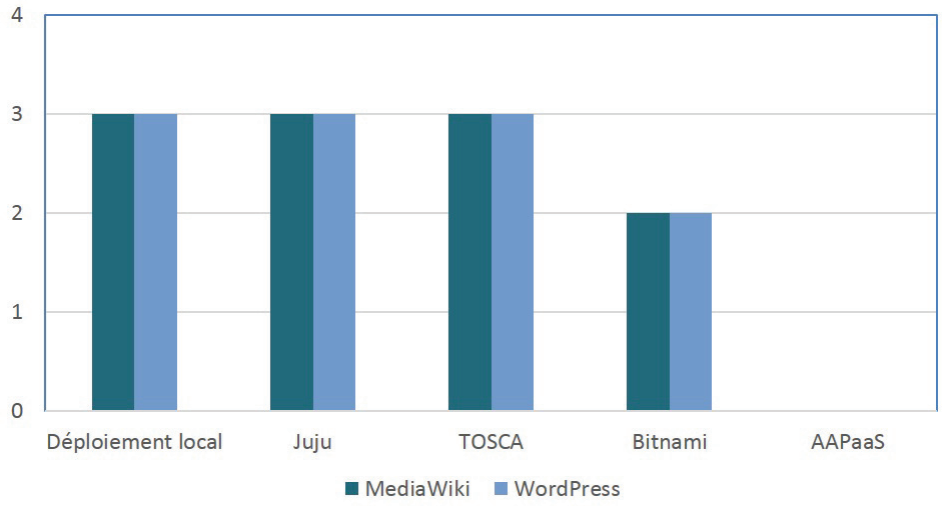


Figure 7.6: Nombre de services que l'utilisateur doit connaître pour le provisionnemnt des applications MediaWiki et WordPress

La Figure 7.7 illustre le nombre de lignes de scripts à écrire pour déployer les applications MediaWiki et WordPress. Depuis cette figure, nous observons que l'utilisateur doit déployer les services métier par le moyen de scripts pour le déploiement classique et avec Juju, tandis qu'avec Bitnami et AAPaaS les scripts sont générés automatiquement. Le déploiement sur une plateforme comme Bitnami est facile à effectuer, mais cela est fait dans une machine virtuelle préconfigurée. La composition de services n'est pas faite dynamiquement, comme c'est le cas pour MADONA. En effet, dans Bitnami les différentes compositions doivent être connues à priori et scriptées de façon statique.

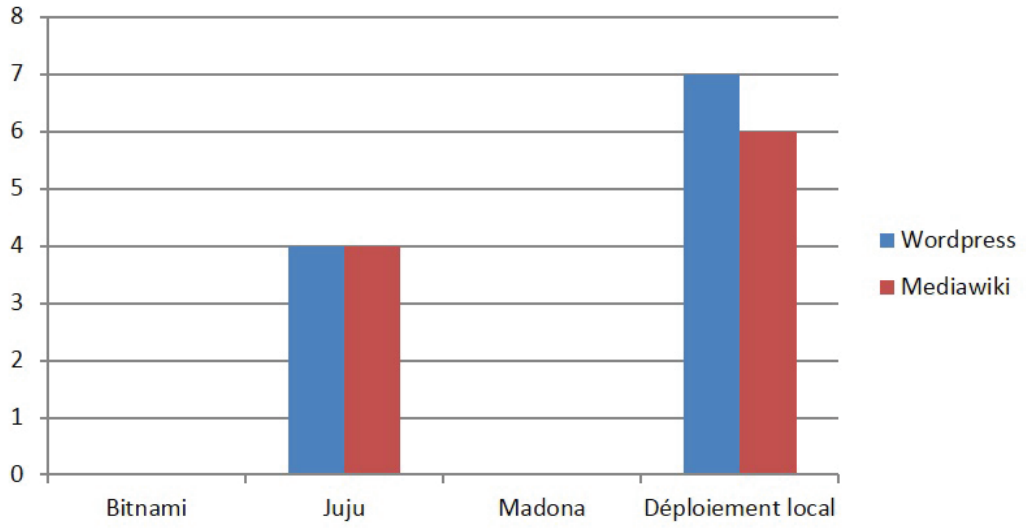


Figure 7.7: Nombre de lignes de scripts à écrire pour déployer MediaWiki et WordPress

7.3.3 Nombre de services invoqués en fonction du nombre de fonctionnalités désirées par l'utilisateur

La Figure 7.8 illustre le nombre de services ou fonctionnalités à connaître par l'utilisateur et le nombre de services invoqués par notre système, en faisant varier le nombre de fonctionnalités désirées (d'abord à 1, puis à 2) et le nombre de contraintes de composition engendrées par les services correspondant aux fonctionnalités désirées. Le nombre de services invoqués dans un plan de composition est égal à la somme du nombre de fonctionnalités désirées (primaire et secondaires) et du nombre de contraintes de composition des services associés à ces dernières. Ce que l'utilisateur doit connaître, ce sont les fonctionnalités désirées. Les contraintes de composition des services métier répondant à ces dernières lui sont épargnées.

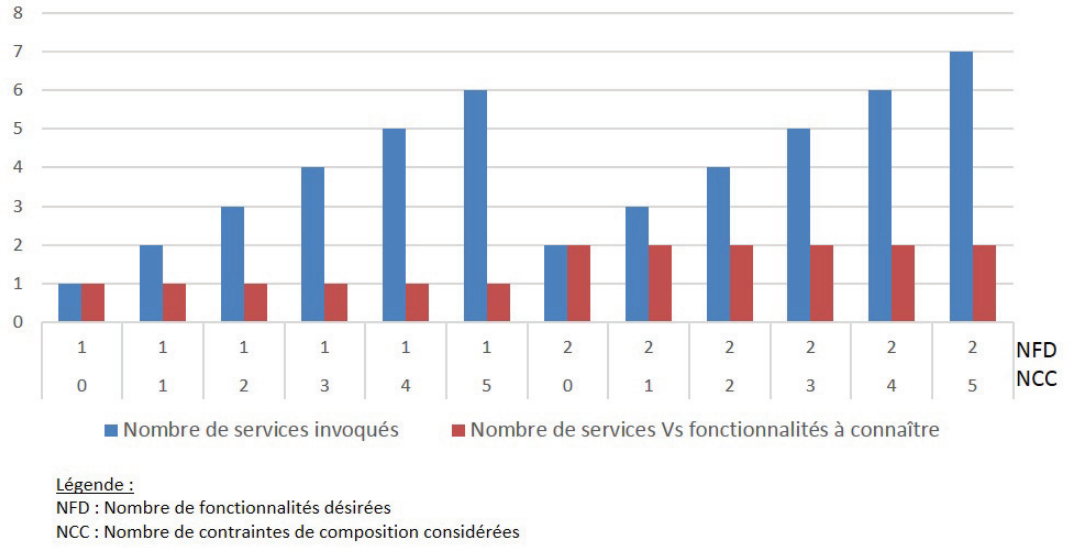


Figure 7.8: Nombre de services invoqués en fonction du nombre de fonctionnalités désirées et de leurs contraintes de composition

7.3.4 Nombre de plans de composition générés en fonction de la prise en considération des possibilités de composition de services

En définissant les compositions possibles pour un service métier, nous réduisons les plans de compositions générés. En effet, soient :

- n le nombre de fonctionnalités désirées par l'utilisateur.
- F_i la i ème fonctionnalité désirée par l'utilisateur.
- $f_1(F_i)$ le nombre de services métier répondant à une fonctionnalité désirée F_i.
- NB le nombre de plans de composition générés.

Si les compositions possibles d'un service métier ne sont pas prises en compte, alors le nombre de plans de composition générés NB est égal à (Équation 7.1) :

$$NB = \prod_{i=0}^{n-1} f_1(F_i) \tag{7.1}$$

Toutes ces combinaisons générées ne représentent pas forcément des compositions réalisables sans intervention de développement. En considérant les possibilités de composition de chaque service métier, nous ne considérons que les compositions réalisables, diminuant ainsi le nombre de plans de composition à évaluer lors du calcul de la note QoS, du calcul du coût, et du classement des plans de composition.

Nous allons à présent évaluer le nombre de plans de composition générés en prenant en compte les possibilités de composition de chaque service impliqué dans les plans. Soient :

- S_i l'ensemble des services métier répondant à la fonctionnalité désirée F_i.

- $s_{(i,k)}$ le k ième service de S_i.

- X(S_i,s) le nombre de services métier de S_i qui peuvent être composés avec s.

Le nombre de plans de composition générés en prenant en compte les possibilités de composition est égal à (Équation 7.2) :

$$NB = \sum_{k=0}^{f_1(F_0)-1} (\prod_{i=1}^{n-1} X(S_i, s_{(0,k)}) \tag{7.2}$$

La Figure 7.9 illustre le nombre de plans de composition générés en fixant $f_1(F_i)$ à 5 (i=1, i<=n), et le nombre de possibilités de composition des services à considérer X(S_i,s)=(5,4,3). L'axe des abscisses représente la variation de n, le nombre de fonctionnalités désirées.

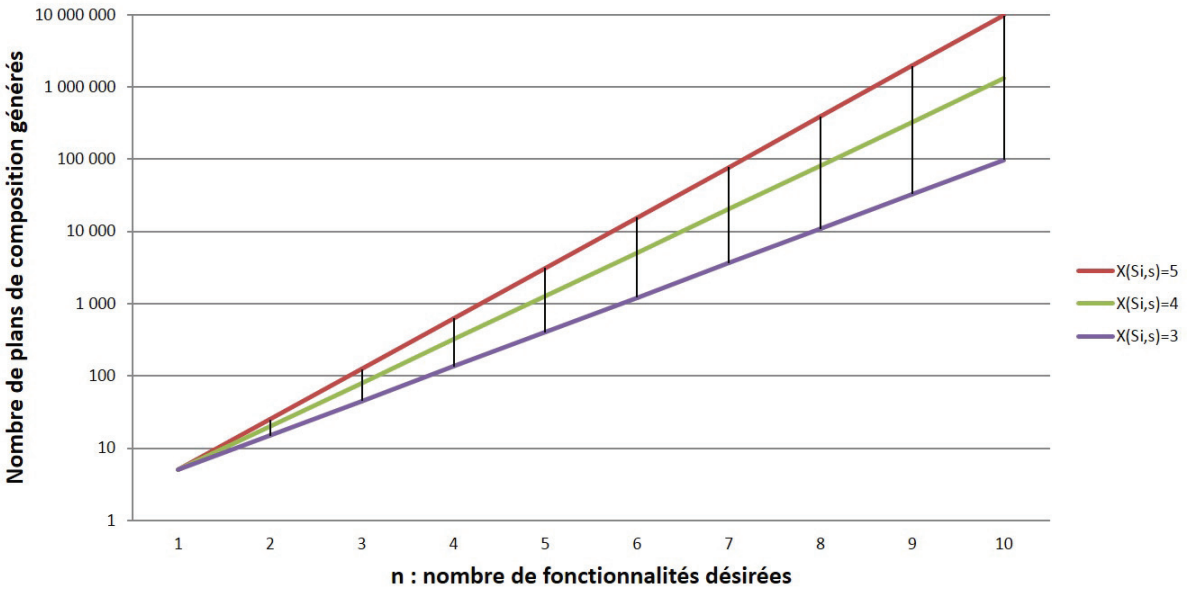


Figure 7.9: Nombre de plans de composition générés en fonction de la prise en considération des possibilités de composition

Pour une application composant 5 fonctionnalités, le passage de X(S_i,s)=5 à X(S_i,s)=3 nous fait économiser plus de 7 fois le nombre de plans de composition générés qui passe de 3 125 à 405. Et pour une application composant 10 fonctionnalités, nous générons 99 fois moins de plans de composition en passant de 9 765 625 à 98 415. De plus, la considération des possibilités de composition nous permet d'automatiser la composition et de supprimer ainsi l'intervention humaine lors de la composition.

7.4 Conclusion

Dans ce chapitre, nous avons comparé notre prototype avec le déploiement (i) local, (ii) sur Juju, et (iii) sur Bitnami de manière qualitative et quantitative. Nous n'avons pas évalué le temps de provisionnement des applications avec le déploiement local à cause de la difficulté d'évaluer un travail complètement manuel. En effet, le temps de découverte des services est très dépendant de comment il est fait, des critères de sélection pris en compte, etc. Le temps de déploiement varie en fonction de la façon dont l'utilisateur est familier avec ce type d'installation.

Nous avons montré l'intérêt de considérer les contraintes et possibilités de composition dans la description des services métier. En effet, la connaissance des contraintes et possibilités de composition permet de réduire les connaissances techniques requises de l'utilisateur pour le provisionnement d'applications en automatisant la génération de plans de composition. De plus, la connaissance des possibilités de composition des services métier permet de ne considérer que les plans de composition représentant des compositions réalisables.

Nous avons aussi montré que MADONA et son prototype "AAPaaS" réduisaient les connaissances techniques requises de l'utilisateur pour le provisionnement d'applications cloud en comparant les quatre approches - MADONA, déploiement local, Juju, Bitnami - sur la base du nombre de services à connaître Figure 7.6, et du nombre de lignes de script à écrire Figure 7.7.

Nous avons par ailleurs évalué le temps de provisionnement des applications MediaWiki et WordPress avec le prototype d'AAPaaS et l'avons comparé avec une solution industrielle afin de montrer que notre prototype provisionne les applications en un temps satisfaisant.

Chapitre 8

Conclusion et Perspectives

Sommaire

8.1 Conclusion

Le travail de cette thèse vise à combiner le cloud computing et le développement orienté service pour automatiser le provisionnement d'applications métier orientées service sur les environnements cloud.

Plusieurs types de services existent (ex : services Web, services métier, services cloud...) (cf. Chapitre 2). Dans ce travail de thèse, nous souhaitons provisionner des applications métier par la composition automatique des services les constituant. De ce fait, nous ne considérons pas les services Web comme les briques de base pour le développement d'applications, mais plutôt des services métier représentant des packages d'applications déployables sur plusieurs infrastructures comme décrit dans [TSB10]. Nous avons ajouté à ces services métier, les IaaS pour le déploiement de l'application métier provisionnée, constituant ainsi notre marketplace de services.

La plupart des langages de description de services décrivent ces derniers comme des entités isolées. Ils se concentrent sur ce que fait le service en négligeant les relations que peut avoir ce dernier avec d'autres services. Aucun des travaux étudiés ne permettait de décrire les possibilités de composition d'un service métier de manière exhaustive. Ils ne permettaient pas non plus de décrire les contraintes de composition. En effet, ces dernières étaient associées à d'autres contraintes (d'environnement et de ressources) dans [NLPvdH12].

Pour pallier ces manquements, nous avons étendu le langage Linked USDL pour décrire les relations de composition des services métier (cette contribution est décrite

dans le Chapitre 4). Nous avons aussi intégré à la description des services, les contraintes de déploiement et les paramètres configurables de chaque service métier permettant ainsi d'automatiser son déploiement et sa configuration. Nous avons choisi de travailler avec Linked USDL pour deux raisons principales. La première est que celui-ci est basé sur le Linked Data. C'est à dire qu'il utilise et lie des concepts venus de différentes ontologies, ce qui facilite son extension, et la réutilisation d'ontologies existantes. La deuxième est qu'en plus de décrire les aspects techniques comme le permettait WSDL, il permet de décrire aussi les aspects métier et opérationnels, notamment ceux liés au coût, à la sécurité, et à la qualité des services.

Notre extension de Linked USDL intervient dans la description des :

(i) **Relations de composition d'un service métier**. Les relations de composition lient un service métier à un autre. Elles peuvent représenter des *contraintes* ou des *possibilités* de composition. Les contraintes de composition représentent les services métier avec lesquels le service métier décrit doit absolument être composé pour bien fonctionner. Les possibilités de composition représentent les services avec lesquels le service décrit peut éventuellement être composé pour enrichir son fonctionnement.

(ii) **Contraintes de déploiement d'un service métier.** Elles couvrent les *contraintes d'environnement* et de *ressources*. Les contraintes d'environnement représentent les logiciels qui doivent être installés sur la même machine sur laquelle sera déployé le service métier (ex : serveur Web). Les contraintes de ressources représentent les caractéristiques (ex : RAM, disque, CPU) de la machine virtuelle sur laquelle sera déployé le service métier.

(iii) **Paramètres configurables du service métier décrit**. Cela concerne ses paramètres personnalisables (ex : login et mot de passe, nom de l'application, son logo...).

Notre seconde contribution (présentée dans le Chapitre 4) a été de définir le vocabulaire RIVAL pour la description des besoins de l'utilisateur en vue du provisionnement d'applications. Dans un objectif de minimisation des connaissances techniques requises pour ce provisionnement, nous avons choisi de permettre à l'utilisateur d'exprimer son besoin sur un formulaire Web, nul besoin de connaître une notation ou un langage pour ce faire. Ces besoins sont ensuite automatiquement traduits en fichier RIVAL pour exploitation lors de la découverte de services. RIVAL permet de décrire les besoins fonctionnels et non fonctionnels à un haut niveau d'abstraction des détails techniques de déploiement et de composition.

Les besoins fonctionnels peuvent être décrits soit par mots clés, soit par le nom des services métier que l'utilisateur souhaite déployer. Nous distinguons les fonctionnalités primaires des fonctionnalités secondaires. La fonctionnalité globale de l'application souhaitée est considérée comme étant une fonctionnalité primaire (ex : gestion de projets dans notre scénario illustratif). Toute fonctionnalité supplémentaire est considérée

comme étant une fonctionnalité secondaire (ex : gestion des plannings dans notre exemple illustratif).

Les besoins non fonctionnels intègrent les préférences de déploiement (fournisseur et localisation préférés) et de coût (coût à ne pas dépasser, monnaie). Ils décrivent aussi les contraintes de QoS de l'utilisateur. Ces dernières sont représentées par des poids affectés par l'utilisateur à chaque paramètre QoS considéré. Ce poids témoigne de la priorité qu'affecte l'utilisateur à un paramètre de QoS.

Notre troisième contribution (présentée dans le Chapitre 5) est MADONA, une méthodologie de provisionnement automatique d'applications métier orientées service sur le cloud. MADONA prend en entrée les besoins fonctionnels et non fonctionnels de l'utilisateur vis à vis de l'application métier souhaitée pour générer de manière automatique et dynamique des plans de composition. Ces derniers sont composés d'un ensemble de relations. Chaque relation représente une composition de services. Les services ne sont composés que s'ils sont composables (prise en compte automatique des possibilités de composition des services). Les contraintes de composition de chaque service d'un plan de composition sont automatiquement prises en compte dans la génération de ce dernier.

Les plans de composition générés sont ensuite notés par rapport à leur qualité de service, et le plan ayant la plus haute note QoS est déployé sur une IaaS automatiquement présélectionnée en fonction des préférences de déploiement de l'utilisateur.

Si aucun service métier de la marketplace ne répond aux besoins fonctionnels de l'utilisateur, ce dernier a la possibilité d'intégrer de nouveaux services.

Notre quatrième contribution (présentée dans le Chapitre 6) est la mise en œuvre de notre approche méthodologique sous forme d'un environnement de provisionnement automatique d'applications métier orientées service sur le cloud "Automatic Application Provisioning as a Service" (AAPaaS). AAPaaS a été implémenté comme une application Web suivant le patron Modèle, Vue, Contrôleur (MVC). Nous avons utilisé l'orchestrateur Juju pour la gestion des déploiement, configuration, et composition des services métier. AAPaaS a été évalué qualitativement et quantitativement dans le Chapitre 7 de cette thèse. Dans l'évaluation qualitative, nous avons comparé le fonctionnement de notre système lors du provisionnement d'applications, avec Juju, Bitnami, et le déploiement local. L'évaluation quantitative a permis d'évaluer les performances d'AAPaaS, notamment le temps de provisionnement d'applications et le temps d'exécution des différentes phases de MADONA. Nous avons évalué la réduction des connaissances techniques requises au provisionnement d'applications par l'évaluation du nombre de lignes de script à écrire et du nombre de services à connaître. Nous avons aussi montré l'intérêt de considérer les contraintes et possibilités de composition lors de la génération des plans de composition. En effet, la connaissance des contraintes de composition de services depuis leur description permet de réduire les connaissances techniques de l'utilisateur. D'un autre côté, la connaissance des possibilités de composition des services depuis leur description permet de ne considérer que les compositions

réalisables lors de la génération de plans de composition.

8.2 Perspectives

La réalisation de notre méthodologie de provisionnement d'applications métier orientées service sur le cloud nous a permis de dégager plusieurs perspectives de travail :

1- **Définir un middleware pour la prise en considération de plusieurs orchestrateurs.** En effet, dans ce travail, nous avons considéré un seul orchestrateur de services pour le déploiement de ces derniers. Comme perspective de ce travail, un middleware peut être mis en œuvre permettant d'utiliser AAPaaS sur d'autres orchestrateurs (Cloudify [cloc]), outils de configurations (Puppet [pup], Chef[che]), ou encore TOSCA [BBKL14]. Des règles de traduction doivent être décrites et formalisées afin de permettre la transformation automatique d'un script de déploiement indépendant de l'orchestrateur utilisé, en un script dédié à ce dernier.

2- **Intégrer une phase de négociation à MADONA.** Cette dernière interviendra sur deux axes :

- Pour l'établissement du SLA (Service Legal Agreement) afin de permettre à l'utilisateur d'avoir une garantie en termes de qualité de service sous forme de contrat le liant au fournisseur de service IaaS. Ces aspects peuvent être décrits via le module usdl-agreement de Linked USDL.
- Pour négocier les préférences de l'utilisateur si aucun plan de composition répondant aux besoins de l'utilisateur n'est généré.

3- **Prendre en compte les aspects sécuritaires dans la sélection des services métier et IaaS.** En effet, le cloud suscite beaucoup de questionnements sur l'intégrité des données et leur confidentialité puisque ces données sont stockées et gérées chez un tiers. Certains utilisateurs pourraient souhaiter avoir plus de détails sur la façon dont seront gérées leurs données. Ces aspects peuvent être décrits via les modules usdl-sec et usdl-ipr de Linked USDL.

4- **Définir ou réutiliser les systèmes de matching sémantiques existants** dans le processus de découverte de services métier afin de mieux répondre aux besoins fonctionnels des utilisateurs.

5- **Mesurer la satisfaction de l'utilisateur** par rapport à la pertinence de la sélection des services, en fonction de son besoin fonctionnel.

6- **Considérer un modèle de coût plus complet** permettant une définition détaillée des paramètres rentrant en ligne de compte lors de l'estimation de l'utilisation des ressources pour le déploiement des services métier, notamment tenir compte

du type de stockage et de la bande passante associés aux machines virtuelles déployées. Nous prévoyons aussi de considérer le coût des services métier constituant l'application générée.

7- **Elargir la notion de services métier.** En effet, nous avons considéré qu'un service métier représentait un package déployable sur plusieurs infrastructures. Nous souhaitons dans le futur intégrer à la notion de services métier ceux déjà déployés sur des infrastructures cloud. Pour ce faire, nous allons ajouter une métadonnée permettant de connaître l'état de déploiement du service (déployé, ou package déployable). Cette donnée sera utile lors de la génération du script de déploiement (haut niveau) afin de ne pas redéployer le service. Toutefois, il est nécessaire de tester la faisabilité de composer de manière automatique un service métier déjà déployé sur un cloud avec d'autres services métier.

8- **Évaluer notre prototype en augmentant considérablement le nombre de services de la marketplace.** Notre marketplace actuelle est composée d'une vingtaine de services métier. Nous souhaitons évaluer la scalabilité du prototype afin de s'assurer que ce dernier provisionne des applications en un temps qui reste satisfaisant même avec un grand nombre de services.

Références bibliographiques

[ACK12] Piyush Aghera, Sanjay Chaudhary, and Vikas Kumar. An approach to build multi-tenant saas application with monitoring and sla. In *Proceedings of the International Conference on Communication Systems and Network Technologies*, CSNT2012, pages 658–661. IEEE, 2012. 14, 47, 50

[ADNC+12] Danilo Ardagna, Elisabetta Di Nitto, Giuliano Casale, Dana Petcu, Parastoo Mohagheghi, Sébastien Mosser, Peter Matthews, Anke Gericke, Cyril Ballagny, Francesco D'Andria, et al. Modaclouds: A model-driven approach for the design and execution of applications on multiple clouds. In *Proceedings of the 4th International Workshop on Modeling in Software Engineering*, pages 50–56. IEEE Press, 2012. 42, 49

[Ald11] Kadan Aldjoumaa. Modélisation intentionnelle et annotation sémantique pour la réutilisation de services métiers. *Thèse de doctorat de l'université Paris I – Panthéon - Sorbonne*, 2011. 16

[amaa] Amazon cloud formation. https://aws.amazon.com/fr/cloudformation/. 23-02-2015. 98

[amab] Amazon ec2. http://aws.amazon.com/fr/ec2/. 17-02-2015. 15, 58, 99

[AMBT14] Yasmine M Afify, Ibrahim F Moawad, Nagwa L Badr, and MF Tolba. Cloud services discovery and selection: Survey and new semantic-based system. In *Bio-inspiring Cyber Security and Cloud Services: Trends and Innovations*, pages 449–477. Springer, 2014. 17, 27

[BA14] Hanane Becha and Daniel Amyot. Consumer-centric non-functional properties of soa-based services. In *Proceedings of the 6th International Workshop on Principles of Engineering Service-Oriented and Cloud Systems*, PESOS 2014, pages 18–27. ACM, 2014. 16, 27

[BBKL14] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. Tosca: Portable automated deployment and management of cloud applications. In *Advanced Web Services*, pages 527–549. Springer, 2014. 12, 130

[BIN10] Muneera Bano, Naveed Ikram, and Mahmood Niazi. Thesis proposal on "requirement engineering process for service oriented software development". In *Proceedings of the 11th International Conference on Product Focused Software*, PROFES2010, pages 84–87. ACM, 2010. 29

[bit] Bitnami cloud hosting. https://bitnami.com/. 27-05-2015. 114

[bpm] Business process model and notation. http://www.bpmn.org/. 24-03-2016. 31, 36, 37

[Car13] Jorge Cardoso. Modeling service relationships for service networks. In *Proceedings of the 4th International Conference on Exploring Services Science*, IESS2013, pages 114–128. Springer, 2013. 21, 27, 28, 36, 62

[CBB+13] Jorge Cardoso, Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. Cloud computing automation: integrating usdl and tosca. In *Proceedings of the 25th International Conference on Advanced Information Systems Engineering*, CAISE2013, pages 1–16. Springer, 2013. 24

[CBCG10] Verónica Castañeda, Luciana Ballejos, Ma Laura Caliusco, and Ma Rosa Galli. The use of ontologies in requirements engineering. *Global journal of researches in engineering*, 10(6), 2010. 29

[CDKB11] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition, 2011. 16

[che] Chef cloud management. https://www.chef.io/solutions/cloud-management/. 27-05-2015. 99, 130

[cloa] Cloud armor. http://cs.adelaide.edu.au/~cloudarmor/ds.html. 14-06-2016. 64

[clob] Cloud foundry. http://www.cloudfoundry.org/index.html. 17-02-2015. 14

[cloc] Cloudify. http://www.cloudify.cc/. 10/02/2016. 130

[clod] Cloudorado cloud computing comparison engine. https://www.cloudorado.com/. 10/02/2016. 64

[CO12] Yuyu Chou and Jan Oetting. Secure system development for integrated cloud applications. In *Proceedings of the 2nd Symposium on Network Cloud Computing and Applications*, NCCA2012, pages 80–87. IEEE, 2012. 46, 50

[CP15] Jorge Cardoso and Carlos Pedrinaci. Evolution and overview of linked usdl. In *Proceedings of the 6th International Conference on Exploring Services Science*, IESS2015, pages 50–64. Springer, 2015. 23

[CSM] Cloud service measurement index consortium. http://csmic.org/. 24-02-2015. 19, 27

[CVT10] Bharat Chhabra, Dinesh Verma, and Bhawna Taneja. Software engineering issues from the cloud application perspective. *International Journal of Information Technology and Knowledge Management*, 2(2):669–673, 2010. 13

[D'A06] Andrea D'Ambrogio. A model-driven wsdl extension for describing the qos of web services. In *International Conference on Web Services*, ICWS2006, pages 789–796. IEEE, 2006. 16, 27

[dbp] Dbpedia. http://wiki.dbpedia.org/. 01-06-2016. 24

[DeM79] Tom DeMarco. Structure analysis and system specification. In *Pioneers and Their Contributions to Software Engineering*, pages 255–288. Springer, 1979. 30

[Eas94] Steve Easterbrook. Resolving requirements conflicts with computer-supported negotiation. *Requirements engineering: social and technical issues*, 1:41–65, 1994. 70

[EG04] Alexander Egyed and Paul Grünbacher. Identifying requirements conflicts and cooperation: How quality attributes and automated traceability can help. *IEEE Software*, 21(6):50–58, 2004. 70

[GAD10] Radha Guha and David Al-Dabass. Impact of web 2.0 and cloud computing platform on software engineering. In *International Symposium on Electronic System Design*, ISED2010, pages 213–218. IEEE, 2010. 45, 50

[gooa] Cloud computing as a necessity. http://cloudipedia.com/2009/12/21/cloud-computing-as-a-necessity/. 01-06-2016. 2

[goob] Goodrelations. http://www.heppnetz.de/projects/goodrelations/. 27-05-2015. 24, 69

[gooc] Google app engine - google cloud platform. https://cloud.google.com/appengine/docs. 17-02-2015. 14

[gra] Grails framework. https://grails.org/. 17-02-2015. 98

[Hat] Sudhanshu Hate. Introducing software engineering-as-a-service. http://www.hpcwire.com/2012/06/19/introducing_software_engineering-as-a-service/. 10-05-2015. 42, 49

[HCS05] Dan Hong, Dickson KW Chiu, and Vincent Y Shen. Requirements elicitation for the design of context-aware applications in a ubiquitous environment. In *Proceedings of the 7th International Conference on Electronic Commerce*, ICEC2005, pages 590–596. ACM, 2005. 30

[HHT02] Jan Hendrik Hausmann, Reiko Heckel, and Gabi Taentzer. Detection of conflicting functional requirements in a use case-driven approach: a static analysis technique based on graph transformation. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE2002, pages 105–115. ACM, 2002. 70

[hp] Hp cloud. http://www.hpcloud.com/. 14-09-2015. 99

[HPW98] Peter Haumer, Klaus Pohl, and Klaus Weidenhaupt. Requirements elicitation and validation with real world scenes. *IEEE Transactions on Software Engineering*, 24(12):1036–1054, 1998. 29

[HS10] Taekgyeong Han and Kwang Mong Sim. An ontology-enhanced cloud service discovery system. In *Proceedings of the International MultiConference of Engineers and Computer Scientists*, pages 17–19, 2010. 17, 27

[ins] Insee - définition de services. http://www.insee.fr/fr/methodes/?page=definitions/services.htm. 26-08-2016. 11

[Jal05] Pankaj Jalote. *An Integrated Approach to Software Engineering*. Texts in Computer Science. Springer, 2005. 29

[jen] Apache jena. https://jena.apache.org/. 30-08-2016. 98

[juj] Ubuntu juju. https://juju.ubuntu.com/. 17-02-2015. 91, 98, 99

[KKH11] Sungjoo Kang, Sungwon Kang, and Sungjin Hur. A design of the conceptual architecture for a multitenant saas application platform. In *Proceedings of the 1st ACIS/JNU International Conference on Computers, Networks, Systems and Industrial Engineering*, CNSI2011, pages 462–467. IEEE, 2011. 41, 47, 48, 49, 50

[KS11] Jaeyong Kang and Kwang Mong Sim. Cloudle: an ontology-enhanced cloud service search engine. In *Web Information Systems Engineering–WISE 2010 Workshops*, pages 416–427. Springer, 2011. 17, 27

[KSA+14] Carlos Kamienski, Ricardo Simoes, Ernani Azevedo, Ramide Dantas, Claudio Dias, Djamel Sadok, and Sueli Fernandes. E2ecloud: Composition and execution of end-to-end services in the cloud. In *IEEE Symposium on Computers and Communication*, ISCC2014, pages 1–6. IEEE, 2014. 43, 48, 49, 50

[KZ11] Hanu Kommalapati and William H. Zack. The saas development lifecycle. http://www.infoq.com/articles/SaaS-Lifecycle, 2011. 10-05-2015. 44, 50

[LDB15] Angel Lagares Lemos, Florian Daniel, and Boualem Benatallah. Web service composition: A survey of techniques and tools. *ACM Computing Surveys*, 48(3):33, 2015. 11, 12

[lina] Linked data - connect distributed data across the web. http://www.linkeddata.org/. 17-02-2015. 4, 23

[linb] Linked usdl. http://linked-usdl.org/. 17-02-2015. 4, 11, 23, 27, 28

[linc] Linked usdl modules. http://github.com/linked-usdl/. 17-02-2015. 23, 24

[LKH12] Jihyun Lee, Sungju Kang, and Sung Jin Hur. Web-based development framework for customizing java-based business logic of saas application. In *Proceedings of the 14th International Conference on Advanced Communication Technology*, ICACT2012, pages 1310–1313. IEEE, 2012. 47, 50

[Lüf14] Egon Lüftenegger. *Service-dominant business design*. Eindhoven University of Technology, 2014. 3, 11

[LZ11] Dongxi Liu and John Zic. Cloud#: A specification language for modeling cloud. In *Proceedings of the International Conference on Cloud Computing*, CLOUD2011, pages 533–540. IEEE, 2011. 17, 27

[LZL+10] Wenyu Liu, Bin Zhang, Ying Liu, Deshuai Wang, and Yichuan Zhang. New model of saas: Saas with tenancy agency. In *Proceedings of the 2nd International Conference on Advanced Computer Control*, volume 2 of *ICACC2010*, pages 463–466. IEEE, 2010. 47, 50

[LZZ+11] Dancheng Li, Wei Zhang, Shuangshuang Zhou, Cheng Liu, and Weipeng Jin. Portal-based design for saas system presentation layer configurability. In *Proceedings of the 6th International Conference on Computer Science & Education*, ICCSE2011, pages 1327–1330. IEEE, 2011. 47, 48, 50

[mar] Synthèses d'études et référentiels de pratiques - markess. http://www.markess.com/syntheses-etudes. 29-09-2016. 2

[MG11] Peter M. Mell and Timothy Grance. The nist definition of cloud computing. Technical report, Gaithersburg, MD, United States, 2011. 13, 14, 15

[mos] Mosaic cloud: Open source api and platform for multiple clouds. http://www.mosaic-cloud.eu/. 27-09-2016. 98

[MPM+04] David Martin, Massimo Paolucci, Sheila McIlraith, Mark Burstein, Drew McDermott, Deborah McGuinness, Bijan Parsia, Terry Payne, Marta Sabou, Monika Solanki, et al. Bringing semantics to web services: The owl-s approach. In *First International Workshop on Semantic Web Services and Web Process Composition*, pages 26–42. Springer, 2004. 16, 27

[Mri07] Michaël Mrissa. Médiation sémantique orientée contexte pour la composition de services web. *Thèse de l'Université Claude Bernard Lyon 1*, 2007. 2, 16

[MWB+11] Zakaria Maamar, Leandro Krug Wives, Youakim Badr, Said Elnaffar, Khouloud Boukadi, and Noura Faci. Linkedws: A novel web services discovery model based on the metaphor of "social networks". *Simulation Modelling Practice and Theory*, 19(1):121–132, 2011. 20, 27, 28, 36, 62

[MZBL13] Gaurav Mitra, Xuan Zhou, Athman Bouguettaya, and Xumin Liu. A request oriented model for web services. In *Proceedings of the 1st Australasian Web Conference-Volume 144*, pages 13–20. Australian Computer Society, Inc., 2013. 31, 34, 36, 37

[NE00] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE2000, pages 35–46. ACM, 2000. 29

[NLPvdH12] Dinh Khoa Nguyen, Francesco Lelli, Mike P Papazoglou, and W-J van den Heuvel. Issue in automatic combination of cloud services. In *10th International Symposium on Parallel and Distributed Processing with Applications*, ISPA2012, pages 487–493. IEEE, 2012. 20, 22, 27, 28, 36, 62, 127

[oas] Oasis-advanced open standards for the information society. https://www.oasis-open.org/. 23-02-2015. 99

[ope] Openstack open source cloud computing software. https://www.openstack.org/. 23-02-2015. 99

[PJ08] Kapil Pant and Matjaz B Juric. *Business process driven SOA using BPMN and BPEL: From business process modeling to orchestration and service oriented architecture*. Packt Publishing Ltd, 2008. 31, 32

[PLL10] Zeeshan Pervez, Sungyoung Lee, and Young-Koo Lee. Multi-tenant, secure, load disseminated saas architecture. In *Proceedings of the 12th International Conference on Advanced Communication Technology*, volume 1 of *ICACT2010*, pages 214–219. IEEE, 2010. 40

[pup] Puppet labs : It automation. https://puppetlabs.com/. 27-05-2015. 99, 130

[RARV12] Stefan T Ruehl, Urs Andelfinger, Andreas Rausch, and Stephan AW Verclas. Toward realization of deployment variability for software-as-a-service applications. In *Proceedings of the 5th international conference on Cloud computing*, CLOUD2012, pages 622–629. IEEE, 2012. 47, 48, 50

[rdf] Resource description framework. https://www.w3.org/RDF/. 27-10-2015. 11, 69

[RFG09] Mohsen Rouached, Walid Fdhila, and Claude Godart. A semantical framework to engineering wsbpel processes. *Information Systems and E-Business Management*, 7(2):223–250, 2009. 31

[Rito6] Peter Rittgen. *Enterprise modeling and computing with UML*. IGI Global, 2006. 11

[saw] Semantic annotations for wsdl and xml schema. http://www.w3.org/TR/sawsdl/. 26-06-2015. 17, 27

[SBA10] Krzysztof Sledziewski, Behzad Bordbar, and Rachid Anane. A dsl-based approach to software development and deployment on cloud. In *Proceedings of the 24th International Conference on Advanced Information Networking and Applications*, AINA2010, pages 414–421. IEEE, 2010. 46, 49, 50

[SK03] Biplav Srivastava and Jana Koehler. Web service composition - current solutions and open problems. In *ICAPS 2003 Workshop on Planning for Web Services*, pages 28–35, 2003. 11

[soa] Soa modeling and bpmn. http://fr.slideshare.net/ayazhamiya/soa-modeling-bpmn. 31

[SWZ+10] Hailong Sun, Xu Wang, Chao Zhou, Zicheng Huang, and Xudong Liu. Early experience of building a cloud platform for service oriented software development. In *International Conference on Cluster Computing Workshops and Posters*, CLUSTER WORKSHOPS2010, pages 1–4. IEEE, 2010. 40, 49

[SZC+07] Wei Sun, Kuo Zhang, Shyh-Kwei Chen, Xin Zhang, and Haiqi Liang. Software as a service: An integration perspective. In *Proceedings of the International Conference on Service-Oriented Computing*, ICSOC2007, pages 558–569. Springer, 2007. 18, 27

[TBAT12] Amirreza Tahamtan, Seyed Amir Beheshti, Amin Anjomshoaa, and A Min Tjoa. A cloud repository and discovery framework based on a unified business and cloud service ontology. In *Proceedings of the 8th World Congress on Services*, SERVICES2012, pages 203–210. IEEE, 2012. 17, 27

[TEL+11] Enric Tejedor, Jorge Ejarque, Francesc Lordan, Roger Rafanell, Javier Alvarez, Daniele Lezzi, Raül Sirvent, and Rosa M Badia. A cloud-unaware programming model for easy development of composite services. In *Proceedings of the 3rd International Conference on Cloud Computing Technology and Science*, CloudCom2011, pages 375–382. IEEE, 2011. 43, 49

[THS11] Wei-Tek Tsai, Yu Huang, and Qihong Shao. Easysaas: A saas development framework. In *Proceedings of the International Conference on Service-Oriented Computing and Applications*, SOCA2011, pages 1–4. IEEE, 2011. 44, 47, 49, 50

[TNL+12] Yehia Taher, Dinh Khoa Nguyen, Francesco Lelli, Willem-Jan van den Heuvel, and Mike Papazoglou. On engineering cloud applications-state

of the art, shortcomings analysis, and approach. *Scalable Computing: Practice and Experience*, 13(3):215–232, 2012. 18, 27

[tos] Tosca language. http://docs.oasisopen.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html#Toc356403635. 23-02-2015. 98

[TSB10] Wei-Tek Tsai, Xin Sun, and Janaka Balasooriya. Service-oriented cloud computing architecture. In *Proceedings of the 7th International Conference on Information Technology: New Generations*, ITNG2010, pages 684–689. IEEE, 2010. 12, 40, 49, 127

[tur] Turtle. http://www.w3.org/TR/turtle/. 26-06-2015. 100

[udd] Universal description discovery and integration. http://www.uddi.org/pubs/uddi-v3.0.2-20041019.htm. 24-05-2016. 16

[uml] Unified modeling language. http://www.uml.org/. 29-03-2015. 29

[usd] Unified service description language. https://www.w3.org/2005/Incubator/usdl/wiki/Main_Page. 27-08-2016. 23

[Wha] What is human services? http://www.nationalhumanservices.org/what-is-human-services. 29-02-2016. 11

[why] Why apps need a new data center stack. http://techcrunch.com/2015/05/10/why-apps-need-a-new-data-center-stack/?ncid=txtlnkusaolp00000602. 10-05-2015. 12

[win] Windows azure. http://azure.microsoft.com/fr-fr/. 17-02-2015. 15, 99

[WMS04] Chris Welty, Deborah L McGuinness, and Michael K Smith. Owl web ontology language guide. *W3C recommendation, W3C (February 2004) http://www.w3.org/TR/2004/REC-owl-guide-20040210*, 2004. 16

[wsb] Web service-business process execution language (ws-bpel). http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. 16-03-2016. 31, 36, 37

[wsca] Web service-choreography description language (ws-cdl). https://www.w3.org/TR/ws-cdl-10/. 16-03-2016. 31, 36

[wscb] Web service-choreography description language (ws-cdl). http://www.ebpml.org/ws_-_cdl.htm. 20-50-2016. 31, 32

[wsd] Web service description language (wsdl). http://www.w3schools.com/xml/xml_wsdl.asp. 12-05-2016. 16, 27

[wsm] Web service modeling ontology (wsmo). http://www.wsmo.org/. 11-01-2016. 20, 27

[XLQY07] Jian Xiang, Lin Liu, Wei Qiao, and Jingwei Yang. Srem: A service requirements elicitation mechanism based on ontology. In *Proceedings of the 31st Annual International Conference on Computer Software and Applications*, COMPSAC2007, pages 196–203. IEEE, 2007. xiii, 31, 32, 33, 36

[You88] Edward Yourdon. *Modern structured analysis*. Prentice Hall, 1988. 30

[YZ15] Xiaobu Yuan and Xieshen Zhang. An ontology-based requirement modeling for interactive software customization. In *International Model-Driven Requirements Engineering Workshop*, MoDRE2015, pages 1–10. IEEE, 2015. xiii, 31, 33, 34, 36, 37

[ZAG+12] Jiehan Zhou, Kumaripaba Athukorala, Ekaterina Gilman, Jukka Riekki, and Mika Ylianttila. Cloud architecture for dynamic service composition. *International Journal of Grid and High Performance Computing*, 4(2):17–31, 2012. 41, 48, 49, 50

[ZC05] Didar Zowghi and Chad Coulin. Requirements elicitation: A survey of techniques, approaches, and tools. *Engineering and Managing Software Requirements*, pages 19–46, 2005. 29, 30

[ZRH+12] Miranda Zhang, Rajiv Ranjan, Armin Haller, Dimitrios Georgakopoulos, Michael Menzel, and Surya Nepal. An ontology based system for cloud infrastructure services discovery. In *Proceedings of the 8th International Conference on Collaborative Computing: Networking, Applications and Worksharing*, CollaborateCom2012, pages 524–530. IEEE, 2012. 23, 27, 28

[ZYL11] Xin Zhou, Li Yi, and Ying Liu. A collaborative requirement elicitation technique for saas applications. In *Proceedings of the International Conference on Service Operations, Logistics, and Informatics*, SOLI2011, pages 83–88. IEEE, 2011. 30

Publications

Revue internationale avec comité de lecture

1. MADONA - a Method for AutomateD prOvisioning of cloud-based component-oriented busiNess Applications | Hind BENFENATKI, Gavin KEMP, Catarina FERREIRA DA SILVA, Aîcha-Nabila BENHARKAT, Parisa GHODOUS, Zakaria MAAMAR | Service Oriented Computing and Applications | Accepted on August 30, 2016

Conférences internationales avec comité de lecture

1. Service-Oriented Architecture for Cloud Application Development | Hind BENFENATKI, Gavin KEMP, Catarina FERREIRA DA SILVA, Aîcha-Nabila BENHARKAT, Parisa GHODOUS | 21st ISPE International Conference on Concurrent Engineering (CE 2014), Beijing, September 8-10, 2014

2. Cloud Application Development Methodology | Hind BENFENATKI, Catarina FERREIRA DA SILVA, Aîcha Nabila BENHARKAT, Parisa GHODOUS | IEEE/WIC/ACM International Conference on Web Intelligence (WI 2014), Warsaw, August 11-14, 2014

3. Methodology for Automatic Development of Cloud-based Business Applications | Hind BENFENATKI, Catarina FERREIRA DA SILVA, Aîcha-Nabila BENHARKAT, Parisa GHODOUS | IEEE CLOUD 2014: 7th IEEE International Conference on Cloud Computing, (Cloud 2014), Alaska, June 27-July 2, 2014

4. Cloud-based Business Applications Development Methodology | Hind BENFENATKI, Catarina FERREIRA DA SILVA, Aîcha-Nabila BENHARKAT, Parisa GHODOUS | WETICE: IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, (WETICE 2014), Parma, June 23-25, 2014

5. Automatic Software Development as a Service (ASDaaS) | Hind BENFENATKI, Catarina FERREIRA DA SILVA, Aîcha-Nabila BENHARKAT, Parisa GHODOUS | CLOSER - 4th International Conference on Cloud Computing and Services Science, (CLOSER 2014), Barcelona, April 3-5, 2014

6. Cloud Automatic Software Development | Hind BENFENATKI, Hamza SAOULI, Aîcha-Nabila BENHARKAT, Okba KAZAR, Parisa GHODOUS | 20th ISPE International Conference on Concurrent Engineering, (CE2013), Melbourne, September 2-6, 2013