Une contribution à la gestion des applications SaaS mutualisées dans le cloud: approche par externalisation

Ali Ghaddar

UNIVERSITÉ DE NANTES

UFR DES SCIENCES ET TECHNIQUES

SCIENCES ET TECHNOLOGIES
DE L'INFORMATION ET DE MATHÉMATIQUES

Année 2014

N° attribué par la bibliothèque

Une contribution à la gestion des applications SaaS mutualisées dans le cloud: approche par externalisation

---

THÈSE DE DOCTORAT DE L'UNIVERSITÉ DE NANTES
Discipline : INFORMATIQUE
Spécialité : INFORMATIQUE

Présentée
et soutenue par

Ali GHADDAR

le 18 07 2013 au LINA, devant le jury ci-dessous

Président	:	Pr. Isabelle BORNE	Université de Bretagne-Sud
Rapporteurs	:	Khalil DRIRA, Directeur de recherche	Université de Toulouse
		Lionel SEINTURIER, Professeur	Université de Lille
Examinateurs	:	Jean-Pierre GUEDON, Professeur	Université de Nantes
		Ali ASSAF, Directeur technique et R&D	Société Bitasoft
		Mourad-Chabane OUSALLAH, Professeur	Université de Nantes
		Dalila TAMZALIT, Maître de conférences	Université de Nantes
		Isabelle BORNE, Professeur	Université de Bretagne-Sud

Directeur de thèse : Pr. Mourad-Chabane OUSSALAH
Co-encadrant de thèse : Dr. Dalila TAMZALIT
Laboratoire : LABORATOIRE D'INFORMATIQUE DE NANTES ATLANTIQUE.
2, rue de la Houssinière, BP 92 208 – 44 322 Nantes, CEDEX 3.

UNE CONTRIBUTION À LA GESTION DES APPLICATIONS SAAS MUTUALISÉES DANS LE CLOUD: APPROCHE PAR EXTERNALISATION

A contribution to the management of multi-tenant SaaS applications in the cloud: an outsourcing approach

Ali GHADDAR

Université de Nantes

Ali GHADDAR

Une contribution à la gestion des applications SaaS mutualisées dans le cloud: approche par externalisation

??+187 p.

Ce document a été préparé avec LaTeX2e et la classe `these-LINA` version v. 1.30 de l'association de jeunes chercheurs en informatique LOGIN, Université de Nantes. La classe `these-LINA` est disponible à l'adresse :

`http://login.lina.sciences.univ-nantes.fr/`

Impression : these.tex – 24/6/2014 – 20:47

Révision pour la classe : these-LINA.cls,v 1.30 2005/08/16 17:01:41 mancheron Exp

Résumé

Le modèle économique du Cloud Computing, plus précisément dans sa couche applicative de services SaaS, a évolué vers une nouvelle approche basée sur l'exploitation des économies d'échelle. Ceci a pu être réalisé en offrant en même temps une unique instance d'application à plusieurs clients dénommés *locataires*, suivant le principe de *mutualisation*. L'objectif principal de ce principe à un niveau applicatif est de réduire les coûts opérationnels du service proposé et de capitaliser sur l'expérience cumulée à travers son partage. Cependant, sa mise en œuvre nécessite de relever un certain nombre de défis liés à sa structure organisationnelle, au sein de laquelle chaque locataire doit avoir l'impression d'utiliser une application qui lui est pleinement dédiée. Cela implique *une gestion dynamique de la variabilité* des besoins de locataires et une *isolation stricte de leurs données*. Dans cette thèse, nous nous intéressons à ce principe de mutualisation et aux principaux défis qui en découlent avant de proposer nos contributions. Celles-ci se résument en trois axes : (i) le premier concerne la spécification d'un méta-modèle de variabilité introduisant de nouveaux concepts de modélisation pour mieux traiter la variabilité et externaliser sa gestion sous forme d'un service. Nous avons pour cela introduit la notion de *VaaS (Variability as a Service)* comme un nouveau membre de la famille des services du Cloud. (ii) Le second axe consiste à étendre la politique de gestion par externalisation, initialement adoptée pour gérer la variabilité, afin de l'appliquer au niveau des données en proposant un *système d'isolation de données sous forme d'un service*. Le principal avantage de ce système est d'isoler les données de locataires d'une manière quasi-transparente aux développeurs, sans introduire de changements majeurs sur les architectures des applications existantes. (iii) Le dernier axe concerne le regroupement des deux premières contributions ainsi que d'autres services liées à l'administration et à la sécurité des applications mutualisées dans une plateforme dédiée, vers une approche globale de gestion de ce type d'applications par externalisation.

Mots-clés : Cloud Computing, SaaS, mutualisation, multi-locataires, externalisaion, architecture orientée service, variabilité, méta-modèle, isolation de données.

Abstract

The business model of cloud computing, especially in its application layer of services SaaS, has evolved into a new approach based on the exploitation of scale economies. This could be done by offering at the same time a unique application instance to several customers called *tenants*, following the *multi-tenancy* principle. The objective of this principle at the application level is to reduce the operating costs of the proposed service and to capitalize on the cumulative experience throught its sharing. However, the implementation of such principle requires addressing a number of challenges realted to its organizational structure, in which each tenant must have the feeling of using a fully dedicated application for his usage. This implies a *dynamic management of variability* in tenants needs and a *strict isolation of their data*. In this dissertation, we focus on the principle of multi-tenancy and the main challenges arising before proposing our contributions. These can be summarized in three axes: (i) the first is the specification of a variability meta-model introducing new modeling concepts to better address the variability and outsource its management as a service. For this, we have introduced the concept of VaaS *(Variability as a Service)* as a new member of the Cloud services family. (ii) The second axis is to extend the management outsourcing policy, initially adopted to manage the variability, in order to apply it at the data level by providing a *data isolation system as a service*. The main advantage of such system is to isolate tenants data in an almost transparent way to developers, without introducing major changes to the architectures of existing applications. (iii) The last axis concerns the combination of the first two contributions and other related services to the administration and security of multi-tenant applications in a dedicated platform, towards a holistic approach of multi-tenancy management by outsourcing.

Keywords: Cloud Computing, SaaS, multi-tenancy, multi-tenant, outsourcing, service oriented architecture; variability; software product line engineering; meta model, data isolation

Remerciements

J'ai toujours pensé que le fait de dire « *rien n'est impossible* », n'était en fait qu'une expression dont l'usage ne servait qu'à nous motiver pendant les moments difficiles. Mais après l'avoir vécu, j'ai réalisé à quel point cette expression reflète la réalité. Rien n'était impossible pour moi, et ceci grâce à des personnes avec qui j'ai eu la chance de travailler, et je profite de cette occasion pour les remercier.

Je tiens tout d'abord à remercier mon directeur de thèse Monsieur Mourad-Chabane OUSALLAH, Professeur de l'Université de Nantes. Je lui suis reconnaissant pour ses encouragements, ses conseils avisés et son enthousiasme pour cette thèse.

Je voudrais également remercier Madame Dalila TAMZALIT, Maître de conférences de l'Université de Nantes pour m'avoir encadré, conseillé et suivi tout au long de ce travail malgré toutes les contraintes et les difficultés. Je la remercie également pour ses corrections précieuses de ce manuscrit.

Un grand Merci à Monsieur Abdalla BITAR, directeur de la société BITASOFT pour m'avoir accueilli au sein de sa société et de m'avoir proposé ce projet si passionnant. Je souhaite encore plus le remercier pour m'avoir toujours offert l'essentiel : sa confiance.

Je remercie également Monsieur Ali ASSAF, directeur technique et R&D de la société BITASOFT, pour m'avoir encadré, appris et transféré les compétences nécessaires pour réaliser ce travail. Je lui exprime ma profonde gratitude.

Je remercie cordialement Monsieur Khalil DRIRA, Directeur de recherche de l'Université de Toulouse et Monsieur Lionel SEINTURIER, Professeur de l'Université de Lille d'avoir accepté d'être rapporteurs de cette thèse ainsi que l'intérêt qu'ils ont manifesté vis-à-vis de mon travail.

Je souhaite adresser également mes remerciements à Monsieur Jean-Pierre GUEDON, Professeur de l'Université de Nantes, et Madame Isabelle BORNE, Professeur de l'Université de Bretagne-Sud de m'avoir fait l'honneur en examinant cette thèse.

Merci à mes amis Nadine, Diana et les « Ali » pour les agréables moments qu'on a passés ensemble. Merci également à mes colocataires Nader et Jaafar pour m'avoir supporté H24 et m'encourager pendant les moments difficiles de la rédaction.

Enfin je tiens à remercier mon premier soutien, ma famille. Je remercie mes parents pour m'avoir toujours poussé à faire ce qui me plaisait. Jamais ils n'ont exprimé le moindre doute et m'ont constamment encouragé et félicité tout au long de ce parcours, sans eux je ne serais pas ce que je suis aujourd'hui. Merci également à ma sœur Hanane et mon frère Hamzeh pour leur amour, leur soutien et à tout ce qu'ils ont pu m'apporter pour franchir les obstacles les plus difficiles.

Table des matières

Liste des figures

Partie I — Contexte et motivations

Partie II — La mutualisation de SaaS : défis abordés, état de l'art et problématiques dégagées

Partie III — Contribution et validation

Partie IV — Conclusion et travaux futurs

Introduction Générale

Ce manuscrit de thèse rapporte les travaux de recherche menés au sein de la Société BITASOFT (Business Information Technology Application Software) en collaboration avec le Laboratoire LINA (Laboratoire Informatique de Nantes Atlantique) de l'Université de Nantes. Cette thèse, en convention CIFRE, traite des problématiques relatives à la gestion des applications SaaS mutualisées dans le cloud. Ce premier chapitre introduit le contexte scientique de ces travaux et leurs enjeux. Il présente également les contributions de cette thèse face aux problématiques soulevées par ces enjeux.

1.1 Cadre de la thèse

Cette thèse s'inscrit dans le domaine du *Cloud Computing* [AFG+10, MG11, NWG+09] (ou informatique dans les nuages) qui représente la cinquième et dernière génération de l'informatique à ce jour [RJ11]. Plus précisément, cette thèse s'inscrit dans le cadre des applications SaaS mutualisées et hébergées dans le cloud, ainsi que les défis conceptuels et techniques qui l'entourent.

L'histoire de l'informatique a montré que chaque progrès, en matière de technologies et de modèles économiques, crée un changement majeur dans la manière dont les logiciels sont conçus, construits et livrés aux utilisateurs finaux. L'invention des ordinateurs personnels accompagnés par la stabilité des réseaux informatiques ont conduit à la naissance de l'architecture client/serveur. Cette architecture a révolutionné, à l'époque, le monde de l'informatique tout en remplaçant définitivement les mainframes, celles qui souffraient d'une inflexibilité et d'un coût assez élevé d'installation et de maintenance. Aujourd'hui, l'accès Internet à haut débit, les architectures orientées services et l'augmentation continuelle des coûts de gestion des applications dans les infrastructures locales des entreprises (*on-premise*), ont conduit la transition vers le Cloud Computing. Comme à chaque changement de paradigme, une nouvelle série de défis conceptuels et techniques apparaissent, et le Cloud Computing n'échappe pas à la règle.

Le principe fondateur du Cloud Computing consiste à déporter vers des centres d'hébergements maintenus par des fournisseurs tiers, le développement, la maintenance et l'évolution des services informatiques (applicatifs et/ou infrastructurels) dont la dimension et la complexité les rendent souvent difficile à administrer par les entreprises ayant un savoir faire souvent peu évolué dans ce domaine. Ces services (également appelés les services du cloud [MG11]) sont tous accessibles sur Internet et facturés d'une manière proportionnelle à leur usage réel. Ceci présente comme objectif principal de décharger les clients de la complexité technologique qui supporte l'exécution de ces services, transformant ainsi leurs dépenses d'investissement en des dépenses de fonctionnement. Les services du cloud sont accessibles à la demande et ils peuvent être uniquement utilisés lorsque l'intérêt d'une entreprise converge avec leurs solutions et leurs qualités. De même, cette utilisation peut être revue (à la hausse ou à la baisse) lorsque les intérêts évoluent, voire être arrêtée si l'entreprise utilisatrice n'y trouve plus aucun intérêt.

Les services du cloud actuellement présents sur le marché sont assez nombreux, ils sont généralement divisés en trois modèles de fournitures : (i) le premier modèle est celui le plus connu concerne la fourniture des logiciels sous forme de services (SaaS : Software as a Service). Il concerne la couche applicative du cloud et celle directement utilisée par les utilisateurs finaux. (ii) Le deuxième modèle concerne la fourniture des plateformes sous forme de services (PaaS : Platform as a Service), en tant que supports

de développement aux applications SaaS. Ces plateformes regroupent un ensemble de fonctionnalités et d'outils permettant d'accélérer le développement et de faciliter le déploiement des applications. (iii) Le troisième modèle concerne la fourniture des infrastructures sous forme des services (IaaS : Infrastructure as a Service), où les matériels physiques en termes de serveurs et de réseaux sont dématérialisés, virtualisés et rendus accessibles en tant que services à travers le web au grand public. Quant aux fournisseurs de ces services, leurs objectifs est de répondre aux besoins des clients en impliquant le minimum de ressources possibles. Une des approches utilisées pour atteindre cet objectif consiste à *mutualiser* les services dont ils disposent, afin de les mettre à disposition et les partager entre tous les clients. Dans ce contexte, des nombreux défis ont émergés dont leur nature diffère selon le type de service (SaaS, PaaS, IaaS) à mutualiser [KMK12].

Dans cette thèse, nous nous sommes particulièrement intéressés par la mutualisation de SaaS et nous abordons les défis qui en découlent. Parmi ces défis nous nous concentrons principalement sur la *gestion dynamique de la variabilité* des besoins de clients et sur *l'isolation stricte de leurs données*

1.2 Motivations

La mutualisation (ou multi-tenant en anglais) de SaaS est un principe d'architecture logicielle relativement nouveau [BZ10b] qui consiste à faire héberger des multiples clients sur une instance unique d'application et de base de données. Par une instance unique nous entendons qu'un seul code est exécuté pour répondre à tous les clients et que les ressources réservées à cette exécution sont toutes partagées, même si des options peuvent être spécifiques à telle ou telle situation. Pour faire une analogie, nous pouvons comparer ce principe à la création d'un immeuble de location où certaines ressources sont partagées entre les locataires (eau, électricité, accès à Internet, gardiennage, buanderie, piscine, etc.) et d'autres qui ne le sont pas (salle de bain et cuisine dans chaque appartement) laissons à la fois un espace de liberté à chaque locataire et profitant des bénéfices économiques du partage. Ainsi, plutôt que de développer une solution logicielle spécifique pour chaque client, les fournisseurs peuvent capitaliser l'expérience cumulée auprès de tous leurs clients en leur faisant partager la même instance d'application, et faire bénéficier à chacun d'entre eux des évolutions demandées par les autres.

Cependant, et comme à chaque fois que le monde de l'informatique et de la technologie de l'information fait apparaitre une nouvelle idée, un compromis nécessite souvent d'être effectué. Le compromis de la mutualisation de SaaS consiste à faire accepter aux fournisseurs de SaaS une augmentation de la complexité de conception et de développement de l'application, pour gagner une réduction en matière de ressources infrastructurelles et humaines nécessaires pour l'exécuter et la maintenir. D'un côté, et étant donnée qu'une seule instance d'application est en cours d'exécution, les tâches quotidiennes de maintenance et de mise à jour de l'application sont considérablement plus faciles à gérer (effectuées une seule fois pour tous les clients), et le coût de l'infrastructure supportant son exécution devient suffisamment minime [BZ10b]. De l'autre côté, la mise en œuvre de la mutualisation de SaaS nécessite de relever un certain nombre de défis liés à sa structure organisationnelle, au sein de laquelle chaque client doit avoir l'impression d'utiliser une application qui lui est pleinement dédiée. Cela implique principalement *une gestion dynamique de la variabilité* des besoins de clients et une *isolation stricte de leurs données*.

Toutefois, entre ces bénéfices de la mutualisation et la complexité qu'elle impose, un équilibre doit être trouvé et maintenu. Ceci a conduit à des alternatives de conception pour réaliser ce type d'applications. Le fait de mieux comprendre ces alternatives et d'analyser profondément leurs compromis nous a permis de mettre en évidence certaines limitations et de proposer une nouvelle approche de gestion de la mutualisation de SaaS. Au départ, nos travaux de recherches se sont concentrés sur la gestion efficace

de la variabilité dans ce type d'applications, en tant que problématique incontournable à résoudre absolument dans ce contexte [KJ11]. Durant cette phase, une compréhension croissante des concepts de la variabilité a été acquise, ce qui nous a permis de contribuer à sa modélisation pour faciliter son traitement et répondre à la nécessité de la gérer d'une manière dynamique [SCG+12].

Autres que la variabilité, le fait d'héberger de multiples clients sur une instance unique d'application et surtout de base de données, crée un nouveau défi représenté par la nécessité d'isoler les données de ces clients. Ce défi consiste à assurer qu'un client ne pourra jamais avoir le privilégie d'accéder aux données des autres locataires, sauf si ce privilège a été explicitement accordé. D'après notre analyse de l'état de l'art concernant les travaux de recherche qui traitent ce défi, nous avons remarqué leur limitation face à la proposition des solutions d'isolation concrètes. Ceci nous a motivé à aborder ce défi dans cette thèse et de mettre en œuvre une solution d'isolation de données simple à intégrer et maintenir.

En parallèle, nous avons analysé la manière à travers laquelle les applications SaaS profitent actuellement des plateformes PaaS pour être développées et déployées. Nous avons voulu évaluer les possibilités de collaboration entre ces deux modèles de manière à ce que les plateformes du cloud intègrent la gestion de la mutualisation de SaaS parmi leurs services proposés. Cette orientation de notre recherche est devenue intéressante à cause de la tendance de plusieurs fournisseurs de PaaS (largement acceptés et utilisés), de couvrir les besoins techniques de la mutualisation au sein de leurs applications clientes, et ce comme une partie intégrante des services de développement et de déploiement qu'elles proposent. Cela a considérablement motivé les fournisseurs de SaaS à utiliser ce genre de solutions, ainsi, la gestion efficace de la mutualisation de leur applications est garantie et maintenue par des experts (les fournisseurs de ces plateformes). Cette approche illustre ce que nous avons explicitement dénommé : *une gestion de la mutualisation de SaaS par externalisation*. Pourtant, notre évaluation des différentes plateformes existantes, et ce malgré l'importance de leur services proposés, montrent une certaine inefficacité de gestion, comme nous le verrons plus en détails dans la section 7.4. Dans cette thèse, une plateforme dédiée à la gestion des applications mutualisées est proposée, visant à améliorer la gestion de la mutualisation de SaaS et à changer relativement la manière de penser sur le traitement de ses défis.

1.3 Contributions

Les contributions de cette thèse s'organisent suivant les trois axes de recherches que nous avons dégagé et que nous explicitons ci-après ainsi que les résultats obtenus :

1. La **première contribution** est la spécification d'un nouveau méta-modèle de variabilité spécialement conçu pour modéliser la variabilité des applications SaaS mutualisées, indépendamment de leurs technologies de développement. Il s'inspire en grande partie des travaux existants sur la variabilité dans le domaine de l'ingénierie de lignes de produits logiciels [PBVDL05, BGH+06]. Ce méta-modèle introduit des nouveaux concepts de modélisation liés principalement aux caractéristiques émergentes du modèle SaaS mutualisé et à notre orientation de fournir la gestion de la variabilité sous forme d'un service. Nous avons pour cela introduit la notion de VaaS *(Variability as a Service)* comme un nouveau membre de la famille des services du cloud. La justification de cette contribution réside essentiellement dans notre orientation plus général de gérer la mutualisation de SaaS par externalisation, celui qui nécessite tout d'abord d'externaliser la gestion de la variabilité. Le méta-modèle de variabilité viens renforcer l'architecture de VaaS à travers ses concepts de modélisation proposés.
2. La **deuxième contribution** consiste à étendre la politique de gestion par externalisation, initialement adoptée pour gérer la variabilité, pour l'appliquer au niveau des données en proposant un

système d'isolation de données sous forme d'un service (DIaaS : Data Isolation as a Service). Le principal avantage de ce système est d'isoler les données des clients d'une manière quasi-transparente aux développeurs de SaaS, sans introduire de changements majeurs sur les architectures des applications existantes. Ce système propose aux développeurs de continuer à interagir avec la base de données en simulant un environnement de développement à locataire unique. L'intérêt de cette approche est de cacher aux développeurs l'existence de plusieurs locataires dans le but d'améliorer leur productivité.

3. La **troisième contribution** est la proposition d'une plateforme dédiée qui regroupe les deux premières contributions ainsi que d'autres fonctionnalités liées à l'administration et à la sécurité des applications SaaS mutualisées (ex : gestion, facturation, surveillance, contrôle d'accès, cryptage, etc.). Pour bien situer cette plateforme par rapport à l'existant, nous avons analysé les caractéristiques d'un échantillon de plateformes (fournissant des services de gestion similaires) parmi celles assez nombreuses présentées actuellement sur le marché. L'objectif de ce travail est de clarifier la différence entre ces plateforme et la notre, pour pouvoir montrer ses avantages des points de vue fonctionnel et économique.

1.4 Confidentialité

Ce manuscrit rapporte les résultats obtenus au cours de cette thèse. Le principe de confidentialité nous interdit toutefois de donner les détails d'implémentation des contributions apportées. La validation de celles-ci se limite au niveau conceptuel nécessaire pour la compréhension de leur intérêt au bon fonctionnement de notre solution de gestion des applications SaaS mutualisées. Cette solution présente, par rapport aux solutions existantes, plus d'efficacité de gestion et élimine les coûts de développement et de maintenance de la mutualisation en externalisant sa gestion à un fournisseur plus expérimenté, dans le même esprit des services du cloud. La structure de ce document est donnée dans la section suivante.

1.5 Organisation du document

Compte tenu de la manière selon laquelle nos travaux de recherches ont évolué, l'organisation de ce manuscrit de thèse est présentée de la manière suivante :

– **Partie I : Contexte et motivations**
Cette partie est composée des chapitres 2 et 3. Le chapitre 2 présente globalement le paradigme du Cloud Computing et ses aspects différents, ainsi que les principes de l'architecture orientée services et le fonctionnement global du paradigme service. Les points de convergence et de divergences entre ce deux paradigmes sont ensuite détaillés pour mieux comprendre leur intérêt et leur rôle dans les contributions apportées. Le chapitre 3 met ensuite l'accent sur une des couches du cloud en particulier : la couche applicative concernant la fourniture des logiciels sous forme d'un service (SaaS). Nous présentons les caractéristiques et les éléments clés de ce modèle avant de rentrer dans le sujet principale de cette thèse : la mutualisation de SaaS. L'importance de ce principe ainsi qui les défis conceptuels et techniques qui l'entourent font le sujet d'une étude détaillée.

– **Partie II : La mutualisation de SaaS : défis abordés, état de l'art et problématiques dégagées**
Cette partie est composée des chapitres 4 et 5. Le chapitre 4 aborde le défi concernant la gestion de la variabilité dans les applications SaaS mutualisées. Nous présentons les approches de modélisa-

tion et de résolution de la variabilité existantes et nous analysons la possibilité de leurs réutilisation dans notre contexte d'application. Les problèmes dégagés à partir de l'analyse de ces approches sont ensuite expliqués et documentés avant de présenter notre contribution pour relever ce défi. Le chapitre 5 aborde le défi d'isolation des données de clients et analyse les différents alternatives existants de conception d'une base de données mutualisée. Il montre ensuite des solutions industrielles qui gèrent l'isolation de données d'une manière relativement efficace. Les problèmes de ces solutions sont aussi dégagés et expliqués pour motiver notre contribution au niveau de l'isolation des données de clients dans une applications SaaS mutualisées.

– **Partie III : Contributions**
  Cette partie est composée des chapitres 6 et 7. Le chapitre 6 présente en détails notre contribution à la gestion de la variabilité dans les applications SaaS mutualisées. Celle-ci propose une externalisation de cette gestion à un fournisseur tiers à travers notre système *VaaS (Variability as a Service)*. Le processus d'externalisation de gestion de la variabilité, l'architecture du système VaaS et le méta-modèle de variabilité nécessaire pour réaliser cette externalisation font sujet d'une explication détaillée. Le chapitre 7 présente la continuité de nos travaux de recherches concernant la gestion de la mutualisation de SaaS, pour appliquer la politique de gestion par externalisation aux niveau de données. Ainsi, ce chapitre détaille en premier temps notre système d'isolation de données sous forme d'un service *(DIaaS : Data Isolation as a Service)* tout en présentant le processus d'externalisation de la gestion de cette isolation et l'architecture du système DIaaS. Ensuite, ce chapitre présente notre plateforme dédiée à la gestion des applications SaaS mutualisées qui regroupe les système VaaS et DIaaS ainsi que d'autres services liées à l'administration et à la sécurité des applications mutualisées, vers une approche globale de gestion de ce type d'applications par externalisation.

– **Partie VI : Conclusion et travaux futurs**
  Cette partie est composée des chapitres 8 et 9. Le chapitre 8 présente la conclusion générale de cette thèse où nous faisons le bilan de ce travail en soulignant nos apports principaux et en mettant en valeur leurs contributions au niveau de gestion des applications SaaS mutualisées. Le chapitre 9, fait sujet d'un retour critique sur la recherche effectuée durant cette thèse et nous suggérons un ensemble de pistes d'améliorations et d'extensions possibles.

PARTIE I

Contexte et motivations

Le Cloud Computing et l'Architecture Orientée Services

2.1 Introduction

Les entreprises actuelles sont confrontées à beaucoup de défis concernant leur flexibilité opérationnelle. La dynamicité du marché et la compétitivité exigent des modifications continuelles dans leurs processus métiers ainsi les services qu'elles apportent à leurs propres clients. Pourtant, leurs systèmes d'information qui doivent supporter la flexibilité des processus métiers et donc la flexibilité de l'entreprise, sont souvent rigides et peu flexibles. Le moindre changement est assez coûteux et même risqué en terme de ses implications [Sul11]. En raison de ces coûts élevés, les entreprises tendent de plus en plus à abandonner l'idée de gérer eux-mêmes la maintenance et l'évolution de (ou d'une partie de) leurs systèmes d'information. Elle tendent plutôt à les externaliser à des entreprises spécialisées, aussi appelées *fournisseurs de services informatiques*.

Le principal avantage de cette stratégie est la réduction maximale des coûts et des risques que impliquent le développement , la maintenance et l'évolution des systèmes d'informations dans les locaux de ces entreprises [HH00]. En effet, la majorité estiment prendre moins de risques en utilisant des services validés et maintenus par des fournisseurs plus expérimentés et trouvent cette option plus rentable.

Selon [KW93], l'externalisation « *est le processus de transfert de (ou d'une partie de) fonctions du système d'information d'une entreprise à un fournisseur externe* ». Le premier modèle de fourniture qui a émergé pour réaliser la notion d'externalisation est celui d'*ASP (Application Service Provider)* [Tao01, CS$^+$01]. Suivant ce modèle, une entreprise externalise toute la pile technologique (technology stack) en termes des couches matérielles et logicielles nécessaire pour mettre en place une application allant des serveurs physiques, passant par les intergiciels (middlewares) et le système d'exploitation jusqu'à l'application elle-même. L'ASP exécute une telle pile complète séparément pour chaque client et assure également sa maintenance et son évolution.

En parallèle avec les ASP, les grilles de calculs [Fos00, Fos03, BLDGS04] ont aussi émergées comme des réseaux inter-établissements fournissant une puissance de calcul et une capacité de stockage à la demande. Elles permettent de définir des organisations virtuelles qui partagent de ressources informatiques à réserver pour un temps limité souvent pour exécuter une expérience particulière. Des exemples des projets implémentant cette technologie sont le projet européen *Data Grid* [SRG$^+$00], et le projet chinois *ChinaGrid* [Jin05].

Un concept similaire a été aussi développé par la communauté d'informatique de l'entreprise, connu sous le nom de l'informatique utilitaire (utility computing) [Rap04, Rap04]. La vision de ce concept est celle d'un fournisseur qui fournit l'accès à des ressources informatiques à la demande, de la même manière qu'on fournit l'eau et l'électricité. L'informatique utilitaire propose aux clients d'utiliser les ressources qu'ils ont en besoin le moment où ils ont en besoin, et dans la quantité et la qualité qu'ils souhaitent. L'un des premiers offres et celui le plus populaire qui applique l'idée de l'informatique utilitaire

est *Elastic Cloud computing* (EC2) [Vog08] de l'entreprise Amazon. EC2 permet aux clients de créer une pile technologique d'application arbitraire sous la forme *d'images de machine* d'EC2, et de démarrer et arrêter des instances de ces images à la demande. Les clients ne paient que pour la puissance de calcul et le stockage qu'ils utilisent réellement.

Dans cette même direction, l'IaaS (Infrastructure as as Service) et le PaaS (Platform as as Service) comme deux nouveaux types de services informatiques ont été inventés pour distinguer à quel degré la pile technologique d'une application est externalisée. Dans le modèle IaaS, uniquement les couches inférieures telles que le matériel physique, le stockage ou les serveurs y compris le système d'exploitation sont fournis, tandis que le modèle PaaS concentre sur la fourniture des outils de développement et de déploiement. En parallèle avec l'apparition de cette communauté qui vise à fournir des ressources informatiques pour développer et exécuter des applications, le modèle SaaS [TBB03, Ma07, BHL08] a commencé d'être largement accepté. Dans ce modèle, un fournisseur propose une pile technologique complète d'application à utiliser par les clients à travers le Web. Typiquement, les clients peuvent adapter certains aspects de l'application selon leurs besoins. La principale différence avec le modèle ASP qui fournit également une externalisation complète de la pile technologique est que le fournisseur de SaaS vise à tirer pleinement parti des économies d'échelle par l'hébergement de plusieurs clients sur la même instance de l'application. On parle alors d'une *application SaaS mutualisée*. Le principal problème qui découle de l'hébergement de multiples clients (également appelé locataires dans ce contexte) sur la même instance d'application est le problème de l'isolation de données de ces locataires [GSH$^+$07] et la gestion dynamique de la variabilité de leurs besoins [MMLP09].

Récemment, tous ces efforts visant à fournir des applications, plateformes et infrastructures informatiques à la demande, passant facilement à l'échelle et facturées d'une manière proportionnelle à leur usage réel ont étés résumées sous le terme Cloud Computing. Le cloud a été nommé d'après la forme dans laquelle Internet souvent apparaît dans les diagrammes d'architecture, comme un réseau de communication dont ces détails ne sont pas connus ni contrôlés. Le cloud intègre des différents types de services (SaaS, PaaS et IaaS) et se caractérise par une facilité d'utilisation, par opposition aux grilles de calcul où l'adoption à l'extérieur de la communauté scientifique était souvent entravée par des modèles et des interfaces très complexes [JMF09]. Néanmoins, le cloud s'appui sur les résultats de recherches effectuées sur les grills de calcul, ainsi que sur d'autres technologies telles que la virtualisation [Gil09], l'informatique utilitaire [RW04], et plus récemment, sur l'architecture orientée services [Pap03, PVDH07]. Du point de vue de fournisseurs du cloud, la particularité de l'architecture orientée services par rapport aux autres technologies est qu'elle fournit un ensemble des concepts et des principes architecturaux qui incluent des méthodes de développement et surtout des techniques qui facilitent la construction des applications SaaS [GSH$^+$07, LZV08, MMLP09].

Dans ce chapitre, nous allons fournir une explication détaillée des paradigmes du Cloud Computing et de l'architecture orientée services, différents mais interconnectés, nécessaire à la compréhension de notre travail de recherche et les contributions apportées dans cette thèse. Ce chapitre est organisé de la manière suivante : la section 2.2 fournie un aperçu général du Cloud Computing en termes de ses caractéristiques essentielles, facilitateurs principaux, modèles de fournitures et modes de déploiement. La section 2.3 présente les concepts et les principes de l'architecture orientée services, que nous considérons comme un modèle d'architecture bien adapté pour la construction des applications SaaS et essentiel pour la validation et la mise en œuvre de nos contributions. Dans la section 2.4, nous nous concentrons surs les points de convergence et divergence entre ces deux paradigmes. Enfin, la section 2.5 conclut ce chapitre.

2.2 Cloud Computing

Selon les auteurs dans [Feu10], le cloud est en train de redéfinir les bases sur lesquelles l'industrie informatique a fonctionné pendant des décennies. Il représente actuellement les voies d'avenir dans le monde de l'informatique et de technologie de l'information et s'appui en grande partie sur les résultats de recherches effectués sur les technologies de virtualisation [Gil09], l'informatique distribuée et utilitaire [RW04], les grilles de calculs [BFH03] et plus récemment, sur le web et l'architecture orientée service [Pap03, PVDH07]. Différentes définitions du Cloud Computing existent actuellement [Gee09, VRMCL08, TOM08, BYV+09]. Celles-ci diffèrent et se déclinent selon des divers aspects, principalement stratégiques. Toutefois, la définition de NIST (National Institute of Standards and Technology) [MG11] fournit un ensemble de principes qui, à notre sens, fournit un consensus général. Selon [MG11] le cloud est « *un modèle de développement permettant l'accès facile et à la demande, via le web, à un pool mutualisée de ressources informatiques configurables (réseaux, serveurs, stockage, applications et services), qui peuvent être rapidement mises à disposition des clients ou libérées avec un effort minimum d'administration de la part de l'entreprise ou du prestataire de service fournissant les dites ressources* ». La figure 2.1 montre un aperçu général *des aspects du cloud* [DWC10, KHSS10], notamment ses caractéristiques essentielles, facilitateurs principaux, modèles de fournitures et modes de déploiement. Dans la suite, nous allons expliquer chacun de ces aspects afin de fournir une description approfondie et plus complète de ce paradigme.

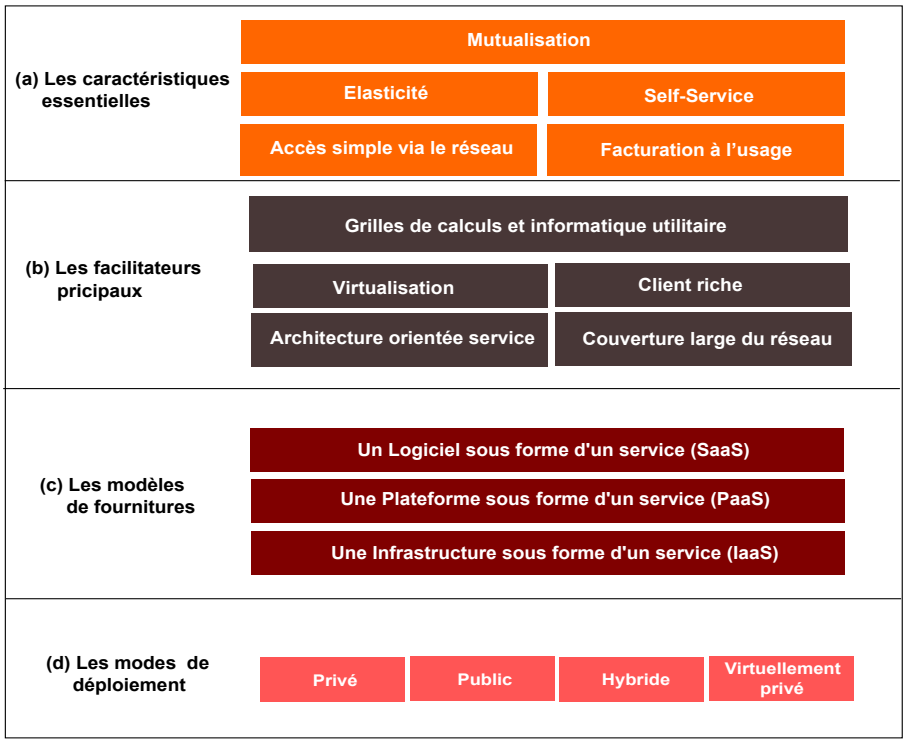


Figure 2.1 – Aperçu général du Cloud

2.2.1 Les caractéristiques essentielles

Le NIST [MG11] met en avant plusieurs caractéristiques essentielles des services du cloud notamment leur mutualisation et élasticité, la simplicité de leur accès via le réseau, leur accès unilatérale (Self-Service) et leur facturation selon l'usage réel (voir figure 2.1 (a)). Nous explicitons chacune de ses caractéristiques ci-après.

La mutualisation : les ressources du cloud sont mises en commun et mutualisées afin de servir de multiples clients (utilisateurs finaux, organisations, éditeurs de logiciels, etc.). Cette mutualisation peut intervenir à des niveaux multiples qu'il s'agisse des services infrastructurels ou applicatifs. Dans le dernier cas, qui s'applique généralement au cloud applicatif représenté par le modèle SaaS, on parle souvent d'une *architecture mutualisée* [CC06] (multi-tenant architecture en anglais). Grâce à cette mise en commun des ressources, ces dernières sont réallouées de façon dynamique en fonction de la demande et les *accords du niveau de service* (Service Level Agreement SLA) [PLL10]. Chaque client est ainsi assuré d'atteindre ses objectifs de performances définis dans le cadre de son contrat. Le principe de mutualisation de SaaS est expliqué en détails dans la section 3.3.

L'élasticité : dans le cloud, de nouvelles ressources peuvent être rapidement mises à disposition des clients pour supporter l'augmentation de leur charge de travail. De même, ces ressources peuvent être rapidement libérées lorsqu'elles ne sont plus nécessaires. Cette élasticité crée pour le client l'illusion d'une capacité infinie qui peut être mise en service à tout moment. Elle permet par exemple de faire face aux pics d'activités et d'envisager de nouvelles applications de calcul intensif qui nécessitent une capacité de calcul importante uniquement pour des périodes spécifiques de l'année (analyse de certains phénomènes naturels ou scientifiques).

L'accès simple via le réseau : les services du cloud sont accessibles au travers du réseau, qu'il s'agisse du réseau de l'entreprise pour un cloud privé ou d'Internet pour un cloud public (voir section 2.2.4 pour les différents types du cloud). Cet accès s'effectue au moyen de mécanismes et de protocoles standards qui permettent l'utilisation de services depuis de multiples types de terminaux (mobile, tablette, ordinateur portable, etc.).

Self-Service : dans le cloud, un client peut unilatéralement avoir accès de manière automatisée à des ressources informatiques (serveur, stockage, réseau, application, etc.), et en disposer tant qu'il en a besoin sans avoir à passer par de longues et complexes étapes de configuration ou d'intervention manuelle de la part du fournisseur. Cette capacité permet aux clients de répondre le plus vite possible aux besoins métiers et de réagir efficacement face à leurs changements. Dans les modèles de fourniture SaaS et PaaS, cette caractéristique du cloud est poussée à l'extrême où l'entreprise cliente s'affranchit d'un grand nombre de contraintes pour la mise en place de ses applications.

La facturation selon l'usage : les fournisseurs de services du cloud contrôlent et optimisent automatiquement l'utilisation de leurs ressources en proposant les moyens nécessaires de mesure de leur utilisation en temps réel, et ce d'une manière appropriée à chaque type de service. Cette utilisation est surveillée et rapportée périodiquement à la fois pour le fournisseur et le client du service afin d'assurer la transparence totale.

2.2.2 Les facilitateurs principaux

Les caractéristiques observées du Cloud ainsi que le succès de ce paradigme sont les résultats de plusieurs facilitateurs [MG11, FP10, WVLY+10] (voir figure 2.1 (b)) qui rendent ses caractéristiques faciles à assurer.

La virtualisation : l'élasticité de services du cloud est facilitée grâce à la virtualisation [AFG+10]. En effet, cette dernière permet le partitionnement des ressources d'un matériel physique en plusieurs ressources logiques organisées dans des machines virtuelles isolées, chacune exécutant son propre système d'exploitation et ses propres applications [SMLF09]. L'objectif est de maximiser l'utilisation des ressources physiques en exécutant autant de machines virtuelles que possible et en désactivant automatiquement celles inutiles pour libérer les ressources qu'elles occupent. De plus, la virtualisation permet la simplification des actions de maintenance du matériel et de rétablissement après panne du fait que les machines virtuelles peuvent être déplacées d'un matériel à un autre (dans la même infrastructure) et être remises en service rapidement sur le matériel non corrompu.

À ce titre, la virtualisation est largement utilisée pour assurer un passage à l'échelle automatique de services applicatifs du cloud. Ce passage peut être soit *vertical*, par la redimensionnement d'une machine virtuelle unique en lui ajoutant plus de ressources (mémoire, bande passante , CPU, disques, etc.) parmi ceux disponibles, et soit *horizontal* par l'exécution de plusieurs machines virtuelles en parallèle tout en équilibrant la charge de travail entre elles (load-balancing). En utilisant la virtualisation, les deux types de passage à l'échelle peuvent être mis en œuvre facilement. Pour passer à l'échelle horizontalement, des instances identiques d'une même machine virtuelle initiale peuvent être instantanément lancées. Cependant, en cas de passage à l'échelle vertical, la machine virtuelle est arrêtée pour être relancée à nouveau avec de nouveaux paramètres définissants les propriétés de performance désirées. Les outils VMware [SVL01], KVN [Hab08] et Xen [BDF+03] sont actuellement le plus utilisés pour implémenter la virtualisation.

L'architecture orientée service : elle est considérée comme la dernière représentante des concepts de modélisation de systèmes distribués, conduisant un changement important dans les modèles de construction des services applicatifs du cloud[FP10]. L'utilisation massive de cette architecture a formée la base générale des applications fortement distribuées, dont leurs blocs de construction sont encapsulées en tant que services autonomes accessibles sur un réseau à travers des interfaces et des protocoles de communications hautement standardisés. De nouvelles fonctionnalités plus complexes peuvent être ainsi mises en œuvre en combinant les services existants à l'aide des interactions basées sur l'échange de messages [PVDH07]. Plusieurs problématiques telles que l'interopérabilité, l'intégration et la réutilisation ont été résolues en suivant les démarches conceptuelles fournies par ce modèle d'architecture, et seules les applications fondées et conçues sur ces principes pourraient bénéficier du cloud à pleine mesure [Rai09].

Les grilles de calculs et l'informatique utilitaire : ces deux technologies [Gil09, RW04] disposent de la relation la plus forte avec le cloud. Cette relation est plus qu'une simple intersection entres leurs caractéristiques, en effet, le cloud a émergé à partir des résultats de recherches effectuées sur ces technologies [FZRL08]. En revanche, le cloud adopte un modèle économique différent visant à démocratiser l'accès aux ressources informatiques et fournir des services plus abstraits utilisables par des non-spécialistes, plutôt que pour des raisons purement scientifiques tel qu'est le cas des grilles. En ce qui concerne l'informatique utilitaire, elle définie les bases du modèle économique dans lequel les ressources informatiques sont représentées comme des services utilisables à la demande, similaire à la manière dont nous utilisons l'eau et l'électricité.

Le client riche : ce concept conduit à une évolution technologique qui transforme le Web d'un environnement uniquement dédié à la publication des informations à grande échelle vers un environnement capable de fournir des applications métiers complexes avec une possibilité d'interaction et de collaboration entre les utilisateurs [BCFC06]. Un client riche se situe à la croisée des chemins entre l'architecture client/serveur (ou client lourd) et le Web (ou client léger). Les RIA (Rich Internet

Application) représentent la catégorie des applications Web qui supportent ce concept de client rich. Grâce à des extensions technologiques du très frustre HTML, les RIA offrent un supplément d'ergonomie aux pages Web. Cela passe à travers l'utilisation des nombreuses nouvelles technologies telles que *JavaScript, Ajax, Adobe Flash, Microsoft Silverlight*, etc. [Law08b] qui permettent le développement des interfaces graphiques sophistiquées et compatibles avec la majorité de navigateurs.

La couverture large du réseau : la couverture globale des connexions internet à haut débit est crucial pour offrir des produits de masse et des interactions lisses avec les services du cloud [RJKG11]. Cela est vrai non seulement pour une bonne utilisation des services, mais aussi pour les interactions entre les différents services du cloud et entre les centres de données. Ces dernières années, il y avait eu une augmentation importante dans la couverture globale des réseaux Internet partout dans le monde et ainsi que dans leur capacité et performance.

2.2.3 Les modèles de fournitures

Les services du cloud sont très nombreux, ils sont généralement divisés en trois modèles de fournitures [AFG+10, MG11, SK11] dont chacun représente une couche d'abstraction qui s'ajoute à la pile technologique nécessaire pour mettre en place une application (confère figure 2.2). Ces couches de *l'architecture du cloud* [ZCB10] ne représentent pas juste une encapsulation des ressources accessibles à la demande, mais elles définissent également un nouveau modèle de développement d'application. Chaque couche établie un rapport de contrôle différent sur la pile technologique entre le client et le fournisseur. Dans la suite, nous allons détailler ces modèles de fourniture et donner des exemples concrets des services industriels pour chacun d'entre eux.

Infrastructure sous forme d'un service (Infrastructure as a Service IaaS) : l'IaaS est la couche la plus basse de l'architecture du cloud. Elle consiste à offrir aux entreprises clientes une externalisation de leur infrastructure informatique (serveurs, stockage, réseau) uniquement. L'entreprise a donc à sa charge le développement de ses applications, leur maintien, mais aussi l'installation des systèmes d'exploitation sur les serveurs et la gestion des bases de données.

Principe : les ressources allouées par les services IaaS sont encapsulées dans des machines virtuelles ou des instances de machines abstraites basées sur un matériel physique partagé. La structure, l'emplacement et l'utilisation de matériels physiques sont ainsi transparents aux clients et organisées en fonction des exigences du fournisseur. Ceci présente deux avantages : tout d'abord, le passage à l'échelle peut être obtenue automatiquement en fonction de la demande (redimensionnement de la machine virtuelle). D'autre part, les cycles de maintenance de l'infrastructure physique y compris la gestion de systèmes d'alimentation et de refroidissement sont effectuées par le fournisseur sans affecter la disponibilité et la performance du service. L'unité de consommation de base dans ce modèle est la machine virtuelle. Elle est généralement proposée en plusieurs formats du style petit, moyen et grand (selon le format, la machine virtuelle a plus ou moins de CPU, de mémoire et de stockage). La consommation peut être aussi facturée en fonction du temps d'utilisation de la machine, du nombre d'entrées/sorties effectué et du volume de données stockées.

Exemples : voici deux exemples de fournisseurs IaaS :

– *Amazon Elastic Compute Cloud (EC2)* [EC209] : EC2 est un service IaaS offert par l'entreprise Amazon. Il est actuellement l'exemple le plus connus pour la fourniture d'une infrastructure sur le Web. EC2 permet aux entreprises clientes de créer des machine virtuelles appelées « images de machine d'Amazon (AMI) », qui sont des images de serveur contenant un système d'exploitation et des composants logiciels nécessaires pour déployer des applications. Amazon facture

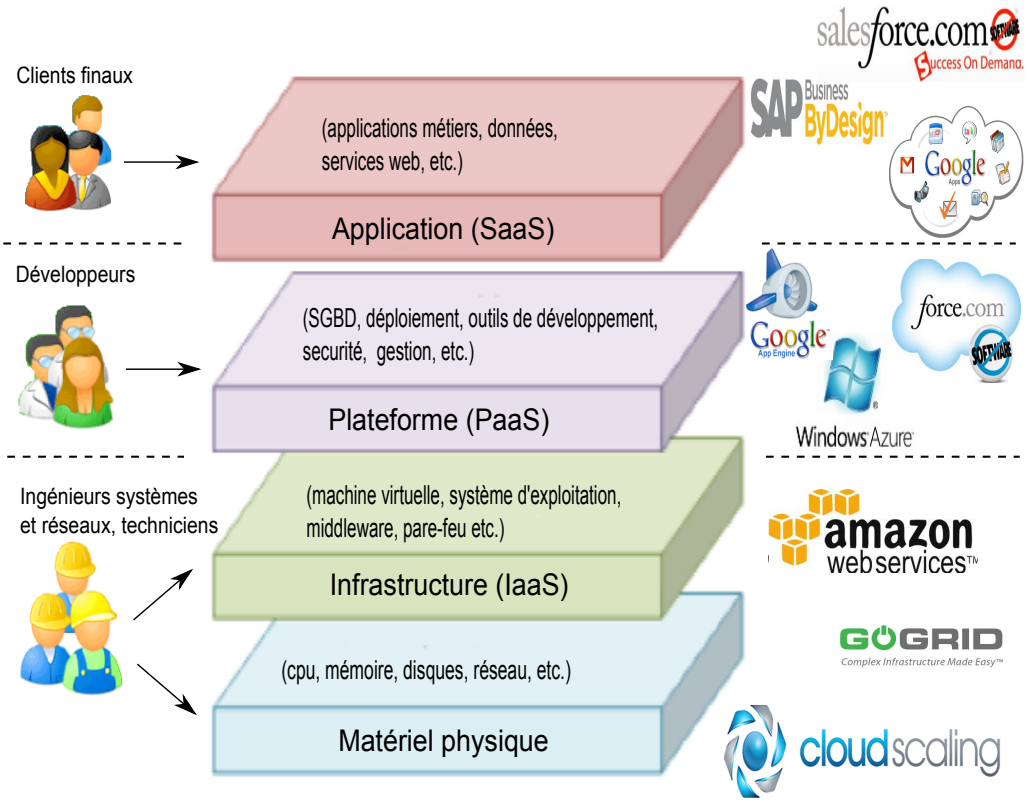


Figure 2.2 – Les modèles de fournitures du cloud

son service en fonction des heures de fonctionnement de chaque machine louée, ainsi que le volume de trafic transféré. La puissance de calcul fournie par cette infrastructure peut être adaptée dynamiquement aux besoins actuels en ajoutant et en supprimant des AMIs (passage à l'échelle horizontal). Les clients d'EC2 dépendent fortement de cette infrastructure car les AMIs créées ne peuvent pas être facilement installées ailleurs en raison du format des machines virtuelles adopté qui est propriétaire à Amazon.

– *GoGrid* [GoG09] : cette entreprise est réputée comme étant le fournisseur de l'infrastructure la plus puissante (en termes de capacité de calcul et de stockage) au niveau mondial. Il permet une approvisionnement automatisée de l'infrastructure via la fourniture d'un API de création des machines virtuelles d'une manière programmée. Aussi, les machines virtuelles louées peuvent être démarrées et arrêtées dynamiquement pour réagir efficacement contre la variation des charges. GoGrid propose pour chaque client un nombre limité de machines qui ne peut pas être dépassé même si la capacité physique louée n'est pas encore entièrement consommée. Similairement à EC2, les machines virtuelles proposées par GoGrid ne peuvent pas être facilement transférées vers une autre infrastructure.

Plateforme sous forme d'un service (Platform as a Service PaaS) : PaaS est un modèle de fourniture de services consistant à offrir aux entreprises clientes un environnement de développement et de déploiement de leurs applications.

Principe : dans ce type de services, une entreprise cliente a uniquement la responsabilité du développement et du maintien de ses applications, puisque l'environnement de développement sous-jacent est externalisé et maintenu par le fournisseur de PaaS. Cet environnement se doit de masquer l'infrastructure aux yeux des développeurs, tout en leur fournissant les briques de base qui permettent de développer des applications de type SaaS. Il s'agit donc d'un niveau d'abstraction au-dessus de l'IaaS dans lequel le cloud fournit une plateforme applicative de programmation. Plus particulièrement, les services de ce modèle fournissent aux développeurs des fonctionnalités essentielles à l'exécution d'une application. Cela inclue aussi bien des fonctionnalités d'accès aux données, d'authentification, de gestion de transactions et de workflow ainsi que des fonctionnalités plus spécialisés pour créer des applications configurables (pour supporter par exemple la variabilité de SaaS). L'accès à la plateformes se fait via des API abstraites rendant la mise en œuvre de l'infrastructure (serveurs, stockage, CPU, réseaux) qui les exécute inaccessible aux clients. Lorsque l'utilisation d'une application développée et déployée sur la plateforme évolue, le fournisseur de PaaS est responsable de faire évoluer proportionnellement les ressources infrastructurelles qui lui ont été réservées.

Exemple : voici deux exemples de fournisseurs PaaS :

– *Microsoft Windows Azure (WA)* [Azu10] : WA est une plateforme qui vise à offrir les produits de Microsoft dans un modèle PaaS. Les services de cette plateforme sont regroupés en quatre catégories [RG11] :

1. La catégorie *.Net Services* qui offre des services de contrôle d'accès et d'organisation de workflow, développés en utilisant le cadre de développement .NET de Microsoft.
2. La catégorie *SQL Services* offre des services d'accès aux données fondées sur le système de gestion de base de données relationnelles SQL Server.
3. La catégorie *Live Services* qui offre des services orientés vers la création et le déploiement des applications sociaux. Ces services comprennent des fonctions de synchronisation des données entre différents appareils et applications, des services de gestion d'identité, des annuaires, de messagerie ainsi que des services géospatiaux et de services de recherche.
4. La catégorie *SharePoint Services* offre l'accès aux capacités de SharePoint dans le cloud.

Les développeurs utilisent les services de la plateforme pour construire leurs propres applications. l'utilisation de ces services doit être précédée par l'installation d'un SDK (Kit de développement logiciel) spécifique à l'environnement de développement supporté : *Visual Studio*. Les librairies de SDK sont accessible à partir d'une variété des langages tels que C#, Java, PHP et Ruby. Les postes de développement utilisés doivent toujours s'assurer d'une connexion Internet pour pouvoir communiquer avec les services de la plateforme et tester l'application avant le déploiement. Les applications construites sur WA dépendent fortement de cette plateforme comme elle est centrée uniquement autour de produits Microsoft et il n'y a pas d'autres fournisseurs PaaS qui fournissent de services similaires.

– *Force.com* [WB09] : Force.com est une plateforme de développement à la demande offerte par l'entreprise SalesForce. Cette plateforme est la plus prééminente en usage aujourd'hui. Elle héberge plus que 47.000 applications qui sont totalement isolées au niveau de leurs données et leurs codes. C'est grâce à une architecture mutualisée conduite par l'utilisation massives de *méta-données*, que l'équipe Force.com a pu réaliser cette plateforme et répondre aux exigences différentes et extrêmes de ce nombre très élevé de clients. Chaque application développée sur cette plateforme ainsi que son modèle de données, ses interfaces graphiques et son code métier

sont transformés en méta-données et sauvegardés dans une seule instance de base de données relationnelle centralisée. Cette représentation de l'application sous forme de méta-données permet sa modification d'une manière dynamique, sans affectés les autres applications hébergées (voir section 5.3.1 pour plus des détails sur le modèle de données de Force.com dirigé par les méta-données). L'utilisation et le développement des applications sur cette plateforme se fait directement à travers le web via un navigateur, sans aucun pré-requis et sans aucune installation des SDK sur les postes de développement.

Comme la plateforme WA, Force.com est une plateforme propriétaire. Actuellement, seul l'entreprise Salesforce offre cette plateforme. Aussi, Force.com supporte uniquement le langage de développement Apex qui ne peut être exécuté que sur cette plateforme, alors que autres langages de programmation couramment utilisés pour les applications d'entreprises tels que Java ou C# ne sont pas supportés. Le langage Apex est limité dans ses capacités, alors qu'un langage comme Java par exemple peut être utilisé pour construire pratiquement n'importe quel type d'application. En effet, Force.com est exclusivement destiné à la construction des applications légères. Le langage Apex est exécuté par ce que Force.com appelle une « *machine virtuelle, virtuelle* » [WB09], mise en place dynamiquement pour chaque nouvelle application pour surveiller l'exécution de son code et détecter les anomalies d'exécution.

Logiciel sous forme d'un service (SaaS) : les services de ce modèle fournissent une application à part entière. Contrairement à PaaS, les services à ce niveau visent principalement les utilisateurs finaux en fournissant une certaine facilité d'utilisation des applications métiers complexes via des interfaces ergonomiques (client riche). Comme SaaS est l'unique modèle du cloud orienté vers l'utilisation finale, il est souvent identifié comme le Cloud Computing par le public.

Principe : aujourd'hui le SaaS est un modèle économique de consommation de logiciels sur le Web utilisé à destination des entreprises tout comme des particuliers, et il fait maintenant partie de leur vie quotidienne. C'est pourtant l'illustration parfaite de la philosophie « Cloud » : plutôt que d'installer une application « en dur » sur un ordinateur, il suffit d'utiliser le navigateur pour se connecter, via un login et un mot de passe, à une application web située sur un serveur distant qui se chargera d'afficher les données appropriées directement sous la forme d'une page web dans le navigateur du client.

Exemple : dans cette sous-section, le système CRM (Customer Relation Management) de l'entreprise Salesforce [Sal09] est expliqué comme un exemple d'application SaaS typique bien connu et peut être facilement intégré avec les systèmes internes des entreprise clientes, via son interface de Web services. D'autres applications SaaS telles que *SAP entreprise by design* [SAP09] pour la planification de ressources d'une entreprise, *Cisco WebEx* [Cis09] pour les téléconférences et d'autres applications plus orientées vers les travaux bureautiques telles que *Google Apps* [GAp09] et *SkyDrive* [Sky08] sont tous déployées sur des infrastructures du cloud et accessibles actuellement à la demande sur Internet.

– *Salesforce CRM* [Sal09] : est un système CRM fournie par l'entreprise Salesforce dans un modèle SaaS mutualisé, et a été construit sur la plateforme Force.com, dont Salesforce est propriétaire. Les clients peuvent souscrire à cette application et commencer à l'utiliser instantanément. Un client peut adapter le système dans ces différents aspects tels que le modèle de données, les interfaces graphiques et le workflow, alors que tous les clients sont servis à partir d'une même instance d'application et de base de données. Donc, Salesforce est un bon exemple de mutualisation de services applicatifs du cloud.

Pour intégrer l'application avec les systèmes internes des entreprises clientes, une interface de service Web décrite en WSDL (Web Service Description Language, voir section 2.3.3) est fournie. Cette interface est offerte en deux variantes, le *partner WSDL* étant le même pour tous les clients et l' *entreprise WSDL* spécifique à chaque client. L'*entreprise WSDL* connecte l'application au système du client en lui fournissant les données et la logique d'une manière conforme à la configuration déjà spécifiée, alors que le *partner WSDL* est générique et peut donc être réutilisé par tous les clients.

Après avoir présenter les trois principaux modèles de fourniture de services du cloud, nous pouvons constater qu'ils diffèrent par le degré jusqu'auquel la pile technologique d'application, constituée des matériels, intergiciels et logiciels, est externalisée [LKN+09]. De même, ils diffèrent par le degré jusqu'auquel les couches de cette pile sont partagées entre les différents clients suivant le principe de mutualisation. Les fournisseurs de SaaS [TBB03] offrent des logiciels qui peuvent être utilisés par de multiples clients en même temps sans qu'ils se préoccupent de la gestion de l'application. Le modèle SaaS est souvent considéré comme une évolution du modèle ASP (Application Service Provider) [Tao01, CS+01] et propose la mutualisation de l'application et de sa base de données. Toutefois, les fournisseurs du cloud offrent non seulement des applications, mais aussi des plateformes de développement suivant le modèle PaaS [Law08a] et des machines virtuelles avec des capacités de stockage dans des infrastructures partagées suivant le modèles IaaS [Vog08]. Ces plateformes et infrastructures peuvent ensuite être utilisées par les clients pour déployer leurs propres applications SaaS. Dans ce qui suit les aspects d'externalisation et de mutualisation sont étudiés pour chacun de ces modèles de fourniture.

2.2.3.1 La découpe d'externalisation

Le terme *découpe d'externalisation* est utilisée ici pour désigner la frontière entre les couches de la pile technologique de l'application qui sont à externaliser d'une part et celles à prendre en charge par le client d'une autre part. La figure 2.3 montre le degré auquel cette pile technologique est externalisée dans les modèles suivants : On-Premise, IaaS, PaaS, ASP et SaaS.

Cette externalisation indique qu'une autre organisation ou entité organisationnelle est contractée pour gérer et maintenir une ou plusieurs couches de la pile technologique de l'application. Cette organisation peut être un fournisseur de service spécifique, ou la même organisation que celle du client de l'application. Dans le modèle « On-Premise » traditionnel, l'ensemble des couches de la pile technologique de l'application est exécuté sous la responsabilité de l'entreprise qui souhaite l'utiliser. Les modèles ASP et SaaS sont à l'opposé, puisque la pile technologique de l'application est entièrement externalisée. Toutefois, la découpe d'externalisation dans les modèle IaaS et PaaS varie selon chaque fournisseur. Le modèle IaaS fournit l'infrastructure, parfois en termes d'une machine physique ou virtuelle vierge (sans aucune installation), et d'autres fois avec un système d'exploitation installé. La même chose est aussi valide pour les services PaaS. Ces derniers proposent parfois un système d'exploitation intégré. Puisque la différence entre IaaS et PaaS est pas communément admise sur ce point, nous considérons qu'un système d'exploitation peut faire partie des couches de la pile technologique externalisées dans les deux modèles. Dans le cas où le système d'exploitation est fournie par un service IaaS, il peut être contrôler par le fournisseur et le client en même temps.

Suivant les principes du cloud, l'externalisation est transitive, c.à.d. qu'une entreprise qui souhaite réaliser une application peut l'externaliser à un fournisseur de SaaS qui à son tour externalise l'infrastructure nécessaire pour l'exécuter à un fournisseur IaaS. Ainsi, un fournisseur de SaaS réagit généralement comme un client et fournisseur en même temps.

2.2.3.2 La découpe de mutualisation

Un autre critère selon lequel sont classés les modèles de fourniture des servies du cloud, est celui de la *découpe de mutualisation*. Cette découpe montre quelles couches de la pile technologique d'une application sont exécutées en mode mutualisé et celles qui ne le sont pas. Nativement, le modèle On-Premise ne peut pas être mutualisé puisque l'ensemble de la pile technologique de l'application est exécuté pour un client sur sa propre infrastructure. Cela est également valide pour le modèle ASP dans lequel la pile technologique est séparément installée pour chaque client. Dans le modèle SaaS, l'application peut être mutualisée et l'infrastructure sous-jacente sera partagée entre tous les clients. Encore une fois, similairement à la découpe d'externalisation, les modèles IaaS et PaaS ne sont pas bien définis concernant la mutualisation des ressources qu'ils proposent. Dans le modèle IaaS, plusieurs clients sont répartis sur une infrastructure commune souvent via des machines virtuels qui peuvent être décalées et déplacées entre le matériel physique pour maîtriser les variations de charge. Si le fournisseur IaaS offre un serveur physique à chaque client, il ne peut pas être considéré comme mutualisé. Il en est de même pour le modèle PaaS, les composants de la plateforme peuvent être partagés ou séparément installés pour chaque client. Dans le modèle SaaS mutualisé, toute la pile technologique de l'application est partagée entre tous les clients.

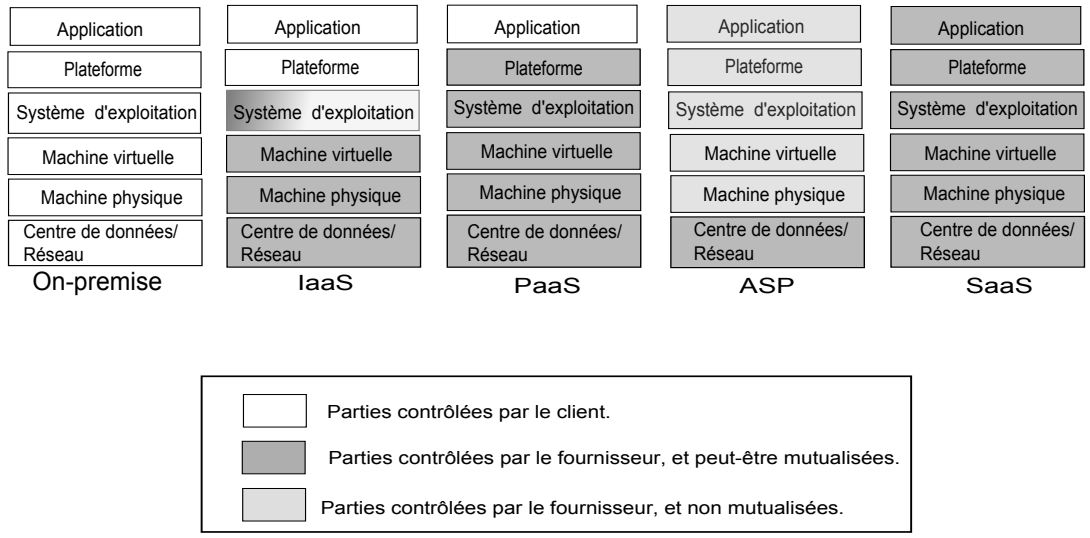


Figure 2.3 – Les différents degrés d'externalisation et de mutualisation de la pile technologique d'une application

2.2.4 Les modes de déploiements

Malgré le succès grandissant du cloud, il est à noter que de nombreuses entreprises hésitent à utiliser ses services [ZCB10, SK11]. Le plus souvent, les raisons de cette hésitation tournent autour de la sécurité. Les sources d'inquiétude principales des entreprises concernaient aussi bien la protection de secrets industriels critiques que le respect des obligations légales et la protection de données personnelles. De ce fait, il existe quatre modes différents pour déployer les ressources informatiques d'une entreprise dans

le cloud [MG11, DWC10, RES10] (voir figure 2.1 (d)), dont chacun est dédié à un scénario d'utilisation spécifique et présente ses propres avantages et inconvénients :

Le cloud privé : dans ce mode, l'ensemble des ressources du cloud sont uniquement exploitées par une seule entreprise, et peuvent être gérées par elle-même ou par une tierce partie. La motivation d'installer un cloud privé est de moderniser l'infrastructure interne et de profiter de certaines caractéristiques du cloud, tout en gardant un niveau de sécurité élevé. Ce mode de déploiement comporte plusieurs avantages. Il optimise avant tout l'utilisation de ressources internes à travers la technologie de virtualisation. Ensuite, il élimine les problèmes de sécurité y compris la confidentialité et la confiance. Et finalement, il assure un plein contrôle sur les activités critiques. Cependant, ce mode est souvent critiqué pour être semblable à l'installation de serveurs propriétaires traditionnels qui ne fournissent pas des avantages du cloud tels que l'absence des coûts d'investissement initiaux et la facturation selon l'usage.

Le cloud public : dans ce mode, les ressources sont mises à disposition du grand public et appartiennent à un ou plusieurs fournisseurs appliquant leur propre politiques de gestion et modèles de facturation. Bien que cela offre plusieurs avantages clés concernant la réduction des investissements initiaux et le déplacement des risques, les clients d'un cloud public ne disposent pas d'un contrôle fin sur leurs propres données et paramètres de sécurité. Pour les plus grandes entreprises (celles qui peuvent s'offrir les solutions métiers les plus élaborées), le cloud public n'est pas encore indispensable pour les applications critiques ou pour celles qui traitent des informations les plus sensibles. Cependant, pour les plus petites entreprises, dont les ressources informatiques sont limitées, les services du cloud public peuvent s'avérer plus efficaces que les applications susceptibles d'êtres déployées en interne en termes de réduction des coûts de gestion et de maintenance.

Le cloud hybride : un cloud hybride est une combinaison entre les deux modes de déploiements précédents, qui tente à trouver l'équilibre entre leurs avantages et leurs inconvénients. Dans ce mode, une partie des ressources considérées comme très sensibles est déployée dans un cloud privé alors que la partie restante est déployée dans un cloud public. Les organisations utilisent généralement ce mode pour à la fois minimiser les coûts et garder un niveau de sécurité acceptable. Même si ce mode de déploiement offre une plus grande souplesse, la conception d'un cloud hybride nécessite la détermination soigneuse de la meilleure répartition des ressources et la gestion de leur mouvement entre les deux clouds.

Le cloud virtuellement privé : ce mode offre une nouvelle alternative pour combiner les clouds publics et privés. La principale différence avec le cloud hybride est que ce mode déploie les ressources entièrement dans un cloud public alors qu'il protège leur accès à travers la technologie VPN (Virtual Private Network). Celle-ci virtualise l'accès aux ressources via le réseau et laisse à chaque client la possibilité de définir sa propre topologie et ses propres mesures de sécurité.

Pour la plupart des entreprises, le choix du meilleur mode pour déployer leurs ressources sur le cloud dépend principalement de leur objectif et de la nature de leur travail. Par exemple, les applications de type commercial sont mieux déployées sur des infrastructures publiques pour la rentabilité. Ainsi, nous allons avoir certains modes de déploiement plus populaires que d'autres. En particulier, il a été prédit que le cloud hybride sera le type dominant du marché [ZCB10]. Cependant, le cloud virtuellement privé a commencé à gagner en popularité depuis sa création en 2009 par l'entreprise Amazon. À noter que les modèles de fourniture sont orthogonaux aux modes de déploiements. À titre d'exemple, une application SaaS peut être déployée sur une infrastructure d'un cloud public ou privé.

2.2.5 Les limitations et les obstacles

Malgré tous les efforts et les avancées technologiques réalisés, il reste encore un certain nombre de limitations qui font du Cloud Computing un modèle imparfait. À travers notre analyse de l'état de l'art concernant ce sujet [DWC10, Vou08, JSGI09, AFG+10, TJA10], nous avons pu identifié les limitations suivantes :

La disponibilité et l'accessibilité : les engagements de performances et de disponibilité de services du cloud pris par les fournisseurs (comme par exemple 99,9% de disponibilité du service [KPR09]) ne sont pas toujours possible à tenir. Il peut se produire des coupures de réseau indépendantes du fournisseur qui auront d'énormes conséquences pour l'entreprise qui aura choisi d'externaliser son système d'information. Cela affecte ainsi l'accessibilité à ces services et rendent impossible de se connecter et de continuer à travailler. Pour cela, de nouveaux outils commencent à apparaître visant à permettre un stockage local des informations afin de pouvoir continuer à travailler sur une application même hors-ligne, tel que l'exemple de *Google Gears*. Ce dernier permet l'accès hors-ligne à des services qui fonctionnent normalement en ligne, à travers un moteur de base de données à installer dans l'infrastructure du client pour mettre en cache les données.

La gouvernance : un des principaux problèmes des services du cloud actuels concerne le stockage des données. Si l'entreprise cliente manipule des données sensibles, il est nécessaire qu'elle sache où elles se trouvent. De plus, la directive européenne 95/46/EC de 1995 impose (pour certains métiers) que les données relatives à l'entreprise et à ses clients soient hébergées en Europe, ce qui est aujourd'hui difficile compte-tenu de la localisation des centres de données (majoritairement sur le sol américain). Pour augmenter leur part de marché, les fournisseurs de services cloud devront donc s'adapter en proposant des offres qui garantissent la localisation géographique des serveurs qui seront utilisés pour le stockage des données. Pour cette raison, Google propose à ses clients utilisant sa plateforme de développement GAE (Google App Engine) [Ciu09], de déployer leurs applications et leurs données dans l'un de ses trois centres de données européens en Irlande, en Finlande et en Belgique. Google justifie cette ouverture comme un moyen de rapprocher les solutions des clients pour une meilleure performance et une redondance européenne. Il en est de même pour des hébergeurs grandissant mais de taille plus modeste tel que OVH.

L'intégration et le développement : un défi important des services du cloud est de pouvoir correctement s'intégrer dans les architectures très complexe des systèmes existants d'entreprises clientes qui se sont développées au fil des années et qui sont souvent très hétérogènes (composées par exemple de mainframes, de systèmes informatiques possédant leurs propres applications installées en dur, d'une partie client/serveur, etc.). Certains de ses systèmes étant historiques et vitales pour les entreprises, leur migration est loin d'être simple. Il faut donc que les services du cloud puissent dans un premier temps s'intégrer dans ces architectures complexes sans les perturber. Des solutions telles que les ESB (Enterprise Service Bus) [Men07] existent déjà mais ne sont pas toujours réellement adaptées aux besoins des entreprises. Certains services de type PaaS proposent des solutions à ces problème d'intégration (tel que la plateforme Force.com) mais l'un des reproches régulièrement adressé à ces services est la contrainte de l'environnement de développement. En effet, ces plateformes sont le plus souvent développées et optimisées pour des langages spécifiques, ce qui oblige les développeurs à se conformer à un environnement de développement imposé. Cela représente évidemment une contrainte du passage au cloud.

La portabilité et l'interopérabilité : l'un des obstacles qui freine encore la migration de nombreuses entreprises vers des services cloud est la portabilité des données. En effet, par manque de standardisation, mais aussi pour leur propre intérêt, la plupart des fournisseurs de services cloud ont un

modèle de stockage de données et de déploiement d'applications qui leur est propre, ce qui signifie que même dans le cas où il sera possible pour une entreprise de récupérer ses données externalisées, elles sont inexploitables. Donc cela veut dire qu'une entreprise ayant choisi un fournisseur de services cloud ne pourra plus en changer, même quand la concurrence serait plus performante. Cette situation pousse donc de nombreuses entreprises à rester prudente avant de se lancer dans le cloud.

2.2.6 Vers une gestion globale sous forme d'un service (Toward offering everything as a Service XaaS)

Les pressions ont été longtemps exercées sur les entreprises et le seront toujours dans le but d'accroître leur efficacité pour contrôler et réduire leur dépenses de manière plus efficace. Cependant, l'industrie logicielle a montré que cette efficacité a souvent été limitée par les modèles de livraison classiques dans lesquels les applications logicielles sont installées au sein de l'infrastructure informatique des entreprises et gérées sous leur responsabilité, ce qui a souvent conduit à de gros investissements par rapport aux résultats escomptés. De ce fait, les entreprises se sont tournées vers des modèles d'hébergement hors établissement à travers l'externalisation, en déplaçant la charge de maintenance de leurs systèmes informatiques vers des fournisseurs externes. Toutefois, il serait mal avisé de se concentrer uniquement sur l'externalisation et le transfert de services informatiques au lieu de choisir la façon dont ces services seront fournis et suivant quel modèle économique. Dans cette direction, les services du cloud ont monté en puissance. Ils proposent un modèle économique innovant et une facilité d'utilisation grâce à des nombreuses avancées technologiques telles que la virtualisation de matériels physiques, l'architecture orientée services et la couverture globale des connexions internet à haut débit. Ainsi, une fois les caractéristiques du cloud sont bien établies, la prochaine étape pour de nombreuses entreprises sera celle de savoir à quel point les différents aspects de l'informatique peuvent être fournis sous forme des services. En fin de compte, tout devient potentiellement un service.

pour cette raison, nous avons commencé à remarquer de plus en plus l'apparition d'une multitude de nouveaux types de services basées sur les caractéristiques essentielles du cloud, tels que *DaaS : Data as a Service*, *TaaS : Test as a Service*, *FaaS : Framework as a Service*, *CaaS : Composition as a Service*, etc. [RCL09], l'acronyme *XaaS : Everything as a Service* [RCL09] a été créé pour désigner ces types de services en raison de leur nombre important qui ne cesse d'augmenter.

Toutefois, il est à rappeler que la notion de service n'est pas une nouveauté du cloud. Ce concept, datant du début des années deux mille (inspiré par la mondialisation dans le commerce actuel), s'intégrait initialement dans une réorientation des architectures logicielles vers un nouveau modèle d'architecture orientée services dont le but affiché est de surmonter des obstacles de développement importants concernant la réutilisation, la baisse des coûts de développement et l'intégration des applications dans des environnements d'exécution hautement volatiles et hétérogènes [HK11]. Dans la section suivante, nous allons détailler ce modèle d'architecture et expliquer les principes de construction qu'il propose. Nous expliquerons également sa relation avec le cloud et comment ses deux paradigmes peuvent être complémentaires.

2.3 Architecture Orientée Services (AOS)

L'AOS a principalement émergé comme un modèle d'architecture pour modéliser et construire des systèmes informatiques distribués [Pap03, PVDH07]. Elle encourage une organisation visant à faciliter la réutilisation des fonctionnalités existantes et à permettre l'atteinte de nombreux bénéfices souhaités

tels que l'amélioration de productivité et la baisse des coûts de développement. L'un des principaux avantages revendiqués pour l'AOS est celui de la possibilité de construire de solutions flexibles qui peuvent réagir à l'évolution des exigences métiers rapidement et de manière économe. En effet, l'AOS met l'accent sur des approches de conception faiblement couplées où des systèmes distants développés sur des plateformes différentes peuvent collaborer et évoluer sans l'introduction de changements majeurs sur leurs architectures existantes.

2.3.1 La notion de service

Le service est la brique de base de l'AOS. Il représente une entité logicielle fonctionnelle déployée et invocable à distance [HK11]. Un service est lié à un fournisseur de services et est utilisé par des clients à partir de sa description. Celle-ci décrit les propriétés du service qui représentent sa particularité métier et permet aux clients de décider si ce service correspond à leurs besoins. De nouvelles applications peuvent ensuite être assemblées en construisant de nouveaux services et récursivement composer les services existants pour atteindre la fonctionnalité désirée. Une composition de services est ainsi fournie à d'autres applications en tant que service. Les services qui ne peuvent pas être décomposés (ou dont la composition est cachée) sont appelés des *services atomiques*. Les services qui sont constitués à partir d'un ensemble d'autres services sont appelés *services composites* (confère section 2.3.4 pour une définition plus détaillée) . Les services (atomiques ou composites) sont caractérisés par une interface bien définie décrivant les propriétés fonctionnelles et non-fonctionnelles du service [Mie10]. Celles-ci se déclinent principalement en :

Propriétés fonctionnelles : elles fournissent les informations nécessaires à l'invocation et à la bonne utilisation des fonctionnalités du service. Ces informations peuvent être catégorisées en partie syntaxique et partie sémantique. La partie syntaxique regroupe les informations strictement nécessaires pour communiquer avec le service (signature, protocole de communication, etc.). La partie sémantique regroupe les informations qui permettent de bien utiliser le service (sémantiques attachées aux fonctions et aux données échangées, comportement extérieur, ordonnancement des invocations aux différentes fonctionnalités, etc.) [HK11].

Propriétés non-fonctionnelles : elles fournissent les informations sur la qualité de service (QoS, performance, coût, sécurité, etc.) et sur les politiques du fournisseur (localisation, législations d'exploitation du service, utilisation des données clients, etc).

2.3.2 L'organisation de l'AOS

Les services de l'AOS sont conçus comme des modules autonomes qui peuvent être facilement réutilisés grâce à une organisation dédiée (voir figure 2.4). Selon cette organisation, les services sont publiés par les fournisseurs de services dans un registre de services [BCE+02, CDK+02]. Ce dernier est ensuite consulté par le client afin de déterminer le service qui correspond à ses besoins tout en analysant sa description fonctionnelle et non-fonctionnelle. Une fois le service découvert, le client récupère sa description à partir du registre et l'intègre dans son application pour l'invoquer. Le fournisseur et le client seront ensuite liés par l'utilisation du service. Tous deux s'engagent à respecter un contrat d'utilisation, en terme de respect de l'interface du service pour le client, et de respect des propriétés fonctionnelles et non-fonctionnelles promises par le fournisseur. Dans une AOS, l'invocation d'un service peut être statique (au moment de la conception ou du déploiement de l'application) : le client interroge le registre pour sélectionner les services souhaités. L'invocation peut également être dynamique à travers l'utilisation des mécanismes spécifiques [CM07, AP07, KSSA09]. Une des propriétés de l'AOS qui nous intéresse est

que son modèle d'architecture ne se base sur aucune technologie d'implémentation spécifique. Les services peuvent être distribués par exemple sur des machines différentes, dans des entreprises et des pays différents [CLS+05, Pap03].

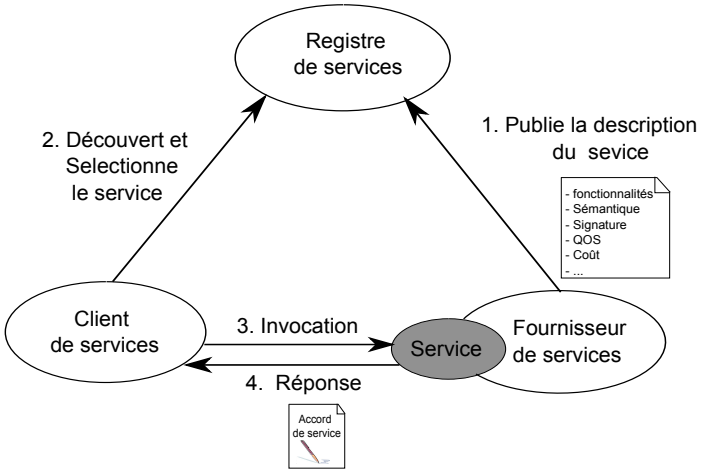


Figure 2.4 – Organisation de l'architecture orientée services

2.3.3 Les services Web

Les *services Web* sont certainement la technologie la plus connue et la plus populaire dans les mondes industriel et académique pour la mise en place d'une architecture orientée services. *La pile technologique de services Web (Web Service stack)* [CLS+05] fournit un ensemble de technologies et de spécifications nécessaires pour cette mise en place. Ainsi, les services sont appelés *services Web* dans cette pile technologique.

L'interface du service est décrite en utilisant un langage de description de services Web appelé *WSDL (Web Service Description Langage)* [CMRW07]. Un fichier WSDL comprend une description des fonctionnalités d'un service, mais il ne se préoccupe pas de l'implémentation de celles-ci. Il contient aussi des informations concernant la localisation du service, ainsi que les données et les protocoles à utiliser pour l'invoquer. En pratique, le fichier WSDL est un fichier XML qui se divise en deux parties :

- La **définition abstraite** de l'interface du service avec les opérations supportées par le service Web, ainsi que leurs paramètres et les types des données.
- La **définition concrète** de l'accès au service avec la localisation, par une adresse réseau du fournisseur de service, et les protocoles spécifiques d'accès.

Les propriétés non fonctionnelles des services sont décrites en utilisant un standard appelé *Web Service Policy (WS-Policy)* [BBC+06].

Le registre de services est aussi un standard de cette pile appelé *UDDI (Universal Description Discovery and Integration)* [CLS+05]. Il s'agit d'une spécification d'annuaire qui propose d'enregistrer et de rechercher des fichiers de description de services Web correspondant aux attentes d'un client. L'UDDI a été initialement conçu par et pour des industriels en ayant pour but d'avoir un standard indépendant

des plateformes d'implémentation. Pour simplifier la recherche, l'UDDI classifie les services dans des catégories selon leurs fonctionnalités et leurs domaines cibles.

Les services Web permettent de mettre en œuvre une architecture orientée services sur des infrastructures hétérogènes et distribuées. Il existe actuellement plusieurs plateformes de développement pour implémenter les services et les déployer sur Internet en utilisant différents langages de programmation tels que Java, C#, C, PHP, et autres. Pour être en mesure d'échanger des messages entre les services hétérogènes, une architecture de messagerie qui abstrait les particularités des différents langages de programmation est nécessaire. Le standard *SOAP (Simple Object Access Protocol)* [CLS+05] fournie une telle abstraction. SOAP est un protocole dont la syntaxe est basée sur XML dont le but principale est de permettre des échanges standardisés de données. Initialement proposé par Microsoft et IBM, la spécification SOAP est aujourd'hui une recommandation W3C. Le protocole SOAP s'appuie sur des standards de communication comme le protocole HTTP, mais il peut aussi utiliser d'autres protocoles tels que SMTP et FTP. Toutefois, l'avantage d'utiliser SOAP avec le protocole HTTP est que la communication est facilitée, en particulier les proxys et les pare-feu peuvent être franchis sans problème.

Il est important de distinguer à ce niveau entre les services Web et l'AOS, en ce sens que l'on peut mettre en place une AOS sans utiliser la pile technologique fournie par les services Web. Réciproquement, on peut utiliser les services Web sans nécessairement faire de l'AOS.

2.3.4 La composition de services

La composition est le processus de construction d'une nouvelle entité logicielle à partir d'autres entités préexistantes. L'entité qui en résulte est dite composite. Cette dernière représente la réification de la collaboration entre les services de manière à exposer le résultat de cette collaboration comme un service à part entière [HK11].

2.3.4.1 Orchestration et chorégraphie

Dans l'état de l'art, une composition de services peut être effectuée soit par *orchestration*, soit par *chorégraphie* [Pel03, PZ06]. Sans entrer dans une liste de définitions et de leur différenciations, nous pouvons définir les éléments conceptuels qui leur spécifient :

L'**orchestration** correspond à un schéma de collaboration centralisée, où une seule entité réunit l'ensemble des appels aux différents services de la composition. Cette entité invoque les services dans un ordre prédéfini. Elle se charge de la récupération des différents résultats et du transfert des données pertinentes pour l'exécution correcte des services invoqués [HK11].

La **chorégraphie** correspond à un schéma de collaboration distribuée où les échanges de messages se font directement entre les services en coopération afin d'effectuer une tâche particulière. Plusieurs services sont considérés dans une chorégraphie où chaque service décrit sa propre partie dans la collaboration [PZ06].

Orchestration et chorégraphie correspondent donc à deux approches de composition. Dans la pile technologique de services Web, l'orchestration est typiquement employée dans une perspective privée, c.à.d. le service composite qui centralise les invocations aux services constituants est une boîte noire. La chorégraphie adopte, de son côté, une perspective publique où les échanges de messages entre les différents services constituants sont directement visibles de l'extérieur [HK11].

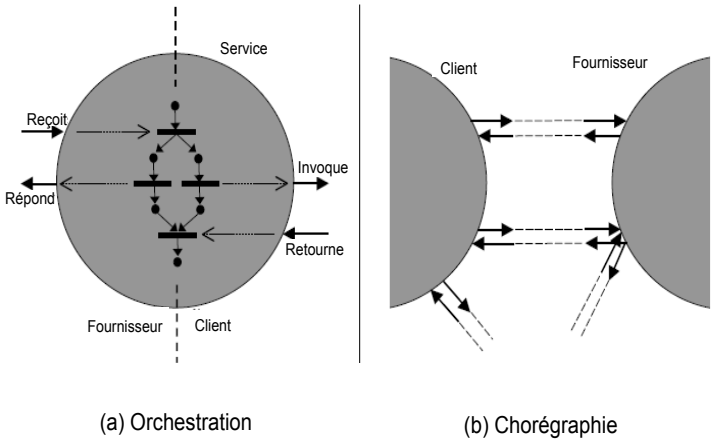


Figure 2.5 – Orchestration et chorégraphie de services [PZ06]

2.3.4.2 Composition statique et dynamique de services

La composition de services (par orchestration ou chorégraphie) peut être statique ou dynamique [DS05, FS04]. Ceci représente la façon dont un service Web est sélectionné comme un composant et utilisé dans l'application à base de services. Nous allons détailler chaque type de composition :

La **composition statique** de services correspond à une orchestration ou une chorégraphie prédéfinie et effectuée lors de la conception, où l'architecture et la conception du système logiciel à base de services sont planifiées. Les services à utiliser sont choisis, liés ensemble, et enfin compilés et déployés. Cela peut bien fonctionner aussi longtemps que l'environnement d'exécution et les exigences de clients ne subissent pas des changements. Dans le cas contraire, des incohérences peuvent être causées. Ainsi, il est inévitable de modifier l'architecture de l'application par la modification de services utilisés dans la composition et le redéploiement de celle-ci après son interruption. De ce fait, la composition statique de services ne fournie pas la souplesse et l'agilité requises dans les cas où il y a de fréquentes modifications dans les exigences et quand les circonstances opérationnelles ne peuvent pas être anticipées. La composition statique est suffisante pour construire des applications bien définies avec des besoins spécifiques qui ne sont pas susceptibles de changer fréquemment. La refonte du système pour s'adapter aux changements nécessite souvent une intervention humaine, ce qui ralentit considérablement la réaction globale au changement. La modification ou la mise à jour d'une composition statique exige habituellement d'arrêter l'exécution du système, d'où le besoin crucial pour de nombreuses applications métiers d'être capable de réagir dynamiquement face aux variations de l'environnement et des exigences de clients.

La **composition dynamique** de services est une étape importante dans la réalisation du besoin précédemment cité. Elle améliore la flexibilité des systèmes car elle permet l'exécution de nouveaux services, même s'ils ne sont pas encore découverts ou conçus à l'avance. Les services peuvent être assemblés sur la base des exigences du système ou de ses clients. Ces derniers n'ont pas besoin d'être interrompus pendant la mise à jour ou l'ajout d'une nouvelle fonctionnalité. Ainsi, la composition dynamique de services fournit une capacité certaine pour rapidement et automatiquement (avec une intervention humaine minimale) adapter le système même à des changements qui n'ont pas été envisagés lors de la conception, tout

en gardant le système logiciel disponible en permanence. L'application ne se limite plus à l'ensemble initial d'opérations qui ont été spécifiées au moment de la conception. Les capacités de l'application peuvent ainsi être étendues à l'exécution.

Toutefois, il y a un certain nombre d'aspects à prendre en considération lors de la réalisation d'une composition dynamique. Mis à part la performance et la robustesse des mécanismes de composition, la question du temps de réponse est la plus cruciale. En effet, une composition dynamique de services est généralement plus lente à exécuter en raison de la découverte et de la sélection de services qui sont effectuées lors de l'exécution, alors que ces deux étapes sont effectuées lors de la conception du système dans le cas de la composition statique. De ce fait, la majorité des approches utilisent la composition dynamique pour manipuler uniquement un ensemble prédéfinis de services, ceux destinés à être remplacés ou désactivés lors de l'exécution [MRD08].

2.3.5 Les couches de l'AOS

L'AOS vise principalement l'abstraction de langages et de technologies à travers la réutilisation standardisée de services et leur alignement avec la stratégie de l'entreprise pour une adaptation rapide sur l'évolution des besoins [EWA06]. En outre, l'AOS permet l'intégration des applications existantes en exposant leur fonctionnalités comme des services pour améliorer la valeur du logiciel et éviter la redondance lors de la construction. Pour obtenir ces avantages et soutenir la base de l'orientation service, ils existe plusieurs couche d'abstraction dans toute architecture orientée services [Ars04, ELK+06, AZE+07, CK07] (voir figure 2.6). Le but de ces différentes couches est de rendre les systèmes informatiques à base de services plus transparents pour les architectes et les développeurs. Ces couches consistent principalement en une composition de services alignée avec le processus métier de l'application. Les services exposés sont implémentés par des composants à l'échelle de l'entreprise qui réalisent les services et sont chargés de maintenir leur fonctionnalité et leur qualité. Les flux du processus métier sont pris en charge par une orchestration de ces services exposés qui sont surveillés et gérés pour assurer leur qualité et le respect des exigences non-fonctionnelles. La figure 2.6 montre une représentation des couches de l'AOS explicitées ci-après :

La couche d'infrastructure : est composée des ressources de données traitées dans les couches supérieures. Elles consistent principalement des systèmes existants (CRM, ERP, etc.), des base de données et des systèmes de stockage de fichiers. Les couches supérieures de l'architecture traitent généralement les données récupérées à partir de cette couche.

La couche de composants : est composée de composants qui sont responsables de la réalisation des services exposés, et du maintien de leur qualité. Ces composants forment un ensemble de fonctionnalités gérées, administrées et contrôlées par l'entreprise ayant financé leur développement. Ils sont ainsi chargés d'assurer la conformité aux accords de niveau de service à travers l'application de bonnes pratiques architecturales (ex : application des patrons de conception). Cette couche utilise généralement des technologies basées sur des conteneurs (container-based technologies) telles que des moteurs d'exécution ou des serveurs d'applications pour faire exécuter ses composants lors de l'invocation des services qu'ils implémentent [Ars04].

La couche de services : est composée de services que l'entreprise choisit d'exposer. Ils sont découverts, statiquement et/ou dynamiquement invoqués et orchestrés en un service composite. Cette couche prévoit ainsi le mécanisme qui transforme les composants fabriqués à l'échelle de l'entreprise en des services exposés à une grande échelle. Les composants d'entreprise fournissent la réalisation de ces services lors de l'exécution à l'aide de la fonctionnalité fournie par leurs interfaces [IB07]. Les

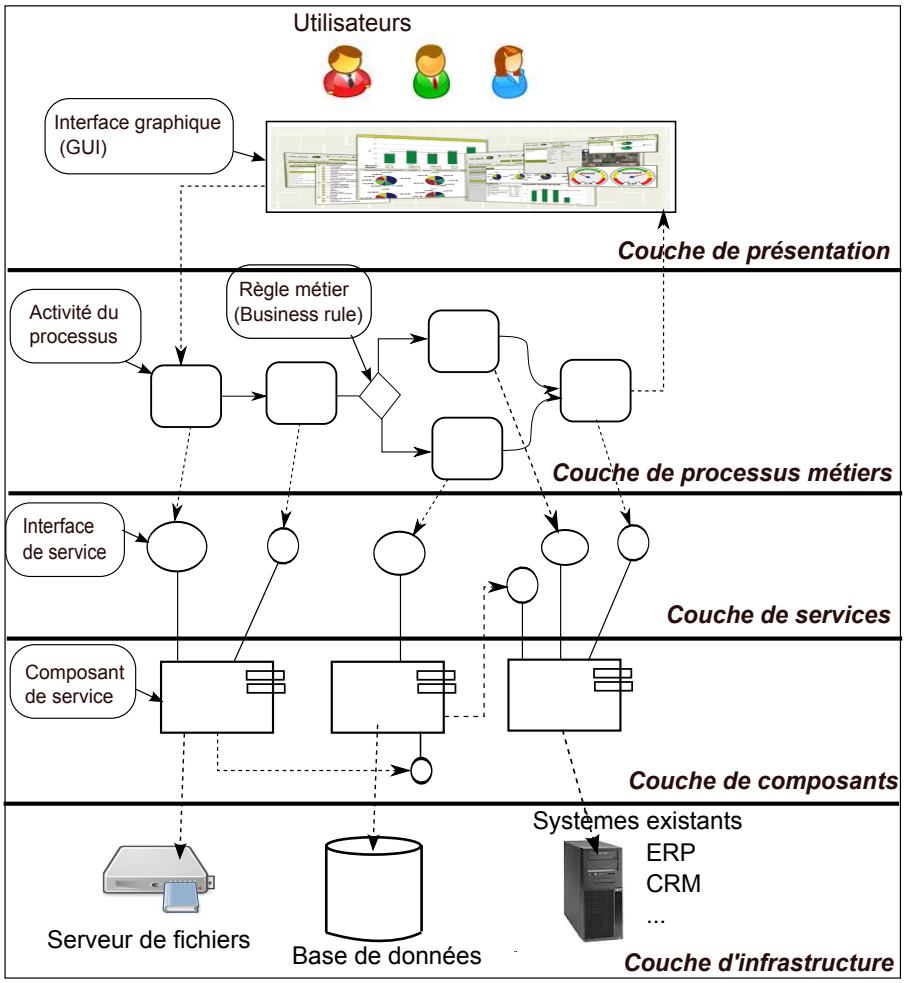


Figure 2.6 – Les couches de l'AOS [AZE+07]

interfaces sont exportées comme des descriptions de ces services et seront exposées à l'utilisation dans un registre de services spécifique.

La couche de processus métiers : est composée des orchestrations de services exposés dans la couche précédente [CK07]. Les services sont regroupés à travers les orchestrations, et donc agissent ensemble en une seule application qui réalise les objectifs métier. Un processus métier est ainsi la fonctionnalité qui organise l'orchestration des services exposés à travers des activités et des règles métiers. Une activité du processus modélise une invocation d'un service ou un traitement spécifique avant ou après cette invocation, tandis qu'une règle métier modélise un point de décision du processus qui contraint l'exécution des activités sous l'occurrence de certaines conditions.

La couche de présentation : est composée des interfaces graphiques permettant l'interaction des utilisateurs avec l'application. Bien que cette couche soit généralement hors de la portée des discussions autour d'une architecture AOS, elle devient de plus en plus importante. L'apparition des nouvelles technologies Web et plus particulièrement les RIA permet l'invocation directe de services

(atomiques ou composites) à partir des navigateurs Web sans avoir à passer par des médiateurs ou des adaptateurs.

2.4 Cloud et AOS

Après avoir expliquer le Cloud Computing et l'AOS en détails, nous allons nous concentrer dans cette section sur la relation entre ces deux paradigmes. Cela concerne leurs points de convergence et leurs points de divergence.

2.4.1 Les points de convergence

En principe, l'AOS et le Cloud Computing peuvent être complémentaires [Rai09, WVLY+10, DWC10]. Tous les deux jouent un rôle important dans l'informatique des entreprises. Ils peuvent être adoptés indépendamment étant deux paradigmes différents ou en même temps, l'OAS pouvant fournir une valeur ajoutée pour soutenir les efforts engagés dans le cloud [Rai09]. Cela est le plus souvent remarqué au niveau applicatif du cloud, où les applications SaaS sont généralement construites suivant les principes de l'AOS [GSH+07, LZV08, MMLP09], et se traduit généralement par l'intégration des composants logiciels distribués lors de la construction de ces applications en utilisant les standards de l'AOS (WSDL, SOAP, UDDI, etc.). l'AOS est aussi essentielle pour la réussite des autres types de services du cloud et plus particulièrement le PaaS et le DaaS [APG+10] par la facilité de réutilisation et d'intégration qu'elle propose.

Le cloud et l'AOS partagent également le concept d'orientation service. Actuellement, nous trouvons de nombreux types de services mis en disposition sur un réseau commun pour une utilisation par les clients. Le cloud met particulièrement l'accent sur les aspects transformant la pile technologique nécessaire pour mettre en place une application en un service (via le modèle SaaS) pour être utilisé auprès des clients. Ainsi, nous pouvons considérer que le Cloud Computing est actuellement un terme qui englobe l'AOS. Celle-ci, mise en œuvre dans la pratique sous forme d'une composition de services, peut être exécutée sur un réseau à travers l'utilisation des plateformes du cloud. Par exemple, la plateforme GAE fournit un environnement de développement à part entière dans lequel les développeurs peuvent développer et déployer leurs applications à base de services.

De même, le cloud et l'AOS se basent sur une certaine forme de relations contractuelles et de confiance entre les fournisseurs de services et les clients consommateurs de services. La réutilisation des fonctionnalités d'un service AOS par un ou plusieurs autres systèmes informatiques est en effet une « externalisation » de cette fonctionnalité à une autre organisation [GR12]. Avec le cloud, l'externalisation est plus souvent manifestée en saveur économique, où les services du cloud sont loués auprès des fournisseurs qui ont tendance à passer vers une très grande échelle de clients. Toutefois, l'externalisation implique que le cloud et l'AOS ont besoin d'un solide réseau pour assurer la connexion entre les fournisseurs et les clients. De ce fait, ces deux paradigmes présentent les mêmes faiblesses structurelles et fondamentales lorsque le réseau ne fonctionne pas ou n'est pas disponible [Rai09].

2.4.2 Les points de divergences

Bien qu'il y ait des complémentarités et des points communs entre le cloud et l'AOS, ils présentent quelques divergences provenant des problématiques différentes auxquelles chaque paradigme s'attaque. Les technologies d'implémentation de l'AOS, tels que les services Web, permettent à une application de composer des services à travers un réseau, et permettent également l'intégration de ces services au

travers d'une variété de langages et de plateformes de développement générant en fin de compte une application qui fait abstraction du langage. Un avantage clé pour l'entreprise qui utilise l'AOS pour développer des applications réside dans la capacité de créer des interfaces cohérentes de différents systèmes dans l'architecture de l'entreprise, économisant ainsi les efforts sur l'intégration future et améliorant la vitesse à laquelle l'intégration peut se produire. Toutefois, le Cloud Computing consiste à exploiter le réseau pour externaliser tout ou une partie de la pile technologique nécessaire pour mettre en œuvre une application. Même si cela peut inclure des composants applicatifs tel que c'est le cas de l'AOS, le cloud va beaucoup plus loin. En effet, il permet d'offrir de nombreuses fonctionnalités en tant que services qui réduisent énormément les coûts d'investissement pour les clients et ce, en comparaison avec des anciens modèles de fournitures tels que l'installation sur site ou le modèle ASP. Par conséquent, malgré que les deux concepts partagent de nombreuses caractéristiques, nous ne les considérons pas comme équivalent et cependant, ils peuvent être adoptés indépendamment ou même en tant qu'activités concurrentes.

2.5 Conclusion

Dans ce chapitre, nous avons expliqué dans un premier temps le modèle du Cloud Computing dans ces différent aspects et caractéristiques essentielles considérées comme les résultats des années de recherches effectuées sur la virtualisation, l'informatique distribué et utilitaire, les grilles de calcul, et l'architecture orientée services. Les caractéristiques du cloud conduisent actuellement une évolution dans l'industrie logicielle et obligent les entreprises à mener des réflexion approfondies sur leur organisation pour développer de nouvelles approches et démarches concernant l'externalisation de leurs services informatiques vers un nouveau monde technologique et modèle économique. Les services du cloud sont divisés en trois modèles de fourniture désignant la frontière entre les couches de la pile technologique d'une application qui sont à externaliser d'une part, et celles à prendre en charge par les clients d'une autre part. Cette externalisation peut concerner l'infrastructure de l'application suivant le modèle IaaS, sa plateforme de développement suivant le modèle PaaS et l'application elle-même suivant le modèle SaaS.

Nous avons également détaillé dans ce chapitre les principes de l'AOS, comme un modèle d'architecture logicielle bien adapté pour la construction d'application SaaS. Dans cette optique, nous avons détaillé le fonctionnement global du paradigme service à travers les préoccupations qui ont dirigé sa définition. Nous avons décrit la notion de service et l'organisation de toute architecture orientée services. Nous avons également résumé les supports techniques qui soutiennent les bénéfices apportés par ce modèle de construction, tels que la facilité d'intégration et de réutilisation des services, et plus particulièrement la composition dynamique de services. Ces caractéristique de l'AOS sont des éléments clés pour les contributions que nous avons apporté dans cette thèse, notamment au niveau de la gestion dynamique de la variabilité des applications SaaS mutualisées, celles que nous allons détailler dans le chapitre suivant.

Les Applications SaaS Mutualisées Dans le Cloud

3.1 Introduction

SaaS, introduit dans la section 3.2, est un type des services du cloud selon lequel un fournisseur publie son logiciel pour plusieurs clients sur Internet. L'environnement du cloud (en termes d'infrastructures et de plateformes) assure l'exécution et la maintenance de l'ensemble des ressources informatiques privées à ces clients par l'intermédiaire de SaaS. De remarquables offres SaaS existent actuellement sur le marché et gagnent un succès considérable auprès des clients. Parmi ces applications, nous citons *Salesforce.com* [Sal09] qui fournit des solutions pour la gestion de la relation client (CRM) ; *SAP entreprise by design* [SAP09] pour la planification des ressources d'une entreprise ; *Cisco WebEx* [Cis09] pour les téléconférences ainsi que d'autres applications plus orientées vers les travaux bureautiques telles que *Google Apps* [GAp09] et *SkyDrive* [Sky08], etc. Toutes ces applications sont actuellement déployées sur des infrastructures du cloud et accessibles à la demande sur Internet.

Afin de réduire le coût total de maintenance et de la complexité opérationnelle, certaines de ces applications utilisent une approche mutualisée qui consiste à héberger tous les utilisateurs des différents clients sur la même instance de l'application. Les utilisateurs d'un client représentent généralement un groupe fermé qui est habituellement chargé et manipulé comme une seule entité désignée en tant que *locataire*. L'application est donc conçue pour servir de multiples locataires avec une seule instance d'exécution. Le partage des ressources se traduit en général par la réduction des coûts dans un environnement du cloud et ceci est le plus souvent remarqué sur le niveau de SaaS (par rapport à IaaS et PaaS), puisque les coûts en terme de maintenance, de mise à jour et d'évolution deviennent les plus minimes [MK11]. Toutefois, pour pouvoir répondre aux attentes des clients en matière de niveau de service, les applications mutualisées doivent assurer un isolement sans faille des locataires et la satisfaction de leurs besoins spécifiques en particulier lorsque l'instance d'application elle-même est partagée, ce qui est le cas lors de la mutualisation.

Cependant, du point de vue des différents travaux de recherches sur la mutualisation de SaaS, ce principe reste encore flou et présente de nombreux concepts et acronymes avec l'absence d'un consensus et d'une définition standard. Cela provoque la confusion, non seulement en raison des définitions différentes qui existent actuellement, mais également à cause des nombreuses hypothèses fondées sur les manières de le mettre en œuvre et le maintenir.

Dans ce chapitre, nous allons exposer les problèmes d'architecture logicielle lors de la mise en œuvre de la mutualisation, tout en soulignant les caractéristiques et les aspects qui font la différence entre ce principe et celui de SaaS à locataire unique (une instance d'application séparément déployée pour chaque client). Ainsi, ce chapitre est organisé de la manière suivante : la section 3.2 présente le modèle SaaS en détails et met particulièrement l'accent sur ses caractéristiques et ses clés de croissance ainsi que ses différents niveaux de maturité. La section 3.3 concentre sur le modèle SaaS assez mature caractérisé

principalement par l'adoption du principe de mutualisation. Les avantages et les défis de ce principe ainsi que son style architectural et le retour sur investissement de sa mise en œuvre sont tous explicités et détaillés. Enfin, la section 3.4 conclut le chapitre.

3.2 Les Applications SaaS

D'une manière informelle, SaaS peut être défini comme : « *un logiciel déployé sous forme d'un service, hébergé et accessible sur Internet* » [CC06]. Le principe de ce modèle est centré autour de l'idée de séparation entre la possession d'un logiciel et son utilisation. Le logiciel est possédé par un fournisseur qui délivre ses services aux clients à travers le web suivant une méthode de payement prenant en compte l'usage réel (voir figure 3.1). Ce modèle est souvent considéré comme une évolution de celui de l'ASP et est généralement proposé en tant qu'alternative aux applications traditionnelles sous licences (modèle on-premise), celles qui nécessitent une installation du logiciel chez le client et ainsi des investissements importants en termes de matériels et de ressources physiques. Techniquement, le modèle SaaS est le résultat d'évolutions technologiques et architecturales. Sur le plan technologique, nous trouvons le développement des réseaux permettant d'accéder à des ressources distantes à moindre coût et des techniques de virtualisation d'infrastructure. Un élément clé pour le développement du SaaS est l'apparition des interfaces utilisateurs beaucoup plus fonctionnelles (les RIA) qui permettent d'obtenir sur une interface Web une ergonomie équivalente voire meilleur par rapport à celle des clients lourds des applications classiques hébergées au sein de l'entreprise. Le concept de SaaS s'inscrit également dans les démarches de l'AOS qui tendent à simplifier de grosses applications monolithiques en services élémentaires pouvant être combinés et réutilisés pour composer des applications répondant aux besoins des processus métiers de clients.

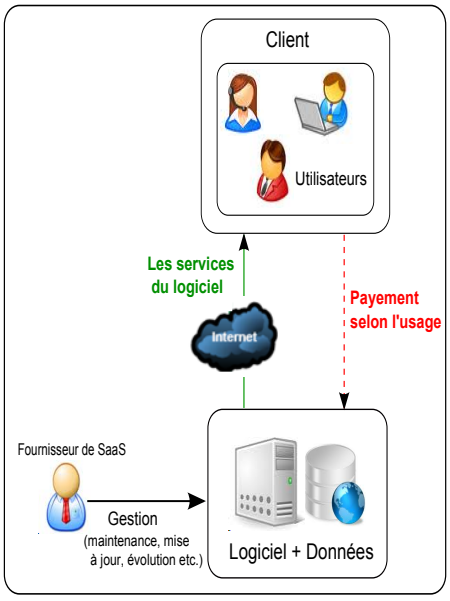


Figure 3.1 – Le modèle de fourniture SaaS

3.2.1 Les caractéristiques et les facteurs de croissance

Les logiciels SaaS fournissent généralement des solutions métiers complexes, inscrites dans des domaines d'applications différents(gestion du projet, gestion de relations clients, santé, éducation, nutrition etc.) [FG+09]. Bien que ces applications ne partagent pas les mêmes critères tant au niveau architectural qu'au niveau du modèle économique, les caractéristiques ci-dessous forment une base commune entre elles :

- **Configuration et personnalisation :** les applications SaaS ciblent potentiellement un nombre important de clients, dont les besoins métiers varient selon des différents facteurs, liés principalement à leurs positionnements géographiques et à leurs échelles industrielles. Par conséquent, les applications SaaS doivent être adaptées de manière à servir effectivement un client spécifique. Les approches largement utilisées pour atteindre cet objectif passent à travers la configuration et/ou la personnalisation, selon la stratégie adoptée pour le déploiement de l'application [SZG+08].
- **Accélération de livraison des nouvelles fonctionnalités :** les applications SaaS, par nature, ne sont pas conçues pour un ensemble des clients prédéfinis. Elles sont plutôt destinées à servir de nombreux clients, potentiellement inconnus. Ainsi, les fournisseurs de SaaS doivent continuer à améliorer les fonctionnalités et la qualité de leurs logiciels en permanence, afin de pouvoir attirer autant de clients que possible. Pour cette raison, ces logiciels sont souvent mis à jour plus fréquemment que les logiciels traditionnels, dans de nombreux cas sur une base hebdomadaire ou mensuelle. Ce rythme accéléré de livraison des nouvelles fonctionnalités est facilité grâce à plusieurs facteurs :
  1. L'application est centralisée, donc de nouvelles versions peuvent être rapidement mises en place pour le client.
  2. L'application est basée sur une pile unique de matériels, infrastructure et système d'exploitation, ce qui rend les tests de développement plus rapide.
  3. L'application est administrée par le fournisseur. Il pourra avoir accès aux données des utilisateurs et leurs comportements (généralement via des outils de reporting) ce qui accélère la détection des erreurs et les fonctionnalités à améliorer.
- **Des protocoles d'intégration ouverts :** les applications SaaS offrent des protocoles d'intégration et des interfaces de programmation (Application Programming Interface ou API) accessibles sur le web, pour pouvoir établir un lien avec les systèmes internes de l'entreprise cliente. Typiquement, ce sont des protocoles basés sur les technologies suivantes : REST (Representational State Transfer) [FT02], SOAP (Simple Object Access Protocol) [SSS00] et JSON (Javascript Object Notation) [Cro06].
- **Des dimensions collaboratives et sociales :** de nombreuses applications SaaS offrent des fonctionnalités qui permettent aux utilisateurs de collaborer et de partager des informations, fonctionnalités principalement inspirées par le succès des réseaux sociaux et d'autres fonctionnalités du web 2.0. Plusieurs applications de gestion de projets fournies actuellement suivant le modèle SaaS permettent aux utilisateurs de commenter les tâches et de partager des documents, ainsi que de voter et de proposer de nouvelles idées. Ces fonctionnalités de collaboration sont uniquement possible à intégrer dans un logiciel centralisé, celui du modèle SaaS.

Ces caractéristiques ont largement contribué à la croissance du modèle SaaS au cours de ces dernières années. Ainsi, les tendances du marché pour l'avenir continuent à être prometteuses. Selon un rapport récent d'IDC (International Data Corporation) [FRA12], le marché du SaaS a atteint 40 milliards de $ en revenus en 2011 et il devrait atteindre 100 milliards de $ en 2016, dont environ 40% de création de nouveaux logiciels dans le monde suivront les principes de ce modèle.

Les *facteurs de croissance* et de l'évolution de ce marché doivent aussi leur succès aux bénéfices induits par le modèle SaaS sur plusieurs niveaux [TZ+10, BOU09] :

- **Niveau économique :** la réduction des coûts offerte par le modèle SaaS est le moteur premier de l'intérêt pour son utilisation. Ceci est principalement dû au facteur d'échelle entre une application gérée par l'entreprise et hébergée sur sa propre infrastructure et la même application hébergée pour de multiples clients par un fournisseur spécialisé sur une infrastructure du cloud « industrialisée ». Ces bénéfices économiques consistent principalement en :
  1. *La réduction des besoins de trésorerie* : une application sous forme de licence hébergée chez le client nécessite des investissements initiaux qui consomment une trésorerie importante avant la mise en production. Par contre, le mode SaaS, et ce comme tous les autres services du cloud, dispose d'un nouveau modèle de facturation selon l'usage réel. Cette facturation est généralement faite sur une base d'abonnements mensuels. Cela apporte aux fournisseurs un flux de revenus continu, et aux clients un amortissement de l'investissement dans la durée de l'utilisation, au lieu de mobiliser de la trésorerie immédiate au moment de l'achat.
  2. *La moindre prise de risque au départ* : en mode SaaS, il est souvent facile de réaliser une phase pilote sans aucun engagement et sans rien dépenser en infrastructure. De plus, le paiement correspond à l'utilisation et le contrat peut être arrêté si la solution ne s'avère pas satisfaisante : le prise de risque est par conséquent faible, alors que le déploiement d'une infrastructure interne et d'une application sous licence achetée intégralement au début du projet reste beaucoup plus risqué.
- **Niveau fonctionnel :** les logiciels SaaS sont identifiés comme étant des technologies grands publics et sont généralement réalisés en collaboration avec des experts métiers dans le domaine de l'application traitée. Cela mène l'application à une meilleur conception et à un respect de normes et de réglementations du domaine, qui permet à l'entreprise cliente d'entrer dans un mode d'optimisation, sécurisation et standardisation de ses processus et ses méthodes de fonctionnement. Cela est généralement supporté par une facilité de prise en main au travers d'interfaces plus intuitives et ergonomiques par rapport aux interfaces traditionnelles en entreprise. Cela rend l'utilisation de logiciels SaaS relativement moins complexe et réduit ainsi les besoins de formation.
- **Niveau écologique :** la réduction de la consommation énergétique et celle de l'empreinte écologique des activités de l'entreprise est devenue une préoccupation majeure. Les applications SaaS sont intéressantes de ce point de vue parce qu'elles transfèrent aux fournisseurs la responsabilité écologique. Les fournisseurs s'appuient à leur tour sur des infrastructures virtuelles, centralisées et optimisées qui mobilisent unitairement moins de ressources que ne pourraient le faire les infrastructures des entreprises clientes individuellement.

3.2.2 Les niveaux de maturité

Quelque soit le domaine d'applications, les fournisseurs de SaaS visent l'augmentation rapide de leur clientèle, afin d'accélérer l'atteinte du seuil de rentabilité et de récompenser les investissements qu'ils ont engagé pour développer et commercialiser leurs applications. Pour anticiper un tel besoin, chaque fournisseur doit rapidement établir une politique de gestion de multiple clients, qui soit la plus adaptée avec son modèle économique et ses intérêts à long terme. Les travaux de recherches effectués par la communauté de SaaS identifient *quatre politiques de gestion de multiple clients* dont chacune est classée comme appartenant à un niveau de maturité de SaaS différent [CC06, KNL08] (voir figure 3.2). Ces niveaux de maturité indiquent les façons dont un fournisseur de SaaS parviendra à supporter efficacement

et d'une manière rentable, un nombre important de clients [LZL+10]. Les attributs clés que doit avoir l'application pour acquérir cette maturité sont les suivants [LGX09, KMY+10] :

- **La configuration :** cet attribut de l'application indique que son adaptation aux besoins spécifiques de clients se fait à travers des choix de configurations, et non à travers la modification directe du code.
- **La mutualisation :** cet attribut de SaaS signifie qu'une instance unique de l'application est utilisée pour servir de multiples clients en même temps.
- **Le passage à l'échelle :** cet attribut de SaaS désigne sa capacité à délivrer ces fonctionnalités avec un temps de réponse constant et des coûts opérationnels « acceptables », même si le nombre de clients ou de données augmente de manière importante.

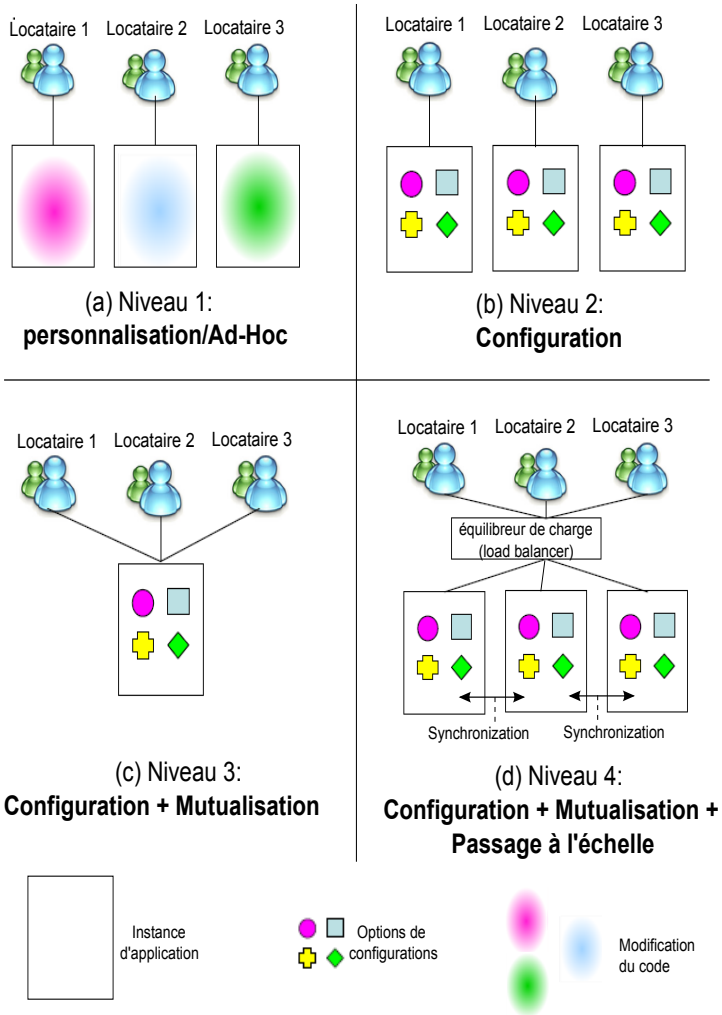


Figure 3.2 – Les niveaux de maturité de SaaS

Chaque niveau de maturité de SaaS se distingue du précédent par l'acquisition progressive d'un nouveau attribut qui apparaît soit au niveau de l'architecture de l'application, soit au niveau de son organisation infrastructurelle. Par la suite, nous allons présenter chaque niveau de maturité de SaaS.

Niveau 1 : Personnalisation/Ad-Hoc : dans ce niveau (voir figure 3.2 (a)), chaque client a sa propre version personnalisée de l'application déployée séparément sur un serveur (physique ou virtuel) dédié. Il n'y a pas de partage entre les clients à ce niveau, et chaque instance de l'application est spécialement personnalisée pour un client à travers une modification directe du code. Ce niveau apporte le plus d'agilité et ainsi le plus simple à atteindre. Cependant, les coûts opérationnels de l'application à ce niveau sont assez élevés. Ce niveau de maturité de SaaS correspond dans la majorité de ces caractéristiques (sauf celle de facturation selon l'usage) au modèle ASP [KNL08].

Niveau 2 : Configuration : ce niveau ajoute de la flexibilité au logiciel par rapport au niveau précédent. Chaque client aura sa propre instance adaptée à travers la précision des choix de configuration parmi les diverses options de configuration fournies par la même version du logiciel [LGX09]. Cette possibilité de configuration permet à chaque client d'indiquer (d'une manière automatisée ou semi-automatisée), la façon selon laquelle l'application doit se comporter pour ses utilisateurs. Au contraire du niveau précédent où chaque version du logiciel est spécialement développée et modifiée pour chaque client, à ce niveau, une seule version du logiciel est conçue. La gestion et la maintenance seront finalement plus faciles, et le développeur n'a plus besoin de maintenir autant de versions que de clients. Cependant, le logiciel sera plus sophistiqué et complexe à développer à cause de la gestion des configurations. Aussi, les tâches d'évolution et de mise à jour quotidiennes (de la partie commune du logiciel) doivent être individualisées, car chaque client exécute une instance séparée. La figure 3.2 (b) illustre ce niveau.

Niveau 3 : Configuration et mutualisation : ce niveau ajoute la mutualisation au niveau précédent (voir figure 3.2 (c)). Tous les clients (également appelés locataires dans ce niveau) exécutent la même instance du logiciel, alors que leurs configurations sont isolées et appliquées d'une manière dynamique (lors de l'exécution du logiciel), comme les identités de locataires ne sont pas connues à l'avance. Ceci diffère du niveau précédent où chaque instance est globalement configurée (avant l'exécution) et se comporte identiquement pour tous les utilisateurs. La mutualisation réduit considérablement les coûts de maintenance et de mise à jour de l'application car ces tâches ne sont plus individualisées mais gérées comme étant pour un seul locataire. Cependant, ce niveau de maturité peut devenir très complexes à gérer à cause de la nécessité d'une gestion dynamique des configurations et d'une isolation stricte des données de locataires. Un autre problème des logiciels SaaS appartenant à ce niveau est l'inefficacité d'un seul serveur (sur lequel l'application est déployée) à supporter la lourdeur de charge imposée par un nombre très important de locataires. Un passage à l'échelle verticale (scale up) pourra être adopté à travers la redimensionnement du serveur en lui ajoutant plus de ressources (mémoire, bande passante , CPU, disques, etc.), à chaque fois une augmentation de charge est prévue. L'inconvénient de cette approche réside dans le cas où le serveur devient trop sous-chargé (période de vacances, désinscription de certains locataires, etc.), les ressources ajoutées consomment de l'énergie inutilement et provoque un rendement décroissant ce qui oblige à augmenter les frais d'utilisation de l'application. De même, l'ajout et le retrait de ressources verticalement ne peut pas prendre effet sans l'arrêt et le redémarrage du serveur (confère section 2.2.2), ce qui impact considérablement la disponibilité de l'application et ainsi la qualité du service.

Niveau 4 : Configuration, mutualisation et passage à l'échelle : ce niveau ajoute la possibilité de passer à l'échelle par rapport au niveau précédent. Il propose de déployer plusieurs instances de l'ap-

plication mutualisée sur des serveurs à puissances identiques (il faut noter que l'utilisation d'un nombre *n* de serveurs à une puissance moyenne ne coûtera pas plus que l'utilisation d'un seul serveur *n* fois plus puissant) [AFG+10]. Pour passer à l'échelle, les locataires n'auront pas à interagir avec les serveurs directement, mais avec un équilibreur de charge (*Load Balancer*, voir figure 3.2 (d)) qui surveille en permanence la charge de travail de chaque serveur, et distribue toute nouvelle demande d'un locataire au serveur le moins chargé. Le nombre de serveurs en cours d'exécution peut être augmenté d'une manière proportionnelle à l'augmentation de nombre de locataires et diminué dans le cas contraire. De plus, le fournisseur peut déterminer la capacité exacte dont il aura besoin à une période donnée sans avoir à sur-estimer (ou sous-estimer) cette capacité, menant ainsi à une performance optimale. Ce niveau sera bien évidement plus complexe à réaliser que celui du niveau précédent, étant donné qu'un système d'équilibrage de charge en temps-réel doit être intégré, et qu'une synchronisation entre les différentes instances du logiciel doit être maintenue en permanence. Aussi, la maintenance sera un peu plus compliqué à cause de l'augmentation de nombre des instances. Pourtant, ce niveau de maturité de SaaS peut potentiellement offrir une meilleure qualité de service et il peut ajuster les ressources infrastructurelles en fonction de la variation de charge réelle.

En conclusion, le niveau de maturité auquel une application SaaS doit appartenir, dépend fortement de la stratégie envisagée par le fournisseur de cette application et ainsi de ses intérêts à long terme. Bien évidement, une application ne possédant pas tout les attributs (configuration, mutualisation et paasage à l'échelle) qui mènent un niveau de maturité maximale, est toujours capable de répondre aux exigences du fournisseur. En revanche, l'abandon d'un de ces attributs (pour des raison du coût ou de complexité) limitera plus rapidement le système en termes de sa capacité à gérer efficacement un nombre important de clients. Les auteurs dans [MUTL09] montrent une possibilité de combinaison de ces différents niveaux dans une seule application pour trouver un bon équilibre entre les bénéfices et la complexité. Toutefois, ces derniers admettent que le choix idéal sera toujours celui d'une application SaaS assez mature (niveau 3 ou 4). Il soulignent également les avantages de ce choix ainsi que les défis à relever en termes de conception et de développement. Dans la section suivante nous allons nous concentrer sur ce choix et détailler ses défis, ainsi que le style architectural pour saisir ses concepts architecturaux et le retour sur investissement de la conformité de l'architecture de l'application avec ce style.

3.3 La Mutualisation de SaaS

La mutualisation de SaaS est un principe d'architecture logicielle relativement nouveau [BZ10b] qui consiste à héberger de multiples clients sur une instance unique d'application et de base de données. Par une instance unique nous entendons qu'un seul code est exécuté pour répondre à tous les clients et que les ressources réservées à cette exécution sont toutes partagées. Suivant ce principe, le fournisseur de SaaS gère tous les locataires en même temps tout en installant pour une seule fois la pile matérielle et logicielle nécessaire. Toutefois, le fait de partager un seul code d'application entre plusieurs locataires présente quelques défis à relever. En effet, l'application doit être conçue d'une manière à être facilement configurable lors de l'exécution pour pouvoir l'adapter dynamiquement aux besoins spécifiques de chaque locataire, tout en isolant leurs configurations et leurs données. Ces défis créent une complexité supplémentaire de conception et de développement qui nécessitent la prise des décisions architecturales pour éviter les risques d'échec [BZ10b] et imposent des contraintes importantes sur leur processus de développement qui impliquent des enjeux nécessitant une expertise de haut niveau [Koz11]. En revanche,

les niveaux 1 et 2 de maturité de SaaS ne traitent pas les mêmes défis car ils utilisent une approche multi-instances (une instance d'application séparément déployée pour chaque client).

Cette section concerne principalement les applications qui suivent les niveaux 3 et 4 de maturité de SaaS, et nos travaux porte également sur ce type d'applications. Mis à part l'organisation infrastructurelle de niveau 4 permettant de passer à une plus grande échelle, ces deux niveaux de maturité gèrent les mêmes défis conceptuels et techniques, et par conséquent, nous n'en faisons pas la distinction dans la suite de ce manuscrit.

3.3.1 Généralités et définitions

Dans une application SaaS mutualisée, un locataire est un groupe d'utilisateurs partageant la même vue sur l'application qu'ils utilisent. Cette vue inclut les données auxquelles ils accèdent, les configurations disponibles et les propriétés fonctionnelles et non-fonctionnelles supportées. Habituellement, les groupes d'utilisateurs sont des membres de différentes entités juridiques qui dans la plupart des cas, ont des intérêts communs ou similaires à utiliser l'application. Ainsi, ils décident d'en dévernir des locataires. Cela impose des restrictions, telles que l'isolation de leurs données et la prise en considération de leur besoins spécifiques fonctionnels et non-fonctionnels. Pour schématiser ce principe, une analogie est souvent faite avec la relation entre les bailleurs et les locataires. La comparaison correspond à la création d'un immeuble de location où les différents locataires acceptent de partager certaines ressources pour réduire les frais de location (eau, électricité, accès à Internet, gardiennage, buanderie, piscine, etc.), mais à condition d'assurer la protection de leur droit à la vie privée à travers un certain niveau d'isolement.

Toutefois, le principe de mutualisation de SaaS reste encore flou. Alors que plusieurs travaux ont tentés de le définir d'une façon non-ambigüe [BZ10b, WB09, KMK12], nous pensons que ces définitions nécessitent encore des améliorations. À titre d'exemple, les auteurs dans [KMK12] définissent une application SaaS mutualisée de la manière suivante :

« La mutualisation de SaaS est une approche qui consiste à faire partager une instance unique d'application entre plusieurs locataires en fournissant à chacun entre eux une partie dédié de l'instance, qui est isolée des autres parties appartenant à d'autres locataires vis-à-vis de la performance et de la confidentialité des données. »

Cette définition fait seulement référence aux aspects de performance et de confidentialité des données, et ne précise pas la nécessité de configurer l'application pour répondre aux besoins spécifiques de locataires. Ainsi, nous la considérons comme une définition incomplète.

D'ailleurs, les auteurs dans [BZ10b] évoquent le besoin de configuration en proposant la définition suivante :

« Une application SaaS mutualisée permet aux locataires de partager les mêmes ressources matérielles en leur offrant une seule instance partagée d'application et de base de données, et leur permettant de la configurer pour répondre à leurs besoins comme si elle s'exécute sur un environnement dédié. »

Bien que cette définition est plus complète que la précédente concernant la nature de la mutualisation et la nécessité de pouvoir configurer l'application, elle présente à notre sens la faiblesse suivante : elle néglige le fait que l'application peut être dupliquée sur plusieurs serveurs afin de réagir efficacement contre les variations des charges telle que le niveau 4 de maturité de SaaS le propose.

En effet, nous croyons qu'une définition complète et non-ambigüe de la mutualisation de SaaS devra unifier les deux derniers niveaux de maturité de SaaS pour pouvoir couvrir autant de situations que possibles. De ce fait, nous proposons notre définition de ce type d'applications comme suit :

Définition 3.1. :
Une application SaaS mutualisée consiste à faire partager ses ressources entre plusieurs locataires, dans le but de réduire sa complexité opérationnelle et de tirer pleinement parti des économies d'échelle. L'application est potentiellement déployée sur plusieurs serveurs afin de réagir efficacement contre les variations de charges et d'optimiser l'utilisation des ressources infrastructurelles. Un locataire de cette application peut la configurer et la re-configurer à n'importe quel moment pour l'adapter à ses besoins. Sa configuration et ainsi ses données sont entièrement isolées de tout impact par d'autres locataires.

Toutefois, et quelque soit le degré de précision de la définition, le principe de mutualisation de SaaS commence à gagner du terrain dans le monde de l'informatique industrielle aussi bien que dans le monde académique, et s'il est mis en œuvre avec succès, il peut fournir plusieurs bénéfices aux niveaux économiques et fonctionnels. Cela le rend de plus en plus intéressant pour les plus grands acteurs du cloud à travers le monde et fondamental pour la continuité et la croissance du marché de SaaS [CWZ10]. Par la suite, nous allons cité les principaux bénéfices de ce principe.

3.3.2 Les bénéfices

Bien évidement, les applications mutualisées héritent de touts les bénéfices du SaaS déjà mentionnés (voir section 3.2.1). Elles présentent de surcroît les principaux bénéfices suivants :

Automatisation de la mise en place de locataires : la manière dont laquelle une application mutualisée est conçue signifie que le logiciel et la base de donnée sont déjà mis en place pour accueillir plusieurs locataires. Cela offre la possibilité d'automatiser leur création et de faciliter la proposition de l'application à titre d'essai par rapport aux applications SaaS à locataire unique où la mise en place d'un nouveau client nécessite l'installation et l'exécution d'une nouvelle instance de l'application : une tâche chronophage.

Réduction de la complexité opérationnelle : la mutualisation d'une application permet à son fournisseur de réduire la complexité de gestion par rapport aux applications à locataire unique. Elle permet une maintenance plus rapide et un déploiement plus facile des nouvelles fonctionnalités pour des locataires en particulier ou pour l'ensemble de la population de locataires à la fois. Elle permet aussi de récupérer plus rapidement les retours de leur expérience et de les analyser pour améliorer l'application sur une base régulière. L'absence de ces bénéfices dans les applications SaaS moins matures (à locataire unique) peut devenir un réel facteur limitant, où les différentes instances de l'application et de bases de données doivent être maintenues et mises à jour séparément, augmentant significativement la charge de travail du fournisseur.

Optimisation des ressources infrastructurelles : La mutualisation d'une application permet de faire évoluer la capacité de ses ressources infrastructurelles d'une manière proportionnelle à la variation de charge réelle. Son objectif est de garder le niveau de performance requis et de tirer profit de la diminution temporelle que subit fréquemment la charge de travail de certains locataires pour récompenser l'augmentation des autres. Pour bien expliquer cette possibilité, prenons l'exemple présenté dans la figure 3.3. Cet exemple montre l'impact de la variation de charge sur les deux types d'applications SaaS : (a) à locataire unique et (b) mutualisée. Dans la figure 3.3 (a), la variation de charge (représenté par l'augmentation du *Locataire 1* et la diminution du *Locataire 4*) provoque un déséquilibre dans la capacité infrastructurelle du système, ce qui exige l'engagement des frais infrastructurelles supplémentaires soit pour augmenter la capacité du *Serveur 1*, soit pour faire déplacer le *Locataire 1* vers un nouveau serveur. Cependant, les applications mutualisées n'exigent

aucune modification infrastructurelle dans cette situation spécifique, car la variation de la charge se repartit dynamiquement sur l'ensemble des serveurs en toute transparence (voir figure 3.3 (b))).

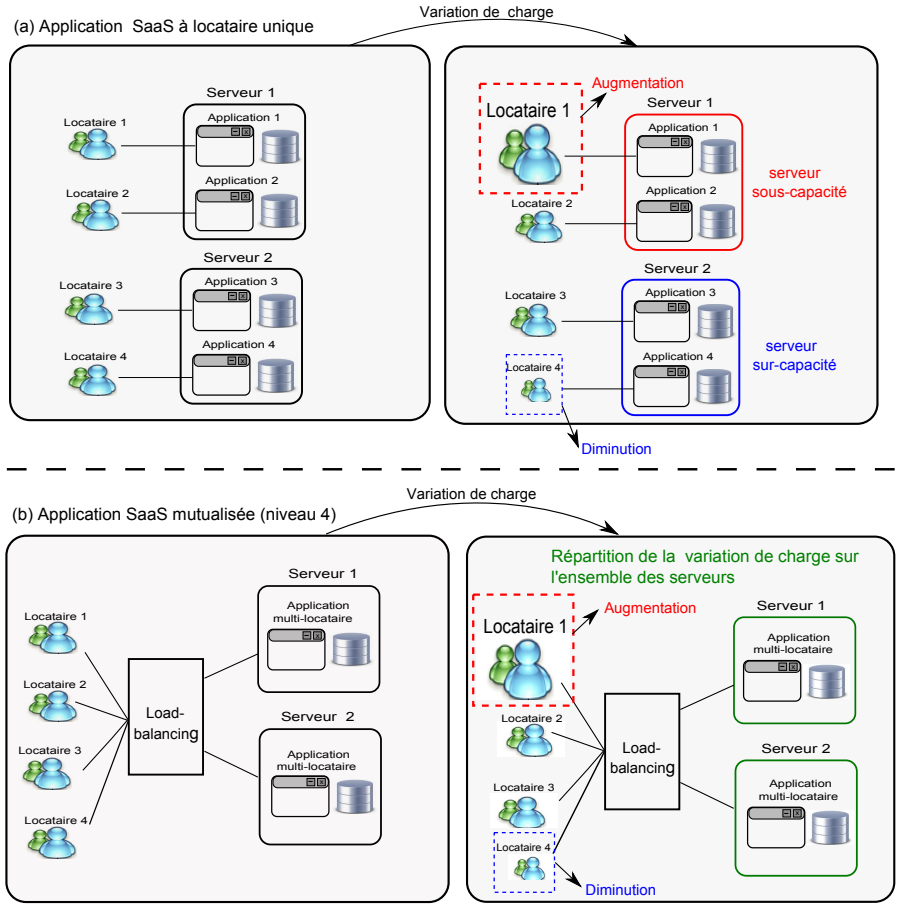


Figure 3.3 – La différence de réaction face à la variation de charge d'une application SaaS à locataire unique et une application mutualisée (niveau 4 de maturité de SaaS)

À une très grande échelle (des milliers de locataires), la variation de charge devient assez fréquente et l'ajustement dynamique de la capacité en fonction de cette charge évitera des coûts infrastructurelles non-négligeables. Le passage à cette échelle se fait toujours d'une façon horizontale (scale out, confère section 2.2.2), par l'ajout progressif des nouveaux serveurs virtuels ou physiques ainsi que par le retrait lors du passage à une échelle plus petite.

Ces bénéfices immédiats de la mutualisation de SaaS auront comme effet de considérablement baisser les coûts d'utilisation de l'application par les locataires à un niveau de prix plus bas que les concurrents [BZ10b]. Cela rend ce principe très attractif pour les fournisseurs de SaaS qui visent à commercialiser leurs solutions dans des segments de marché toujours considérés comme inaccessibles, les petites-moyennes entreprises par exemple, comme elles ont souvent des ressources financières assez limitées et ne peuvent pas supporter le coût élevé d'une application dédiée.

3.3.3 Les défis

Malheureusement, la mutualisation de SaaS présente une série de défis conceptuels et techniques à relever avant de pouvoir tirer profit de tous ses bénéfices. Même si certains de ces défis existent pour les applications SaaS à locataire unique, ils apparaissent sous une forme différente et sont plus complexe à résoudre. Dans cette section, nous allons cité les principaux défis [BZ10b, KMK12, GSH+07, PLL10] :

La gestion de la variabilité : pour une application mutualisée, la capacité de maintenir des configurations et des adaptations dynamiques de l'application pour satisfaire les exigences spécifiques des locataires est un facteur de facilitation essentiel. Cette gestion a été largement traitée dans l'état de l'art en tant que problématique incontournable à surmonter absolument dans ce contexte [ML08, MMLP09, KJ11, SCG+12]. Sa complexité réside dans le fait qu'elle affecte tous les aspects de l'application, de l'aspect fonctionnel tel que l'apparence et le processus métier, jusqu'à l'aspect non-fonctionnel concernant par exemple la qualité de service.

L'isolation des données de locataires : dans une application mutualisée, les données de tous les locataires sont consolidées dans une instance de base de données unique. De ce fait, il est primordial de mettre en place un système d'isolation de données capable de garantir qu'un locataire n'aura jamais le privilège d'accéder aux données d'autres locataires (potentiellement des concurrents) sauf si ce privilège a été explicitement accordé [WGG+08].

Le contrôle d'accès aux ressources de l'application : une application mutualisée exige l'exclusivité d'accéder à une partie de ses ressources par un ensemble de locataires bien précis. Cela est généralement lié au modèle économique de SaaS basé sur le principe de facturation selon l'usage, celui qui oblige, dans un environnement mutualisé, à avoir une même ressource accessible pour un locataire et interdite pour un autre. Les applications SaaS moins matures ne supportent pas cette exclusivité étant basées sur le principe de locataire unique. Cette exigence des applications mutualisées nécessite une extension des mécanismes de contrôle d'accès existants (RBAC [FK09], TMAC [GMPT01], etc.) afin de prendre en considération le concept de locataire.

La maintenance : une application à locataire unique peut avoir plusieurs instances en cours d'exécution et elles peuvent être toutes différentes les unes des autres en raison de leur personnalisation. Dans la mutualisation, ces différences n'existent plus puisqu'une seule instance d'application est configurable au moment de l'exécution (grâce à la gestion dynamique de la variabilité). S'il est évident que la mutualisation peut apporter des avantages sérieux pour le déploiement en réduisant au minimum le nombre d'instances d'application et de base de données, la maintenance de l'application proprement dite est plus délicate. En général, l'introduction de la mutualisation dans un système logiciel augmente sa complexité, ce qui est susceptible d'affecter le processus de maintenance traditionnel [BZ10b].

Toutefois, les différentes manières de relever ces défis ne sont pas encore bien compris et documentés. Cependant, il est difficile pour les développeurs de concevoir et de développer une application mutualisée, car les concepts architecturaux et les décisions nécessaires à la conception ainsi que leurs compromis nécessitent une grande expertise. Les tentatives pour fournir une documentation structurée des concepts architecturaux de ce type d'applications se concentrent principalement soit sur un aspect particulier (par exemple la couche de base de données [AGJ+08, FDFI10]) soit sur certaines technologies et environnement de développement bien précis (.NET [CCW06], IBM/Java [tMT10, OGTP09]). Ainsi, ces architectures documentés sont difficiles à transférer à d'autres systèmes. Par la suite, nous allons expliquer le style architectural de ce type d'application comme étant une perspective plus abstraite

et indépendante de la technologie, et nécessite la prise de décisions quant aux choix liés à la conception impliqué et les compromis d'architecture lors de la mise en œuvre de la mutualisation de SaaS.

3.3.4 Le style architectural

Selon [FT02], un style architectural est un ensemble de contraintes coordonnées, qui limitent les rôles des éléments architecturaux et les relations autorisées entre eux dans toute architecture qui est conforme au style. Comparé avec les patrons de conception qui sont plus descriptifs et résument des solutions éprouvées, les styles architecturaux ont un caractère plus normatif. Ils posent des contraintes sur l'espace de conception d'une architecture logicielle pour mener au mieux sa réalisation et atteindre facilement ses propriétés désirées. Conduit par ce principe, les auteurs dans [Koz10, Koz11] proposent un style architectural qui défini les éléments fondamentaux de toutes architectures des applications SaaS mutualisées (voir figure 3.4). La majorité des éléments architecturaux figurant dans ce style (composants, connecteurs, données, etc.) présentent une extension de ceux qui existent dans des styles architecturaux classiques (Client/Serveur, pipe-and-filter, multi-couches, REST, etc.). Toutefois, la nouveauté de ce style consiste à introduire un élément architectural additionnel : celui de la gestion de *méta-données*. Le rôle de cet élément est de fournir les informations nécessaires à l'adaptation dynamique de l'application dans ses différents aspects (interfaces graphiques, processus métiers, services, etc.). En effet, les méta-données sont des données non-exécutables qui ont pour rôle de définir la manière dont l'application doit se comporter dans son ensemble ou dans une partie de ses fonctionnalités. Cela évitera le codage en dur de ses fonctionnalités *variables* et permettra de les adapter aux différentes exigences en temps réel. L'utilisation des méta-données dans ce contexte est justifiée par la nature dynamique de l'environnement mutualisé, difficile à supporter par une application compilée entièrement de façon statique. Le même raisonnement est valable pour les données de l'application : les méta-données permettront de définir un schéma de base de données adaptable et flexible [LHX12]. L'utilisation de méta-données a été également confirmée par de nombreux travaux de recherches [CC06, KNL08, NG09], comme étant le choix idéal pour adresser la gestion dynamique des variations et des configurations. Citons le travail présenté dans [WB09] qui préconise l'utilisation des méta-données dans le développement de la plateforme mutualisée (Force.com) de l'entreprise SalesForce.

Selon ce style, le gestionnaire de méta-données est responsable de la gestion des adaptations et des configurations nécessaires. Il est responsable de récupérer dans un premier temps les méta-données spécifiques aux locataires et ensuite, soit de les diriger vers l'application pour qu'elle adapte elle-même en traitant ces informations, soit de générer à la volée du code spécifique. Il peut y avoir des différentes portées de configuration de l'application par le locataire de sorte que les différentes unités d'une entreprise peuvent être traités différemment. Selon ce style, une application mutualisée doit permettre aux locataires de la configurer à n'importe quel moment et doit ainsi fournir les outils nécessaires pour cela.

Cependant, l'utilisation d'un nombre important de méta-données signifie une grande divergence entre les exigences de locataires et un temps de configuration assez considérable qui contredit le principe de mutualisation et de partage de l'application. De ce fait, l'architecte doit définir le niveau exact de méta-données que l'application doit soutenir. Plus de méta-données implique plus d'endroits variables et donc plus de complexité de développement. Ainsi, nous croyons que les applications qui exigent une variabilité trop élevée ne sont pas bien adaptées à une approche mutualisée au niveau applicatif du cloud, mais plutôt au niveau de la plateforme et de l'infrastructure.

Toutefois, le style architectural présenté propose une solution abstraite pour relever les défis de la mutualisation, qui nécessite d'être concrétisée. En fonction de l'application à offrir et selon l'option concrète utilisée, différentes mesures avec des complexités variées sont à gérer. Cela nécessite l'engage-

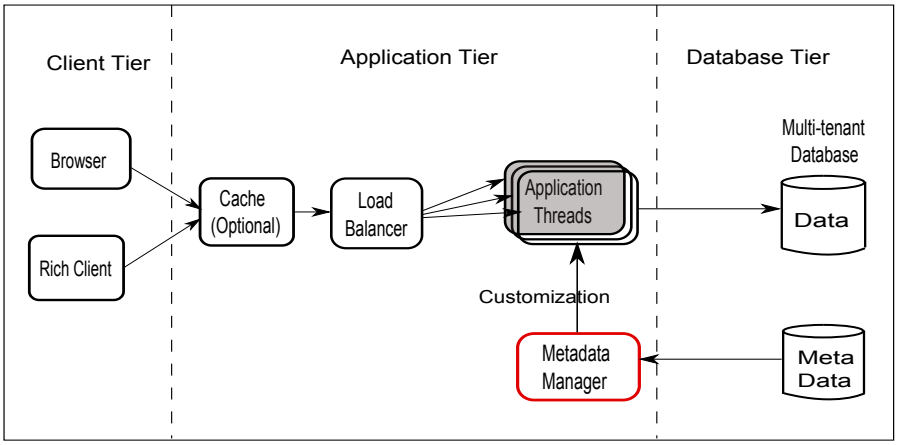


Figure 3.4 – Un style architectural pour les applications SaaS mutualisées [Koz10]

ment dans des nouveaux développement et des investissements sur la mutualisation de SaaS pour aboutir à une solution simple à intégrer et facile à administrer. Le seuil de rentabilité de cet investissement est calculé en fonction de deux facteurs principaux : le coût initial de développement et les coûts récurrents qu'il implique. En conséquence, chaque activité effectuée pour réduire les coûts récurrents, tôt ou tard va être amortie. Par la suite, nous allons détailler une étude simple que nous avons effectué pour évaluer le retour sur investissement (RSI) dans le développement d'une application SaaS à locataire unique, et la même application développée en mode mutualisé. Le but de cette étude est de mieux comprendre les enjeux de la mutualisation de SaaS et la raison pour laquelle ce principe est intéressant du point de vue économique.

3.3.5 Le retour sur investissement

Pour évaluer le RSI d'une application SaaS mutualisée, nous avons identifié un ensemble de paramètres pour établir les équations de gain et celles du coût d'investissement. En particulier nous voulons mesurer les surcoûts introduits par la mutualisation et la manière dans laquelle ils seront amorties. Les équations que nous proposons prennent en considération l'axe du temps à partir du moment de la mise de l'application sur le marché et sur les mois qui suivent. L'équation 3.1 permet de calculer le gain actuel d'une application SaaS. Pour simplifier le raisonnement, nous ne différencions pas à ce niveau entre les deux types d'application (à locataire unique et mutualisée) et nous considérons que le modèle de facturation adopté est le même. Pour une application SaaS à locataire unique, un locataire signifie un client qui utilise une application dédiée.

$$Gain(mois) = \sum_{i=1}^{i=mois} NbLocataire(i) * F_L \tag{3.1}$$

$$NbLocataire(i) = (i * f_{ins}) - (i * f_{desins}) \tag{3.2}$$

* f_{ins} :la fréquence moyenne d'inscription de nouveaux locataires par mois.
* f_{desins} :la fréquence moyenne de désinscription de locataires par mois.
* F_L : la facturation moyenne selon l'usage mensuel de l'application par un locataire.

Dans cette exemple, nous fixons la valeur de f_{ins} à 2, la valeur de f_{desins} à 0,2 (une désinscription tous les 5 mois) et la valeur de F_L à 100 €. Ces valeurs sont estimées à partir de notre expérience dans la gestion de ce type d'applications.

L'équation 3.3 permet de calculer le coût d'investissement actuel pour une application SaaS à locataire unique.

$$Cout_{SaaS}(mois) = C_{Dev} + \sum_{i=1}^{i=mois} I_{SaaS}(i) + M_{SaaS} \tag{3.3}$$

$$I_{SaaS}(i) = (\frac{NbLocataire(i) * Charge_L}{Capacite_S} * F_S) \tag{3.4}$$

* C_{Dev} : le coût de développement de la partie métier de l'application.
* $I_{SaaS}(i)$: les dépenses totales en termes d'infrastructures pour une application SaaS à locataire unique au mois *i* de sa mise sur le marché (équation 3.4). La valeur de ces dépenses est calculée en fonction du nombre de serveurs installés pour exécuter tous les locataires multiplié par la facture mensuelle de l'utilisation de chaque serveur. Le nombre de serveurs est déduit en divisant la charge de travail totale des locataires exprimée en transaction par seconde divisée par le nombre moyen de transactions par seconde que supporte un serveur.
* $Charge_L$: la charge moyenne d'utilisation de l'application par un locataire exprimée en transaction par seconde (T/S).
* $Capacite_S$: le nombre moyen de transactions par seconde que supporte un serveur.
* F_S : la facture mensuelle d'utilisation d'un serveur à capacité moyenne chez un fournisseur IaaS.
* M_{SaaS} : le coût de maintenance de l'application par mois inclue les corrections des erreurs, les mises à jour, les développements spécifiques, etc.

Dans cet exemple, nous fixons la valeur de C_{Dev} à 100 K€, la valeur de $Charge_L$ à 50 T/S, la valeur de $Capacite_S$ à 500 T/S, la valeur de F_S à 120 € (selon les tarifs indiqués par notre fournisseur IaaS) et la valeur de M_{SaaS} à 1 K€ comme une estimation selon le nombre de développeurs engagés et le temps qu'ils passent sur les tâches de maintenance (sachant que cette valeur peut augmenter avec le nombre de locataires).

L'équation 3.5 permet de calculer le coût d'investissement actuel pour une application SaaS mutualisée.

$$Cout_{SaaSMut}(mois) = C_{Dev} + C_{Mut} + \sum_{i=1}^{i=mois} I_{SaaSMut}(i) + M_{SaaSMut} \tag{3.5}$$

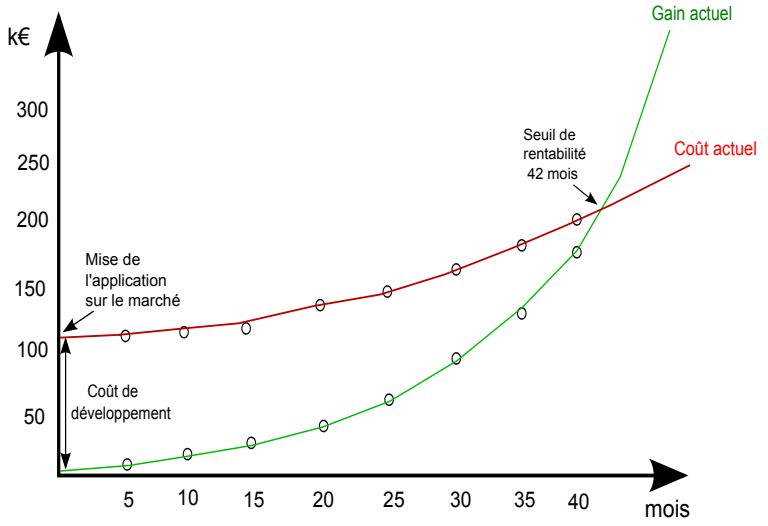
$$I_{SaaSMut}(i) = \alpha * I_{SaaS}(i) \; tel \; que \; 0,4 < \alpha < 0,8 \tag{3.6}$$

$$M_{SaaSMut} = \beta * M_{SaaS} \; tel \; que \; 0,4 < \beta < 0,8 \tag{3.7}$$

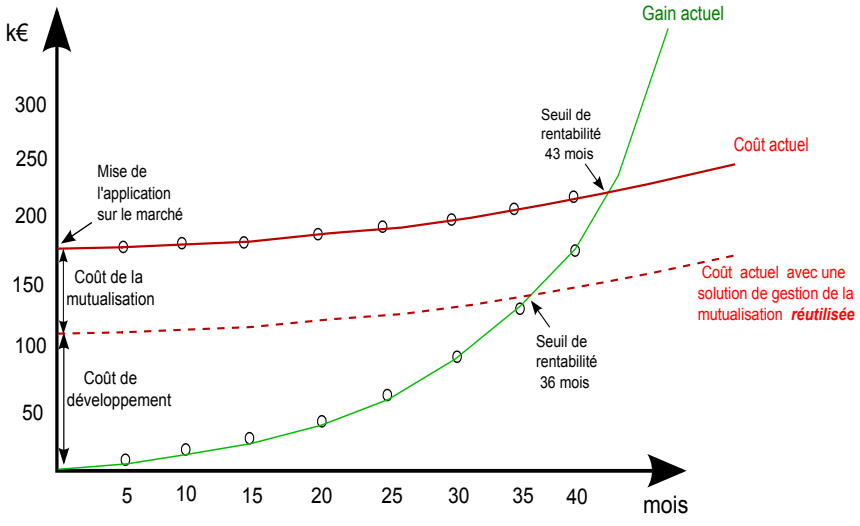
* C_{Mut} : le coût de développement d'une solution de gestion de la mutualisation et ces défis.
* $I_{SaaSMut}(i)$: les dépenses totales en termes d'infrastructure pour une application SaaS mutualisée au mois *i* de sa mise sur le marché.
* $M_{SaaSMut}$: le coût de maintenance de l'application par mois incluant les corrections des erreurs, les mises à jour, les développements spécifiques, etc.

Dans cet exemple, nous fixons la valeur de C_{Mut} à 75 K€. Les valeurs de $I_{SaaSMut}(i)$ et de $M_{SaaSMut}$ dépendent de l'efficacité et la maintenabilité de la solution de mutualisation. Plus la solution adoptée est

maintenable, plus les dépenses évitées sont importantes. La valeur de α et β sont fixées à 0,6. Ces valeurs doivent absolument être plus petites que 1 pour que la mutualisation puisse apporter ces bénéfices.



(a) RSI SaaS à locataire unique



(b) RSI SaaS mutualisée

Figure 3.5 – Le retour sur investissement pour une application SaaS à locataire unique et la même application développée en mode mutualisée

La figure 3.5 montre la courbe du gain par rapport à celle du coût d'investissement calculées en fonction des équations présentées. Nous pouvons remarquer que les deux types d'application ont presque le même seuil de rentabilité malgré les dépenses initiales beaucoup plus élevées pour une application SaaS mutualisée. Cela revient à la baisse considérable du coût de maintenance et de besoin en infrastructure apporté par la mutualisation, compensant d'un côté les dépenses initiales et augmentant de l'autre côté la

marge de bénéfice. Naturellement, la situation dans la pratique est beaucoup plus complexe que suggère la figure 3.5. Parmi d'autres considérations sont le temps consommé avant la mise de l'application sur le marché et l'évolution des coûts de développement et de maintenance en fonction de nombre de locataires. Néanmoins, cette figure illustre un compromis fondamental qui est souvent ignoré dans l'analyse de la mutualisation.

Un autre point considéré comme extrêmement important et même stratégique pour une entreprise qui s'investit dans la mutualisation de SaaS, celui de l'élaboration d'une solution de gestion qui soit facilement réutilisable dans de nouvelles applications. Cela évitera la duplication des efforts lors du prochain développement et permettra d'atteindre plus rapidement le seuil de rentabilité avec une marge de bénéfice beaucoup plus élevée (voir 3.5 (b), courbe en pointillés rouge). Dans nos travaux de recherche, nous avons tenté de trouver une solution de gestion de la mutualisation qui soit tout d'abord maintenable et efficace à administrer et ensuite facilement réutilisable et ce, indépendamment de l'architecture de l'application ou de sa technologie. Nous nous concentrons essentiellement dans cette thèse sur les deux premiers défis de la mutualisation : *la gestion dynamique de la variabilité* des besoins de locataires et l'*isolation stricte de leurs données*. Nous présentons dans la suite l'état de l'art effectué pour poser les éléments nécessaires à l'analyse de chaque défi, et les approches suivies pour fournir une solution de gestion qui répond aux besoins identifiés.

3.4 Conclusion

Dans ce chapitre, nous avons principalement analysé le modèle SaaS et présenté ses différents niveaux de maturité. Le principe de mutualisation qui caractérise les applications SaaS suffisamment matures a été également analysé et détaillé en tant que principale sujet de cette thèse. L'intérêt d'adopter ce principe est justifié par la solution qu'il propose pour réduire le nombre d'instances d'application nécessaires pour exécuter et gérer l'ensemble de locataires. Nous avons également montré que le déploiement et la mise à jour de ce type d'applications deviennent beaucoup plus facile et moins coûteux, en raison du fait qu'un nombre plus restreint d'instances doivent être gérées.

Cependant, nous avons évoqué que la mutualisation de SaaS fait état d'une hausse de la complexité du code à cause des défis qu'elle implique tels que la gestion dynamique de la variabilité des besoins de locataires et l'isolation de leurs données. Ces défis nécessitent l'implémentation de nouvelles fonctionnalités dans l'architecture de l'application, ce qui en soi augmente la complexité du code et donc rend la maintenance plus difficile. Ainsi, nous insistons sur le fait que l'élément crucial pour la mise en œuvre *réussie* de ce principe réside dans la qualité et la propreté de solution de gestion. Une telle solution doit être facile à intégrer et efficace à administrer. Étant donné que la mutualisation est un principe d'architecture logicielle relativement nouveau, en particulier dans le monde du génie logiciel, très peu de recherches ont été effectuées sur ce sujet. Nous avons essayé de fournir à travers ce chapitre les principaux avantages et défis de la mutualisation ainsi que ses concepts architecturaux à travers l'explication du style architectural de toute architecture d'application mutualisée. Ce travail a été mené par l'analyse des travaux de recherches existants, et plus particulièrement par l'analyse de certains cas industriels (voir section 6.2) que nous avons développé et qui forment, à notre avis, une source d'information précieuse, car ils représentent concrètement les problèmes actuels de l'industrie de SaaS.

PARTIE II

La mutualisation de SaaS : défis abordés, état de l'art et problématiques dégagées

Premier Défi abordé : Gestion de la Variabilité des besoins de locataires

4.1 Introduction

Tel que nous l'avons montré dans le chapitre précédent, le modèle SaaS est généralement conçu de façon à tirer parti des avantages apportés par les économies d'échelle en offrant la même instance d'application à plusieurs locataires en même temps, suivant le principe de mutualisation. La mise en œuvre réussie de ce principe exige une architecture spécifique permettant à chaque locataire d'avoir l'illusion d'être l'unique locataire de l'application et au fournisseur d'installer pour une seule fois la pile technologique matérielle et logicielle nécessaire (confère section 3.3). Toutefois, pour supporter et attirer un grand nombre de locataires, l'application doit être adaptable et configurable pour répondre aux exigences variables de chaque locataire. Par conséquent, les fournisseurs SaaS font face à un nouveau défi (confère sous-section 3.3.3) : *la gestion dynamique de la variabilité* des besoins de locataires au sein de l'application.

Dans ce chapitre, nous abordons ce défi de gestion ainsi que l'analyse que nous faisions pour le faire passer d'un simple besoin en un objet manipulable de première classe. Nous nous basons dans cette analyse sur une nouvelle voie de recherche qui a émergé dans la communauté de SaaS. Cette dernière consiste à réutiliser des concepts de modélisation et de gestion de variabilité employés avec succès dans le domaine d'ingénierie des lignes de produits logiciels pour soutenir la gestion de la variabilité dans les applications SaaS mutualisées. La clé de cette approche consiste à définir et à exploiter une modélisation de variabilité explicite, dont le but est d'établir un lien entre la variabilité de l'application en termes de besoins de locataires et la variabilité permise par les alternatives de développement et de réalisation. À travers le modèle de variabilité et les informations sur les locataires déjà hébergés, les fournisseurs de SaaS sont accompagnés dans leur décision sur les variations de l'application à gérer.

Ce chapitre est organisé de la manière suivante : la section 4.2 exploite les concepts de gestion de la variabilité proposées par le domaine d'ingénierie des lignes de produits logiciels et analyse la possibilité de leur réutilisation dans le contexte des applications SaaS mutualisées. Ensuite, la section 4.3 présente et classifie les approches existantes de gestion de la variabilité dans une architecture orientée services, étant donné que l'AOS constitue le modèle d'architecture adopté pour la construction nos applications SaaS. La section 4.4 explique les limites que nous avons détectées dans les approches proposées et prépare le terrain pour notre contribution. Enfin, la section 4.5 conclut le chapitre.

4.2 Gestion de la Variabilité

Le concept de variabilité provient de l'industrie automobile dans laquelle différentes combinaisons des options définissant une gamme de produits représentés sous forme de variantes (type de moteur, couleur, vitres, châssis, etc.), peuvent être choisies lors de la fabrication d'une voiture. Dans la discipline

du génie logiciel, ce concept est tout d'abord introduit dans le domaine des lignes de produits logiciels où la variabilité est définie comme « *la capacité d'un système logiciel ou d'un artéfact à être efficacement étendu, modifié, adapté ou configuré pour une utilisation dans un contexte particulier* » [SVGB05]. Dans ce contexte, le logiciel est développé par un éditeur du logiciel puis fourni à un client pour être installé et exécuté dans sa propre infrastructure. Cela signifie que les variantes choisies par ce client doivent être compilées et intégrées dans le logiciel avant son exécution.

Cependant, dans un contexte SaaS mutualisé, le logiciel est toujours développé par un éditeur mais il est servi à tous les locataires via Internet. En principe, toutes les variantes sont à composer au moment où les locataires accèdent aux fonctionnalités du logiciel lors de son exécution, via la mise en place d'une architecture centralisée et dynamique. Cette dernière permettra ainsi à un fournisseur de SaaS de faciliter la commercialisation de son application en exploitant rapidement des nouveaux *segments de marché* (parties de marché ayant des caractéristiques communes) à travers la construction des réseaux de revendeurs qui profitent de la capacité de mise en place automatisée de nouveaux locataires (confère section 3.3.2). De ce fait, la variabilité dans les applications SaaS mutualisées peut être classée en deux catégories [KJ11] : (i) *la variabilité de segment* identifiée par un segment du marché dont un ou plusieurs locataires font partie. Des exemples de cette catégorie de variabilité incluent les différents standards de devises, les règles fiscales privées des pays, les dispositions différentes pour les petites entreprises et celles individuelles, etc. ainsi que (ii) *la variabilité de locataire* générée par les exigences spécifiques de chaque locataire. Des exemples de cette variabilité peuvent également concerner l'ergonomie de l'application ou des fonctionnalités spécifiques. Suivant cette distinction, une meilleur séparation des préoccupations et une plus grande souplesse de configuration de l'application pourra avoir lieu. Nous suivons cette distinction et nous la trouvons assez pertinente pour traiter un de problèmes liés à ce défi que nous allons détailler plus loin dans ce manuscrit (voir section 6.3).

Toutefois, *la gestion de la variabilité* dans les applications SaaS mutualisées est un terme plutôt large qui nécessite d'être bien identifié ainsi que les activités à réaliser pour effectuer cette gestion. D'après notre analyse de l'état de l'art concernant ce défi [SR11, MMLP09, SRS+10, MLP08, KJ11], nous avons pu identifier trois activités de gestion principales :

1. *La séparation entre la commonalité et la variabilité :* en identifiant les parties de l'application qui sont communes entres les locataires et les autres parties qui varient d'un locataire à un autre.
2. *La modélisation de la variabilité :* en décrivant d'une manière formelle la variabilité identifiée et supportant sa réification en tant qu'objets manipulables de première classe.
3. *La résolution de la variabilité :* en adaptant dynamiquement l'architecture de l'application mutualisée en fonction de son modèle de variabilité et les configurations de chaque locataire.

Dans la suite de cette section, nous allons détailler chacune de ses activités.

4.2.1 La séparation entre la commonalité et la variabilité

Dans une application SaaS mutualisée, les besoins fonctionnels et non-fonctionnels des locataires sont susceptibles de varier. L'origine de cette variabilité provient généralement de la répartition de ces locataires à travers différentes zones géographiques et différentes échelles industrielles, établissant une différence entre leurs attentes et leurs exigences. Cette situation implique que les fournisseurs de SaaS sont confrontés à deux objectifs qui doivent être équilibrés [MMLP09]. Tout d'abord, et afin d'attirer suffisamment de locataires, ils doivent répondre à leurs exigences spécifiques en fournissant différentes configurations et adaptations du logiciel. Deuxièmement, ils doivent s'assurer que l'application conservent suffisamment de commonalité, afin que le principe de mutualisation reste rentable à adopter.

Pour trouver un tel équilibre, les fournisseurs doivent s'éloigner de la construction d'une application rigide capable de tout faire ou d'offrir peu d'options de configuration. Au lieu de cela, ils doivent concevoir l'application d'une manière qui leur permettra de la faire évoluer avec le temps et de répondre au fur et à mesure aux besoins des nouveaux locataires, voire même aux changements des besoins de locataires existants. Faisant ainsi, les fournisseurs ne doivent pas perdre de vue que l'objectif final de la mutualisation est d'exploiter les points communs entre les locataires et que ces points restent suffisamment élevés pour qu'une seule instance d'application soit justifiable et viable. Ainsi, le développement de l'application doit impliquer le maintien de cet équilibre entre la commonalité et la variabilité sur une base continue. Il s'agit de profiter autant que possible de l'existence de la commonalité, et de gérer de manière appropriée la variabilité lorsque nécessaire.

Conduites par ce raisonnement, des nouvelles approches commencent à émerger dans l'état de l'art, qui tentent de réutiliser des techniques de gestion de la variabilité bien établies dans le domaine d'ingénierie des lignes de produits logiciels, pour soutenir la variabilité dans les applications SaaS mutualisées [MLP08, SR11, MMLP09]. Dans la suite, nous allons détailler les concept de gestion proposés par les recherches effectuées sur les lignes de produits logiciels, et comment celles-ci peuvent être réutiliser dans notre contexte.

4.2.1.1 Les lignes de produits logiciels

Les lignes de produits logiciels (LDP) est un paradigme du génie logiciel visant à réduire les coûts de développement et accélérer le temps de mise des applications sur le marché, à travers la modélisation des applications logiciels similaires comme une ligne de produits. Les entreprises comme *Nokia*, ou *Motorola*, qui ont adopté l'approche LDP pour gérer la diversité des logiciels dans le domaine de téléphones mobiles [MT00, Nor02], sont la preuve qu'une telle adoption peut apporter des améliorations importantes en termes de productivité et de réduction du temps de mise de nouveaux produits logiciels sur le marché.

Plus précisément, une LDP est un ensemble de produits logiciels partageant des caractéristiques et des fonctionnalités en commun. Ils répondent aux besoins spécifiques d'un client ou d'un segment de marché particulier et sont élaborés de manière planifiée à partir d'un ensemble d'artéfacts logiciels de base. Un artéfact est un élément architectural qui permet de construire un logiciel, par exemple un document de spécification, un service, un composant, un fichier de code, etc. Au lieu de décrire un système logiciel unique, le modèle de LDP décrit l'ensemble des produits dans un même domaine. Cela peut être accomplie en faisant la distinction entre les éléments communs à tous les produits d'une ligne et les éléments qui peuvent varier d'un produit à l'autre. Le succès grandissant de l'approche LDP dans l'industrie logicielle est dû à sa capacité d'offrir les bons moyens aux entreprises d'exploiter la commonalité et la variabilité à travers leurs produits.

Selon les principes de LDP, le processus de développement d'une application est divisé en deux phases (voir figure 4.1) [ML97, VdL02, PBVDL05] :

La phase de l'ingénierie du domaine : cette phase a pour objectif la conception et le développement du cœur du système logiciel y compris sa variabilité. Le but est d'établir une plateforme logicielle réutilisable [ML97]. Il s'agit de faire du développement pour la réutilisation et non pour un produit spécifique. Cette phase est composée de trois étapes :

1. *L'analyse du domaine :* cette étape englobe toutes les activités pour capturer et formaliser les connaissances du domaine afin de le définir. Son entrée se compose des exigences de ce dernier (la stratégie, les buts, les contraintes, etc.) et la sortie comprend un modèle abstrait qui représente les différentes possibilités [PBVDL05]. Les exigences du domaine sont

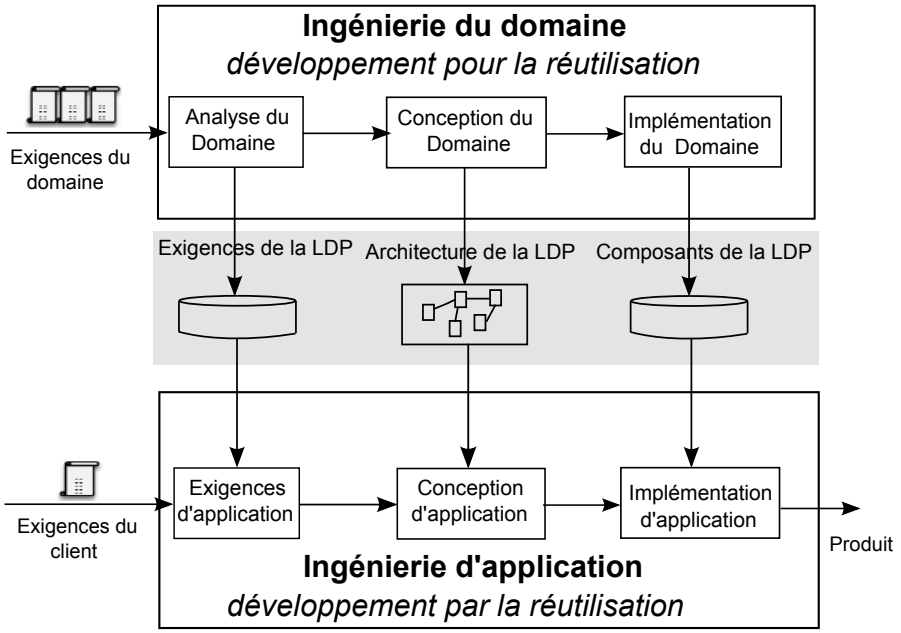


Figure 4.1 – Les phases du processus de développement de LDP

une partie fondamentale de l'analyse et contribuent à l'anticipation des changements à venir dans les lois, les normes, les technologies et les besoins du marché. Les expériences et les compétences des experts du domaine représentent également des ressources implicites qui impactent fortement le résultat de cette étape. Dans l'état de l'art, il existe plusieurs méthodes pour guider l'analyse du domaine dont la plus connue est celle du *modèle de caractéristiques FODA* [BHST04]. Le domaine dans FODA est décrit dans un modèle de caractéristiques (une caractéristique est appelée *feature*), spécifié sous forme d'arbre dont les nœuds représentent les caractéristiques du domaine et les arcs spécifient des liens de composition entre eux. Les caractéristiques représentent les différentes fonctionnalités du système et l'illustrent par une vue structurelle. Les parties communes de LDP sont modélisées par des caractéristiques obligatoires qui définissent les fonctionnalités à inclure dans tous les produits membres de LDP. Les parties variables sont modélisées par des caractéristiques optionnelles ou alternatives, qui définissent respectivement les fonctionnalités à inclure seulement dans certains produits et celles mutuellement exclusives par produit. La Figure 4.2 montre un simple exemple de modèle de caractéristiques pour une ligne de produits de voitures. Les caractéristiques obligatoires sont représentées par des rectangles avec des cercles pleins au dessus, tandis que les caractéristiques optionnelles sont représentées par des rectangles avec des cercles vides au dessus. La caractéristique `Climatisation` dans l'exemple est optionnelle. La variabilité du moteur est spécifiée par une caractéristique alternative `Moteur` avec deux caractéristiques variantes `Essence`, et `Diesel`. Une caractéristique alternative est représentée par un arc de cercle à travers les arcs des caractéristiques variantes.

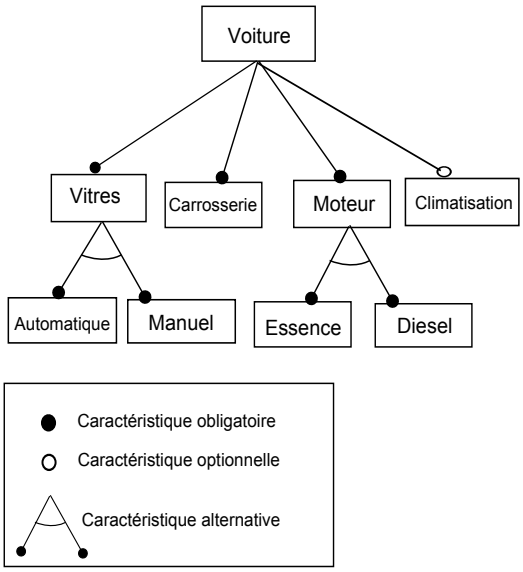


Figure 4.2 – Le modèle de caractéristiques d'une LDP de voitures

2. *La conception du domaine :* cette étape apparaît à la suite de l'analyse du domaine et repose sur ses résultats. Elle englobe toutes les activités pour définir l'architecture de référence et l'ensemble des composants logiciels réutilisables. Une architecture de référence définit de façon abstraite une structure commune de haut niveau pour tous les produits de la ligne. Cette structuration met en évidence les composants importants et leurs relations. Ensuite, cette architecture est successivement affinée de façon à faire explicitement apparaître la variabilité identifiée pendant l'analyse du domaine. Elle permet d'automatiser la dérivation des produits en définissant de façon systématique comment les artéfacts réutilisables doivent être assemblés. Ainsi, toutes les architectures des produits dérivés doivent être « conformes » à l'architecture de référence de la ligne de produits.
3. *L'implémentation du domaine :* cette étape consiste à implémenter l'architecture de référence définie dans la conception du domaine sous forme de composants adaptables et extensibles (variabilité à l'intérieur de composants). Ces derniers vont être réutilisés dans la phase d'ingénierie de l'application pour la construction de chaque produit. Ils existe plusieurs façons pour développer de composants adaptables. Les plus connues sont celles basées sur les techniques classiques telles que la réflexion, l'interface de configuration, la génération de code, la compilation conditionnelle et l'extension par *« plug-in »* [JGJ97, Bos00].

La phase de l'ingénierie d'application : cette phase a pour objectif d'adapter la plateforme réalisée pendant l'ingénierie du domaine aux besoins individuels de chaque client. La plateforme adaptée est ainsi appelée *application*. Elle concerne un produit spécifique de la ligne. Il s'agit d'un processus d'assemblage et de configuration des artéfacts réutilisables en conformité avec l'architecture de référence. Cette phase est également composée de trois étapes :

1. *L'identification des exigences de l'application :* cette étape consiste à identifier les exigences de l'application afin de déterminer les artéfacts à réutiliser à partir de la plateforme. Son entrée comprend le modèle de caractéristiques élaboré durant l'analyse du domaine ainsi

que les exigences du client. Sa sortie précise les caractéristiques qui doivent faire partie du produit à dériver. Il peut y avoir de nouvelles exigences d'un client qui n'ont pas été capturées au cours de l'analyse du domaine. Pour cela, une préoccupation spécifique à cette étape est celle de la détection de différences entre les exigences et les capacités, et d'évaluer ainsi ces différences au regard de l'effort d'adaptation requis pour les intégrer.

2. *La conception de l'application :* cette étape consiste à produire l'architecture de l'application. Cela passe à travers la spécialisation de l'architecture de référence en résolvant sa variabilité. Cette étape introduit également les changements spécifiques à l'application en termes d'artéfacts non identifiés durant la conception du domaine. L'entrée de cette étape se compose des exigences du client, et la sortie se compose d'une architecture stable du produit.
3. *L'implémentation de l'application :* cette étape consiste à créer l'application désirée. Les principales préoccupations sont la sélection des composants réutilisables à partir de la plateforme, ainsi que la réalisation de ceux spécifiques à l'application (qui ne font pas partie du domaine). Ainsi, les composants réutilisés et spécifiques sont assemblés pour former le produit. L'entrée de cette étape se compose de l'architecture de l'application et la sortie se compose d'une application exécutable.

Discussion par rapport à SaaS : les concepts fournis par le paradigme de LDP dépassent la simple idée de réutilisation de composants de manière ad-hoc vers une réutilisation systématique planifiée de l'ensemble des artéfacts d'un logiciel, et ce d'une manière centrée autour de la manipulation des concepts de commonalité et de variabilité. L'objectif est d'automatiser la dérivation de nouveaux produits logiciels adaptés aux besoins spécifiques des clients et des marchés.

Cet objectif est partagé avec les services applicatifs du cloud et a particulièrement motivé l'introduction des concepts de LDP pour supporter la variabilité du SaaS mutualisé. Les avantages d'une telle introduction incluent la fourniture d'un support pour améliorer la communication et la modélisation de la variabilité et de soutenir son évolution sur une base continue. Toutefois, la variabilité de LDP est explicite, c.à.d. la manière selon laquelle les différents produits varient est connue à l'avance et est ainsi déjà planifiée. Ceci peut être un facteur limitant pour les fournisseurs de SaaS puisqu'une variabilité déjà planifiée limite en général la flexibilité permise face à la nature dynamique de l'environnement de SaaS qui nécessite parfois la gestion des situations de variabilité imprévus [MMLP09]. Cependant, la gestion d'une variabilité « infinie » est encore une question ouverte et un axe de recherche qui manque de maturité. De ce fait, les travaux existants qui réutilisent les concepts de LDP dans le contexte de SaaS mutualisé sont en mesure d'établir un équilibre entre le niveau de variabilité à supporter par l'application et la complexité de gestion d'une telle variabilité.

4.2.1.2 La variabilité des LDP

Dans toute approche d'ingénierie supportant les LDP, la variabilité est la première activité à considérer. Elle est utilisée pour identifier les éléments qui différencient les produits à dériver. Bien évidemment, sa gestion demande plus d'efforts que celui nécessaire pour la gestion de commonalité. En effet, les artéfacts communs dans une LDP sont identifiés et utilisés tels quels pour la construction de tous les produits. Cependant la variabilité demande, en plus de son identification et sa *modélisation*, des mécanismes pour sa *résolution*, ceux qui sont responsable *d'instancier* l'architecture de référence et d'assembler ses composants pour aboutir à un produit spécifique. Toutefois, en dehors du contexte des lignes de produits, la variabilité peut concerner un seul produit, c.à.d. la variabilité fait partie du produit et elle est résolue après que le produit soit délivré et installé dans son environnement d'exécution. Dans le contexte des

lignes de produits, la variabilité doit être explicitement spécifiée et faisant partie de la ligne de produits et ce, contrairement à la variabilité d'un seul produit. En effet, la variabilité dans les lignes de produits doit être résolue avant que le produit ne soit délivré et installé dans son environnement d'exécution. De ce fait, il existe deux dimensions où la variabilité peut apparaitre [BFG+02, PBVDL05] : le temps et l'espace (voir figure 4.3) :

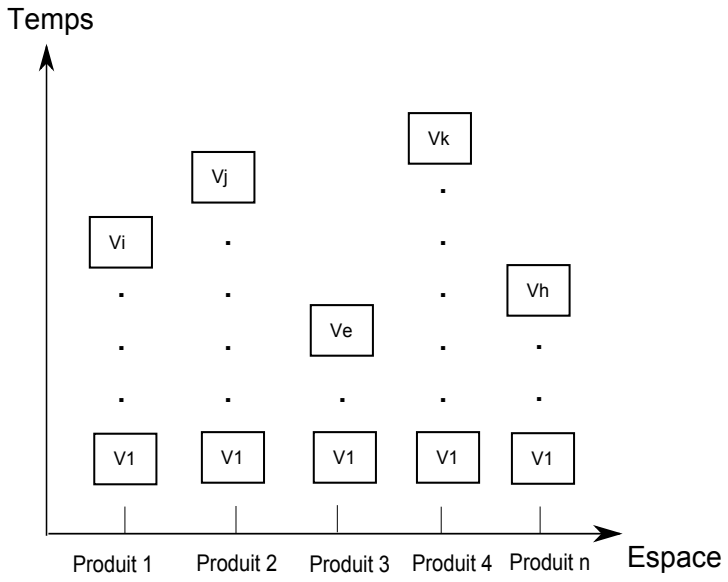


Figure 4.3 – La dimension espace et temps dans la variabilité

La variabilité dans le temps : cette dimension concerne la modification d'un produit spécifique de la ligne après sa dérivation. Cela est généralement dû à la réalisation des améliorations au niveau de qualité et/ou de fonctionnalité des artéfacts qui composent le produit et qui ne faisaient pas partie de la ligne au moment de sa dérivation.

La variabilité dans l'espace : cette dimension concerne la variation entre plusieurs produits de la même ligne. Les même éléments logiciels sont utilisés dans plusieurs produits et la variation concerne principalement des variations de fonctionnalités, c.à.d. les produits diffèrent dans les fonctionnalités qu'ils supportent. Lorsque l'on parle de variabilité dans les LDP, cela signifie la variabilité dans l'espace.

Discussion par rapport à SaaS : la figure 4.3 montre la variabilité dans le temps des produits d'une version à une autre et la variabilité dans l'espace de la ligne de produits d'un produit à un autre. Bien que la distinction entre ces deux dimensions est essentielle pour comprendre les concepts de gestion de la variabilité apportés par l'approche LDP, elle contredit avec le principe de mutualisation de SaaS. Nous rappelons que ce dernier représente un seul système logiciel dont l'accès à ses fonctionnalités se fait à travers Internet par l'ensemble de ses locataires en même temps, et non à travers la dérivation de plusieurs produits configurables à déployer et à faire évoluer séparément pour chacun d'entre eux.

Pour résoudre cette ambiguïté, nous considérons une application SaaS mutualisée comme une ligne de produits où chaque produit est *virtuellement* dérivé à travers les configurations de locataires. Ces derniers ont l'illusion d'accéder à une application (ou un produit) dédiée, tandis que dans la réalité chacun accède à une partition virtuelle du système que nous avons tendance à appeler produit pour garder le même logique de raisonnement offert par les LDP lors du traitement de la variabilité, et plus particulièrement, lors de sa modélisation.

4.2.2 La modélisation de la variabilité

La modélisation de la variabilité est une activité essentielle à réaliser lors de gestion de la variabilité dans une application SaaS mutualisée. Elle permet de décrire d'une manière formelle et explicite les variations de l'application tout en identifiant les artéfacts qui varient et comment ils varient. Cela revient à explicitement documenter la variabilité et à la réifier en tant qu'objets traitables de première classe. Selon les auteurs dans [PBVDL05], la modélisation *explicite* de la variabilité lors de sa gestion permet d'améliorer la prise de décisions en forçant les développeurs à documenter et à justifier les raisons qui les poussent à l'introduire. Elle permet également d'améliorer la communication entre les différents acteurs de l'application en fournissant une abstraction des artéfacts variables sans la nécessité de rentrer dans les détails techniques.

4.2.2.1 Les points de variations et les variantes

D'après les concepts de gestion et de modélisation de la variabilité proposés par les approches de LDP, la variabilité est principalement documentée à travers l'identification de ce qu'on appelle des *points de variations*. Ces derniers identifient précisément des endroits dans l'architecture de LDP où des changements/variations peuvent apparaître. Un point de variation concrétise une décision de conception retardée à un stade ultérieur, dans lequel une idée plus claire sur le contexte du client pourra être connue [JGJ97]. Un point de variation peut être caractérisé par un ou plusieurs choix, appelés des *variantes*. Celles-ci représentent des choix architecturaux possibles sur le point de variation. Lorsqu'un choix architectural est effectué et implémenté, le point de variation devient résolu.

La dérivation d'un produit particulier à partir d'une ligne de produits se fait ainsi en choisissant les variantes désirés sur les points de variations disponibles. Cependant, le moment exact où ces choix sont effectués est variable. Ce moment est généralement appelé *« binding time »* . D'après l'état de l'art [Kru02], les choix de variantes sont principalement effectués :

- au moment de la *décision de réutiliser un composant donné*. Lorsqu'un composant est choisi pour être intégré dans l'architecture du produit à dériver, des choix de variantes concernant la variabilité interne de ce composant doivent être effectués,
- au moment de *l'implémentation du produit*, c.à.d lors de l'assemblage de ces composants au sein de son architecture, et ce en faisant le choix entre les composants alternatives,
- au moment de *la compilation du produit* assemblé pour éliminer le code non utile [PBVDL05],
- au moment du déploiement pour tenir compte de la spécificité de l'infrastructure sur laquelle le produit devra être installé,
- au moment de *l'exécution du produit* pour réagir dynamiquement face aux changements de l'environnement d'exécution et des exigences du client.

Lors de la dérivation d'un produit spécifique, il est possible d'utiliser plusieurs binding times afin de traiter les différentes décisions de conception retardées. Cela permet de prendre certaines décisions au début du cycle de vie du produit, lors de la conception architecturale et aussi plus tard, lors de l'exécution.

Cette approche permet d'intégrer les compétences des différentes personnes intervenant dans un projet. Certaines décisions, par exemple, peuvent être prises par les développeurs assemblant divers composants métiers alors que d'autres décisions peuvent être prises par le client pour configurer l'application au cours de l'exécution.

4.2.2.2 L'intégration des informations de la variabilité dans les artéfacts

Durant la phase de l'ingénierie du domaine (confère section 4.2.1.1), l'étape d'analyse du domaine identifie la variabilité de LDP au niveau d'abstraction le plus élevé (à travers le modèle FODA). Cette identification conduit à introduire une variabilité dans plusieurs éléments de conception et ensuite raffinée en variabilité au niveau de composants qui implémentent la LDP. Une approche pour modéliser la variabilité sur ces différents niveaux d'abstraction (analyse, conception, implémentation) consiste à introduire ses informations (en termes de points de variations et des variantes) comme une partie intégrante des artéfacts de développements situés sur chaque niveau (voir figure 4.4 (a) comme un exemple d'intégration des informations de la variabilité au niveau de la conception du domaine). Contrairement au modèle de caractéristiques FODA qui supporte la modélisation de la variabilité ainsi que les caractéristiques communes, les artéfacts de développement logiciels traditionnels tels que les diagrammes d'UML, n'étaient pas conçus pour supporter nativement la modélisation de la variabilité. De ce fait, différents travaux ont essayé d'étendre les méta-modèles d'UML pour introduire la variabilité dans ses modèles et ses diagrammes fréquemment utilisés, comme par exemple les modèles de cas d'utilisation [vdML02, BHP03, Gom04], les diagrammes statiques (classes et composants) [WG04, CJB00, ZJ05], les diagrammes dynamiques (séquences, machines à états et processus métiers) [ABM00, Gom00, ZJ05, SP06] et ainsi d'autres artéfacts d'implémentation [BFG+02, VGBS01]. Une classification de ces différentes techniques pour modéliser la variabilité se trouve dans [SD07]. Nous allons brièvement citer quelques exemples :

- Les auteurs dans [ZJ05] montrent l'intégration des concepts de la variabilité dans le diagramme de classes d'une LDP (voir Figure 4.4). Cette intégration est réalisée à travers l'utilisation du concept d'héritage en combinaison avec les stéréotypes (le mécanisme d'extension d'UML). Chaque point de variation dans cette approche est représenté par une super-classe abstraite (stéréotypées « *Variation* »), héritée par des sous-classes concrètes (stéréotypées « *Variant* ») qui modélisent les différentes variantes. Les classes stéréotypées « *Optional* » indique l'optionnalité de leur existence dans un produit spécifique de la LDP. Les auteurs de cette approche présentent également un mécanisme de dérivation de produits basé sur la technique de transformation de modèles [JABK08], et appliquent la même méthodologie sur les diagrammes de séquences pour dériver l'aspect dynamique des produits.
- Les auteurs dans [Gom04] présentent une approche similaire pour modéliser la variabilité dans les modèles de cas d'utilisation. Il introduisent la variabilité en utilisant trois stéréotypes : « *kernel* » pour identifier les cas d'utilisations nécessaires pour tous les produits d'une LDP , « *optional* » pour identifier les cas d'utilisations à inclure dans certains produits et « *Alternative* » pour identifier les cas d'utilisations qui sont mutuellement exclusifs par produit. Les auteurs ne présentent aucun mécanisme de dérivation. L'utilité de leur travail est restreinte à un but descriptif et limitée à des objectifs de documentation et de communication de la variabilité avec les clients d'une manière plus proche à leur compréhension à travers les cas d'utilisation.
- Les auteurs dans [SP06] étendent les diagrammes de processus métiers (BPMN) d'UML avec les stéréotypes suivants : « *VarPoint* », « *Alternative* », « *Variable* », « *Default* », « *Optional* » et

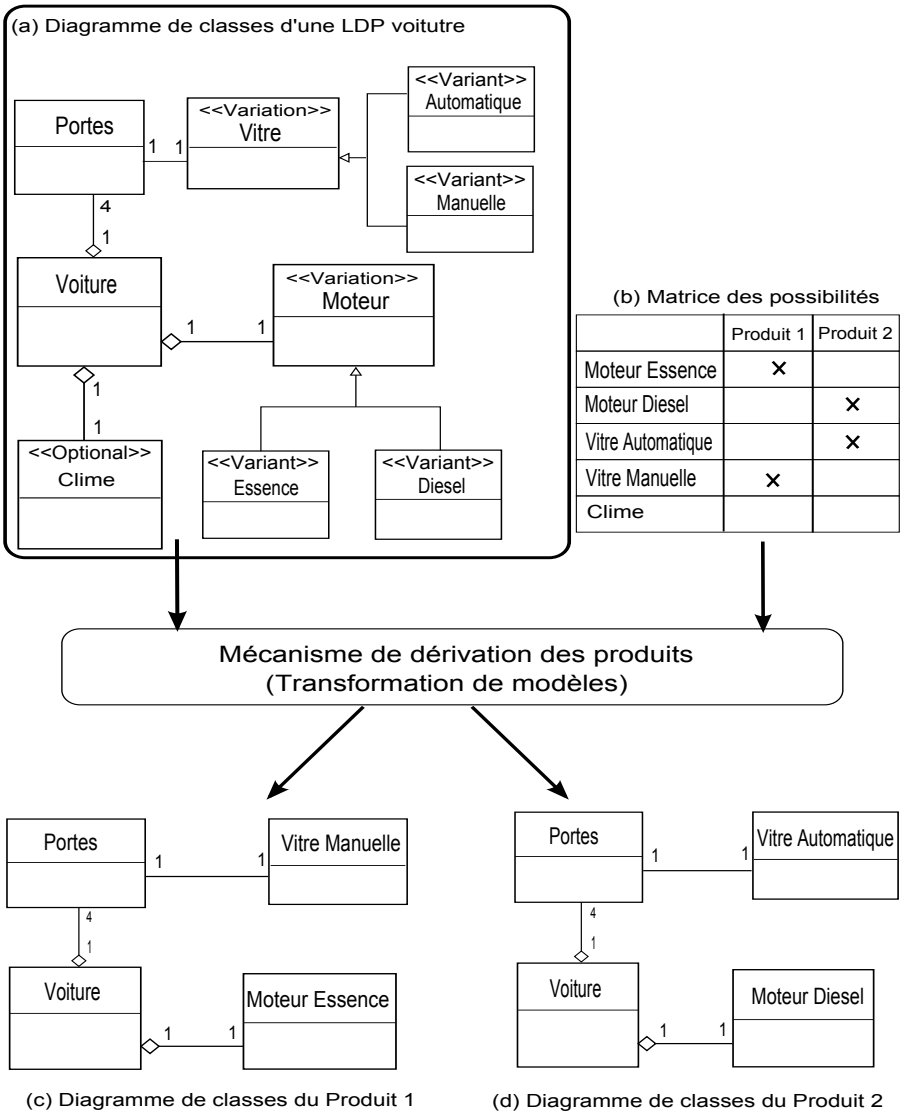


Figure 4.4 – Manipulation d'une LDP de voitures en diagramme de classes d'UML selon l'approche de [ZJ05]

« *Null* ». Ce nombre relativement important de stéréotypes a pour but de tenir compte de la variabilité à un niveau assez détaillé et de couvrir des situations de modélisation très complexes.

Discussion par rapport à SaaS : ces différentes approches partagent toutes le fait qu'elles traitent la variabilité comme une partie intégrante d'artéfacts de développement et non comme une préoccupation séparée. Bien que cela offre une méthode pratique et rapide pour gérer la variabilité, de nombreuses limitations peuvent être rapidement aperçues. Le problème d'une approche où la variabilité est intégrée dans les artéfacts de développement réside dans le fait que cette variabilité est dispersée à travers ces différents artéfacts. Cela augmente la taille et la complexité du modèle de variabilité. De plus, puisque les différents artéfacts peuvent être annotés avec des notions de variabilité différentes, le fait de décrire les dépendances entre la variabilité dans ces artéfacts devient assez difficile. Par exemple, il devient difficile de déterminer

quelles informations de variabilité durant l'analyse du domaine ont influencées quelles informations de variabilité dans la conception et l'implémentation. Par conséquent, ces différentes manières de modélisation ne s'intègrent pas dans une vision globale de variabilité pour l'ensemble du système et remettent en question l'intérêt de leur utilisation dans le contexte des applications SaaS mutualisées. En effet, l'architecture de ce type d'applications est généralement orientée services, comportent différentes couches d'abstraction (processus métiers, services, composants, infrastructure) et une diversité d'artéfacts qui est plus complexe que celle des architectures à base de composants, pour lesquelles les approches de LDP ont été initialement développées. Ainsi, l'intégration directe des informations de la variabilité dans les artéfacts de développement liés à l'AOS n'assure pas l'efficacité de gestion que nous cherchons à réaliser.

4.2.2.3 Orthogonalité de définition

Conduit par la nécessité de décrire les dépendances de variabilité à travers les différents artéfacts et d'uniformiser sa manière d modélisation et du traitement, certaines approches ont été proposées suggérant de définir la variabilité dans un modèle orthogonal [BGL+04, MA02, BLP05]. L'approche la plus connue est celle d'OVM (orthogonal variability model) [PBVDL05, MHP+07]. OVM présente une méthode uniforme pour modéliser la variabilité d'une LDP dans un modèle séparé, sans nécessité d'étendre les artéfacts de développement utilisés avec des nouvelles notions (exemples de stéréotypes pour les diagrammes d'UML). Cette séparation permet de réduire la taille et la complexité de la variabilité, et ainsi de la traiter comme une nouvelle vue architecturale. La figure 4.5 montre le méta-modèle d'OVM, alors que la figure 4.6 montre un exemple d'instanciation de ce méta-modèle pour modéliser la variabilité d'une LDP de voitures. Les concepts de modélisation fournis par OVM sont les suivants :

- *Point de variation (PV), variante et dépendance de variabilité :* un PV dans OVM (représenté par un triangle) documente un ou plusieurs endroits variables d'une LDP auxquels une variation peut se produire. Il indique l'existence de différentes solutions, chacune d'entre elles entraînera une fonctionnalité différente pour le produit à dériver. Une variante (représentée par un rectangle) documente une solution possible d'un PV. Selon OVM, les PV et les variantes peuvent être soit facultatives (*Optional*, par exemple la ligne pointillée entre le PV `climatisation` et la variante `Oui`) soit obligatoires (*Mandatory*, par exemple la ligne continue entre le PV `Sécurité` et la variante `Air bag`), à spécifier lors de la modélisation des dépendances entre eux (*Variability Dependency*). Le *binding* d'un PV obligatoire doit être toujours effectué, c.à.d., au moins une variante de ce PV doit être choisie. Les variantes obligatoires doivent être choisies sur un PV alors que celles optionnelles peuvent, mais ne sont pas obligées à être choisies et peuvent ainsi être transformées en des alternatives. Contrairement à d'autres approches de modélisation de la variabilité, OVM permet de choisir plusieurs variantes sur un même PV et le nombre de choix autorisé est limité par des cardinalités (les attributs min et max de la classe *Alternative choice*).
- *Artéfacts de développement et dépendance d'artéfacts :* ce concept est introduit par OVM pour établir des liens de traçabilité entre les éléments de la variabilité (en termes de PV et des variantes), et les artéfacts de développement concernés. L'association *Represented by* indique les artéfacts qui représentent un PV, tandis que l'association *Realized by* indique les artéfacts qui implémentent les variantes. La classe *Development artéfact* représente tout type d'artéfact. Certains artéfacts particuliers liés à un domaine précis peuvent être des spécialisations de ce concept, donc le méta-modèle d'OVM peut être étendu dans ce sens.
- *Contraintes :* OVM défini un ensemble de contraintes décrivant en particulier des règles de dépendance entre les PV, les variantes et entre les PV et les variantes (*Constraint Dependency*). La contrainte *Excludes* précise une exclusion mutuelle des variantes ou des PV pour interdire les

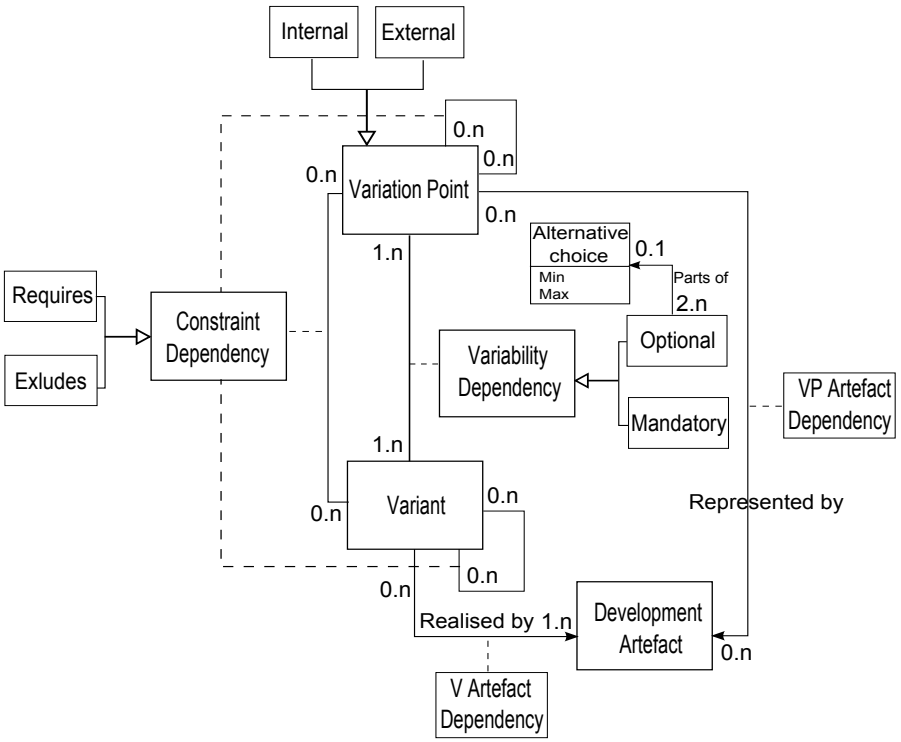


Figure 4.5 – Le méta-modèle d'OVM selon [PBVDL05]

choix incompatibles ; par exemple, pour interdire l'existence de deux variantes dans un produits en même temps. La contrainte *Requires* spécifie une implication ; c.à.d. si une variante est choisie, une autre variante doit être également choisie (par exemple, dans la figure 4.6, la variante `Anti vol` nécessite le choix de l'option `Double vitrage`).

– *Variations interne et externe :* OVM distincte entre les PV qui ne sont visibles que pour les développeurs (*Internal*), et les PV qui sont communiqués aux clients (*External*).

Discussion par rapport à SaaS : dans la littérature, OVM a été proposée uniquement pour documenter la variabilité. Ainsi, il n'existe pas de mécanismes spécifiques à ce modèle qui définissent la manière dont la variabilité peut être résolue. Cependant, des différents travaux de recherches [PBP06, LSGF08, ML08] ont proposés d'étendre OVM avec de concepts de modélisation supplémentaires, permettant de référencer (à travers le modèle de variabilité) les éléments variables de l'architecture et de les manipuler à travers des mécanismes de dérivation réflexifs. Dans ce sens, cette thèse propose une extension du méta-modèle d'OVM dans le but de résoudre la variabilité des applications SaaS mutualisées. L'utilisation d'OVM dans notre contexte est justifié par la complexité de l'architecture des applications SaaS, celle que nous cherchons à réduire par une approche de gestion de variabilité non-intrusive. La nature de l'extension que nous proposons a été identifiée grâce à une évaluation de certains aspects qui font la différence entre la variabilité de LDP et celle du SaaS mutualisé. Nous détaillons ces différences plus tard dans ce chapitre.

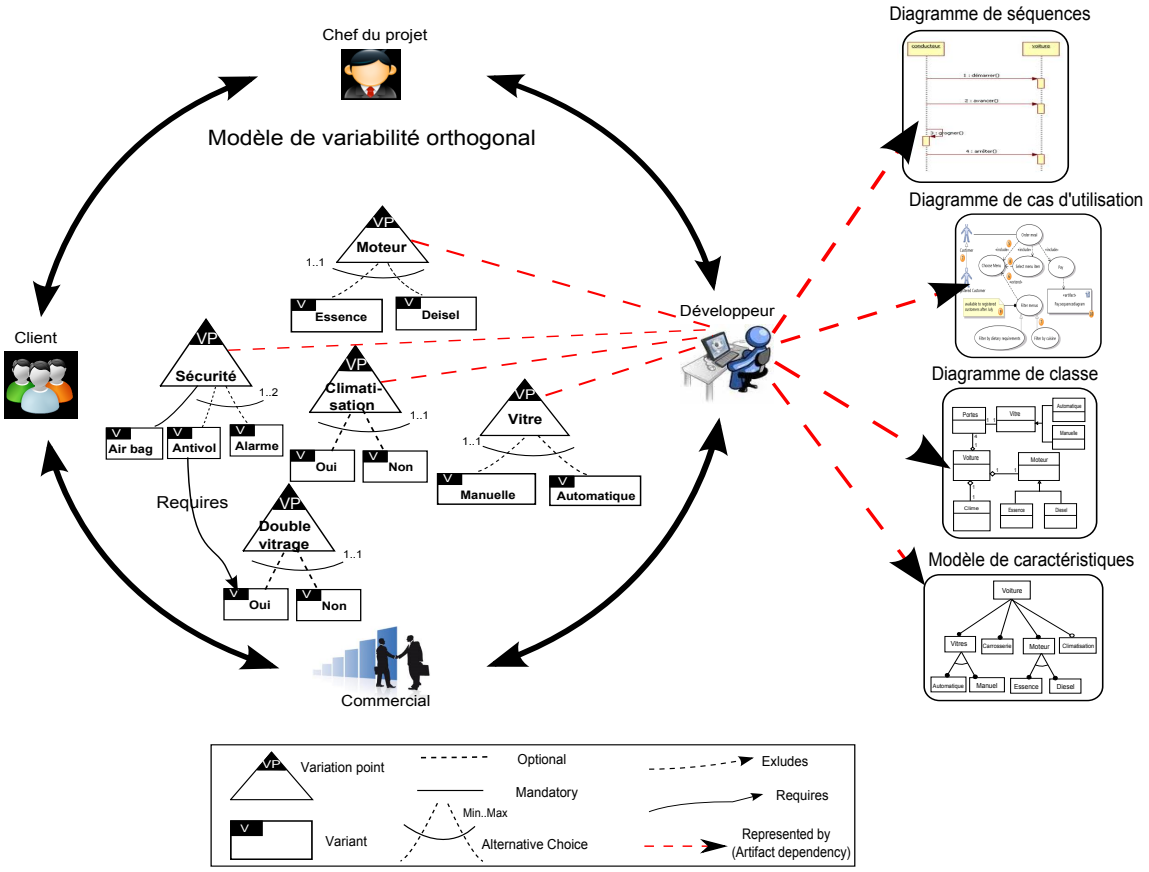


Figure 4.6 – Exemple d'instanciation du méta-modèle d'OVM

4.2.3 La résolution de la variabilité : adaptation dynamique de l'architecture

De plus en plus de logiciels sont confrontés aux changements d'exigences qui les obligent à s'adapter en permanence pour rester disponibles et quasiment sans interruption [MBNJ09]. Ces systèmes, que l'on peut qualifier de dynamiquement adaptatifs, présentent de hauts degrés de variabilité et sont conceptualisés comme des lignes de produits dans lesquelles la variabilité serait résolue à l'exécution [MKBJ08].

Pour une application SaaS mutualisée, l'adaptation dynamique de l'application guidée par les configurations prédéfinies de locataires représente un aspect crucial pour pouvoir répondre à leurs exigences variables sans affecter la disponibilité du système. En effet, si l'adaptation envisagée pour un locataire se fait d'une façon statique, l'application sera forcement interrompue pour être recompilée et redéployée. Ceci va interdire l'accès à ses services et ses fonctionnalités pour les autres locataires, et ce pour des modifications qui ne sont pas à l'origine de leur propres exigences. Puisque le nombre de locataires peut augmenter de façon continuelle, les interruptions deviennent plus fréquentes et peuvent conduire à des très graves problèmes de disponibilité.

4.2.3.1 Les lignes de produits dynamiques

Les lignes de produits dynamiques (LDPD) [HHPS08, DMFM10] ont émergés comme une catégorie spécifique des LDP qui traitent une variabilité *tardive* uniquement conçue pour répondre aux exigences qui subissent des changements lors de l'exécution. Les produits dérivés à partir d'une LDPD présentent une capacité d'adaptation dynamique sans aucune intervention ou interaction. Plus particulièrement, les LDPD visent principalement à dériver de produits configurables avec une certaine autonomie qui leur permet de s'adapter à de nouvelles situations sans avoir besoin d'arrêter leur exécution. Dans une LDPD, un produit configurable est dérivé à partir d'une ligne de produits de manière similaire à la norme de LDP. Toutefois, la capacité d'adaptation dynamique implique l'implémentation de trois mécanismes supplémentaires : [CPTC08] : (i) le *décideur*, (ii) l'*analyseur* et (iii) le *configurateur*. Le *décideur* est en charge de capturer toutes les informations liées à l'environnement qui suggère un changement (par exemple, le changement dans les informations sur les préférences des utilisateurs). L'*analyseur* doit connaître l'ensemble de la structure d'un produit afin de prendre une décision sur les fonctionnalités qui doivent être activées et désactivées. Le *configurateur* est responsable de l'exécution de la décision pour adapter dynamiquement le produit. Par conséquent, de nouvelles fonctionnalités peuvent être ajoutées à un produit existant ou même retirées lors de l'exécution. À titre d'exemple, les auteurs dans [TRCPB07] présentent un processus de construction d'une LDPD qui part d'un principe de correspondance univoque (one-to-one mapping) déjà établi entre les caractéristiques de LDPD et les composants de son architecture. Du fait, chaque caractéristique du LDPD est implémentée par un composant bien identifié, leur processus permet la dérivation de produits dynamiquement adaptables à travers une capacité d'activation et de désactivation des caractéristiques désirées lors de l'exécution. Cette adaptation se passe à travers l'implémentation d'un composant supplémentaire qui est chargé d'activer ou désactiver dynamiquement les liaisons entre les composants. Également, les auteurs dans [CFP08] présentent une LDPD basée sur le développement dirigé par les modèles et la modélisation de variabilité. Les modèles de variabilité sont interprétés au temps d'exécution pour reconfigurer les systèmes selon les fluctuations de l'environnement. Cette approche introduit le concept d'adaptation précalculée pour accélérer la réaction dans les scénarios de panne et améliorer l'autonomie de produits. L'adaptation est déclenchée lors de changements de ressources, par exemple lorsqu'une nouvelle ressource est installée ou une ressource n'est plus disponible. Ces ressources sont des capteurs, des actionneurs et de données externes. L'adaptation est également déclenchée lorsque les utilisateurs explicitement active ou désactive une fonctionnalité du produit.

Discussion par rapport à SaaS : les approches existantes proposant une mise en œuvre de l'approche LDPD [TRCPB07, BSBG08, CFP08, DMFM10] ont particulièrement mis l'accent sur l'adaptation dynamique des systèmes à un contexte d'utilisation spécifique. Ils dérivent des produits qui sont en mesure de prendre des décisions autonome sur les variantes à activer et désactiver, et ce en fonction des informations obtenues du contexte d'exécution qui leurs alertent de l'occurrence d'une variation. La majorité de points de variations d'une LDPD sont conçus d'une manière à être résolus au temps d'exécution et les artéfacts logiciels réalisant les variantes appropriées sont tous installés avec le produit dérivé pour qu'il puisse les sélectionner lors de son auto-adaptation.

Toutefois, les mécanismes d'adaptation et de configuration proposés par ces approches sont toutes orientées intrinsèquement vers la dimension de l'application et ne supportent pas le concept de locataire. Les cadres d'adaptation dynamiques orientée aspects [KRL$^+$00, CBCP02] sur lesquels la majorité des approches de LDPD se basent généralement [DMFM10] fournissent des moyens de configuration pour inspecter et adapter la structure des applications lors de l'exécution (remplacement dynamique de composants). Cependant, ces cadres se basent sur une configuration globale qui affecte tous les locataires de l'application en cas de leur utilisation dans un contexte SaaS mutualisé. Ceci nécessite d'abord leur

extension pour supporter le concept de locataire. Pour cela, d'autres cadres de développement orienté aspects tels que *JAC* [PDFS01], *JBoss AOP* [JTSJ07] et *Spring AOP* [JHA+08] ont amélioré leurs capacités de modularisation et d'adaptation dynamique par l'ajout des extensions spécifiques au niveau de l'application et au niveau de chaque utilisateur à l'aide d'une configuration déclarative. Il existe également des cadres de développement orientés aspects [TVJ+01, LJ06] qui supportent un tissage dynamique et distribué d'aspects, et qui prend en charge de multiples configurations (pour de multiples locataires en cas de leur utilisation) qui peuvent coexister dans la même instance d'application. Ces cadres sont donc adaptés à une utilisation dans un contexte SaaS mutualisé. Cependant, ils sont tous conçus pour une utilisation dans des architectures d'applications à base de composants, et ne supportent pas les applications construites à base de services.

4.2.3.2 Les lignes de produits orientées services

Les lignes de produits orientées services (LDPOS) sont des lignes de produits dynamiques [LK10, Kho12], dans lesquelles les produits sont développés selon les principes de l'architecture orientée service. Les LDPOS profitent de la flexibilité de l'AOS et de la réutilisation systématique planifié offerte par les LDP pour maximiser la production des systèmes similaires à base de services. Les LDPOS sont classées dans la même catégorie des LDPD grâce à la ressemblance observée entre leur propriétés et leur objectifs, à la différence des LDPD (et ainsi les LDP) qui sont généralement développées à base de composants. Bien qu'il existe des similitudes entre les composants et les services, il existe aussi des différences importantes qui apparaissent principalement dans leur modèles de construction. Les services sont découverts comme des éléments à boîte noire à partir des registres externes, composés de façon dynamique et faiblement couplés tandis que les composants sont fortement couplés et supporte peu de dynamisme et de flexibilité. En d'autres termes, une fois l'assemblage de composants créé, les changements dans l'architecture sont difficilement gérés au cours de l'exécution (ces changements sont restreints à l'activation et la désactivation des composants). En revanche, les services sont plus faciles à activer et à désactiver dans une application donnée, à remplacer par d'autres services fournissant une meilleur qualité et même à changer l'ordre de leur exécution d'une façon dynamique [KSSA09, CM07]. Ce niveau de flexibilité plus élevé de services permet une gestion de variabilité plus avancée et évolutive, et fait de l'architecture orientée service un modèle de construction bien adapté pour la construction des applications SaaS en général [LZV08] et mutualisées en particulier [MMLP09, GSH+07]. Toutefois, tel que dans le cas des LDPD, une LDPOS consiste à faire dériver des produits configurables à base de services séparément déployés pour chaque client. Ces produits ont la capacité d'être dynamiquement adaptés en manipulant les services qui les composent. Ils présentent ainsi une plus grande flexibilité par rapport au composants grâce au couplage faible entre les services.

Discussion par rapport à SaaS : sur les plans conceptuel et technique, à l'instar des approches de LDPD, les approches existantes d'adaptation dynamique des services (voir section 4.3) peuvent être utilisées pour adapter l'architecture orientée services d'une application SaaS mutualisée, même si ces approches n'étaient pas conçues pour ce contexte d'utilisation. Elles nécessitent d'être *légèrement* étendues pour supporter le concept de locataire. Ainsi, les mécanismes existants de substitution ou de changement de l'ordre d'exécution dynamique de services restent valides et utiles, à l'unique différence que l'existence de plusieurs locataires utilisant la même instance d'application fait augmenter le nombre de points de variation où l'utilisation d'une adaptation dynamique de services est nécessaire. Cela ne signifie pas que le nombre des points de variation (dans l'ensemble du système) sera augmenté, *mais le temps de résolution de ces points, dans leur majorité, sera retardé jusqu'au moment de l'exécution*. Vu que ceci aura

bien évidemment des impacts sur la performance du système, il y a un intérêt à mettre en place des infrastructures d'exécution puissantes qui passant facilement à une grande échelle. Sur ce point, l'utilisation des infrastructures du cloud peut apporter une solution.

4.3 Application SaaS mutualisée à base de services : approches de variabilité existantes

Les applications SaaS traitées dans cette thèse sont construites suivant le modèle de construction proposé par l'AOS (confère section 2.3). Ce choix est lié à nos besoins industriels en termes d'interactions standardisées entre nos différents systèmes distribués et hétérogènes qui peuvent être facilement réalisés à travers l'utilisation des services Web. Comme nous l'avons déjà évoqué, l'AOS est fréquemment utilisée pour la réalisation des applications SaaS [GSH+07, LZV08, MMLP09]. En effet, grâce à ses caractéristiques importantes en termes d'évolution, de flexibilité et de couplage faible entre les services interagissants, ce modèle de construction permet de réagir rapidement aux changements des besoins métiers, un aspect crucial pour la réussite d'une application SaaS. Ainsi, notre travail s'inscrit dans le cadre d'architecture d'une application orientée services. Toutefois, quand une application est construite par la composition d'un ensemble de services (suivant les principes de l'AOS), la gestion et l'implémentation de la variabilité doivent être manipulées suivant les spécifications et les artéfacts communément utilisés dans ce modèle de construction. En effet, une application basée sur l'AOS implique plusieurs couches d'abstraction (voir figure 4.7), dont la variabilité de la composition de services est uniquement un sous-ensemble du problème. La préoccupation de variabilité est plus large et apparaît dans des formes différentes avec des granularités différentes. D'après notre analyse de ce qui a été réalisé dans la communauté AOS sur la thématique de la variabilité et d'adaptation dynamique de services (pas forcément dans un contexte de SaaS mutualisé), nous avons pu constaté qu'une approche qui implémente la variabilité d'une manière générale et applicable sur toutes les couches est loin d'être réalisable. Pour cette raison, la majorité des approches proposées visent une couche spécifique, et parfois, dans un contexte spécifique. Cela a conduit à l'identification de différents types de variabilité dans ce modèle d'architecture [CK07]. Dans ce qui suit, nous présentons les types de variabilité associés à chaque couche de l'AOS et les travaux existants pour les gérer.

4.3.1 La variabilité sur la couche de processus métiers

Cette couche est responsable de l'orchestration et la composition des services pour réaliser les objectifs métiers de l'application. Typiquement, un processus métier est une fonctionnalité (à gros grains) défini par un workflow qui organise l'exécution d'un ensemble d'activités et de règles métiers [ELK+06, CK07]. Ces dernières modélisent les points de décision du processus qui contraignent l'exécution des activités spécifiques sous l'occurrence de certaines conditions. Afin de simplifier la réflexion sur la variabilité, nous considérons chaque activité du processus comme une invocation d'un service Web. Le langage BPEL (Business Process Execution Language) [JEA+07] est généralement le plus utilisé pour définir la composition de services gérée sur cette couche. Nous décrivons dans ce qui suit les types de variabilité identifiés sur cette couche et les travaux existants pour les gérer.

Types de variabilité :

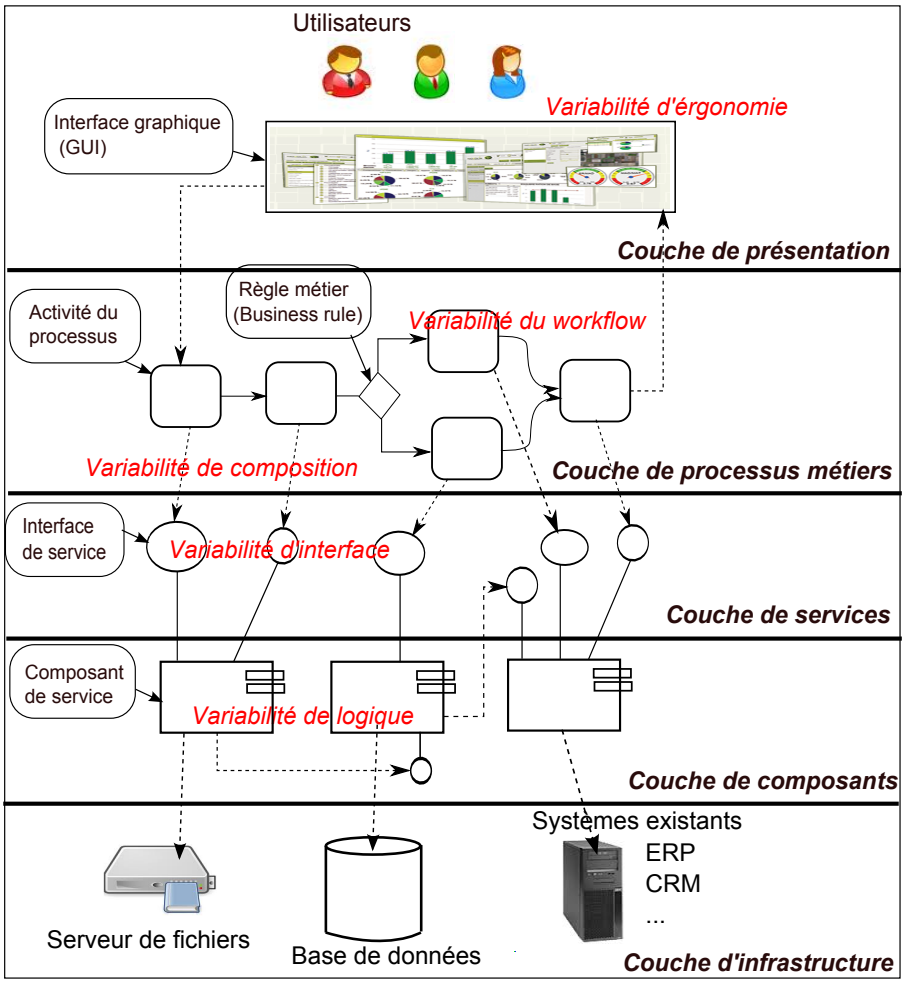


Figure 4.7 – Les couches de l'AOS [AZE+07] et les types de variabilité [CK07]

– *Variabilité du workflow :* elle correspond à la modification de la structure du processus. Cela peut être accompli à travers la désactivation des services non-désirées, le changement de l'ordre d'exécution de certains services et la manipulation de règles métiers du processus suivant les exigences des clients.
– *Variabilité de composition :* elle correspond à la modification des choix de services qui participent à la composition. Pour chaque service candidat, il peut y avoir d'autres services similaires qui supportent la même fonctionnalité mais l'implémentent suivant une autre logique d'implémentation ou une meilleur qualité. Ainsi, ce type de variabilité consiste à sélectionner les services les plus appropriés en fonction des besoins de clients. Il faut noter que le remplacement d'un service par un autre indique que les deux services sont partiellement ou totalement équivalents, et jouent le rôle des alternatifs sur un point d'invocation variable dans le processus.

Travaux existants :

– Dans [tBGFZ10], les auteurs évoquent les points communs entre les LDP et les AOS. Ils s'intéressent plus particulièrement aux préoccupations de variabilité, d'évolution et d'adaptation des systèmes dans ces deux domaines. Ils mettent en avant que l'AOS peut bénéficier des techniques de gestion des variations de LDP, dont l'objectif est de trouver des techniques de conception souples pour adapter facilement et automatiquement la composition de services aux besoins spécifiques de clients lors de l'exécution. Les solutions proposées doivent aussi permettre une adaptation des services en réaction à un changement de contexte lors de l'exécution du système. Un objectif implicite de ce travail est d'identifier les points de variabilité au moment de l'exécution dans le but d'accroître l'adaptabilité du système. Les auteurs s'intéressent aux langages de composition de services et indiquent la nécessité d'étendre ces langages en vue de traiter cette préoccupation capitale de gestion de la variabilité dynamique.
– Les travaux présentés dans [KSSA09, CK07] sont une application du besoin ressenti dans le travail précédent. Les auteurs de ces travaux essayent d'étendre le langage BPEL afin d'intégrer les informations de la variabilité (en termes de points de variations, variantes et contraintes de dépendances) dans sa définition, ce qui se traduit par une adaptation dynamique de la composition de services au moment de l'exécution du processus. Plus particulièrement, les auteurs dans [KSSA09] présentent le langage VxBPEL qui est une extension du BPEL avec laquelle les information de la variabilité sont explicitement modélisées dans la définition du processus à travers des mots clés supplémentaires.
– Le travail de [ML08] adresse les mêmes objectifs en termes d'adaptation du processus à la différence qu'il modélise la variabilité d'une manière séparée (approche non-intrusive). Les auteurs présentent une nouvelle approche de modélisation de variabilité appelée *« Variabiliy Descriptors »* à travers laquelle les artéfacts du processus sont référencés (via des *Locators*) et dynamiquement manipulés à l'exécution en fonction de chaque client. Cette approche permet, en plus de l'adaptation de la composition de services, la modification de l'ensemble du processus pour atteindre une variabilité de composition et du workflow. Ce type d'approche est intéressant car il ne complique pas la définition du processus et permet d'établir une séparation entre les développeurs qui s'occupent uniquement de la partie métier et les experts qui s'occupent de la variabilité.
– Les approches dans [SRS+10, NCH11] consistent à intégrer les informations de la variabilité dans les méta-modèles des langages de modélisation des processus métiers d'UML. Le but de ces efforts est de modéliser la variabilité à un niveau architectural, et d'automatiser ensuite la génération des processus configurables d'une manière similaire à la dérivation des produits configurables de LDPD (voir section 4.2.3.1). Plus concrètement, le travail de [SRS+10] propose un cadre spécifique à la modélisation de la variabilité des processus définies en BPMN (Business Process Modeling Notation). La variabilité est modélisée suivant le modèle COVAMOF [SDNB04] et intégrée dans le méta-modèle du BPMN à travers l'utilisation des stéréotypes. Une fois le processus est modélisé (y compris sa variabilité), le cadre automatise la génération d'un processus VxBPEL équivalent. Ce dernier est ainsi prêt au déploiement et à l'exécution sur le serveur de l'application.
– Dans [SLLW09], les auteurs proposent un mécanisme d'adaptation concernant la composition de services dans le contexte des applications SaaS mutualisées. Le système vient pallier les limitations de BPEL en termes d'adaptation dynamique, tout en permettant aux locataires de configurer le processus et de s'assurer de la validité de leurs configurations en accord avec des règles de configuration prédéfinies. Pour implémenter la variabilité du processus, les auteurs différencient entre la notion d'un service abstrait qui représente l'expression formalisée des besoins pour un service, et le service concret qui représente le service réel concrètement utilisé dans la composition. Le but de cette différenciation est de permettre une définition abstraite du processus de l'application

à concrétiser lors de l'exécution par le mécanisme proposé. Ce dernier remplace dynamiquement les services abstraits par les services concrets qui correspondent aux configurations de chaque locataire.

- Les auteurs dans [AV07] s'intéressent à l'adaptation de règles métiers. Ils proposent une méthode pour les découpler du processus de l'application et les modéliser séparément à travers l'utilisation des tables de décision. Ces tables représentent une structure commune pour décrire les règles qui varient d'un client à l'autre sous forme de méta-données facilement modifiables lors de l'exécution. Les auteurs proposent aussi un cadre sous forme d'un service à travers lequel les experts peuvent créer et modifier les règles métiers et ensuite les exposer sous forme de services Web invocables à partir du processus. Cette méthode est basée sur l'utilisation du langage RuleML [BTW01] spécifique à la publication des règles métiers sur le Web.
- Dans [SBNH05], les auteurs proposent une approche appelée *DySOA (Dynamic Service-Oriented Architecture)* qui permet de gérer et de garantir les paramètres QoS de l'application dans un environnement dynamique. Ils proposent un mécanisme qui étend les applications à base de services pour les rendre auto-adaptatives, tout en permettant aux développeurs d'explicitement modéliser les éléments qui gèrent l'évaluation de la QoS et la configuration de la composition de services. Cette approche est basée sur une modélisation explicite de la variabilité. Un point de variation dans *DySOA* définit un élément particulier de la spécification du système où une variante peut être appliquée, bien qu'elle définisse un changement comportemental ou fonctionnel. Les paramètres QoS de l'application sont constamment surveillés et évalués (qualité, disponibilité, temps de réponse, etc.). Quand une violation est détectée, *DySOA* est responsable de résoudre le point de variation correspondant pour maintenir la qualité de service dans un niveau acceptable.
- Dans [HBR08], les auteurs proposent l'approche *PROVOP (Process Variants by Options)* pour gérer une large collection de variantes du processus. Celles-ci sont formalisées à partir d'un processus de base suivant une approche opérationnelle qui applique un ensemble d'opérations de changement bien définis. *PROVOP* offre une prise en charge complète de la variabilité tout au long du cycle de vie du processus, permettant une configuration maintenable des variantes. L'idée consiste à représenter les variantes par un ensemble d'opérations de changement qui permettent la migration du processus de base vers un processus spécifique. Ces opérations sont définies en tant que modèles d'actions, où les actions consistent principalement à *insérer, supprimer* ou *déplacer* les activités du processus.

4.3.2 La variabilité sur la couche de services

Cette couche encapsule les services Web composés dans l'application. Chaque service est identifié par un fichier WSDL décrivant les fonctionnalités qu'il fourni. Les services implémentent les activités du processus. Selon [CK07], il existe un seul type de variabilité sur cette couche, celui de la *variabilité d'interface*. Ce type de variabilité se produit généralement suite à une modification de la composition de services dans la couche du processus en remplaçant un service par un autre avec une interface incompatible. Les travaux proposés pour gérer la variabilité à ce niveau consistent à identifier et à résoudre les incompatibilités syntaxiques et sémantiques sur les messages d'entrées et de sorties ainsi que les signatures des méthodes de services. Par la suite, nous allons fournir un ensemble des travaux de gestion de la variabilité à ce niveau.

Travaux existants :

– Les auteurs dans [JWY+10] s'intéressent aux incompatibilités sémantiques des interfaces de services. Plus particulièrement, ils étudient les incompatibilités des ontologies attachées aux interfaces des services réutilisés. Ils proposent de résoudre ces incompatibilités par une méthode d'extraction de ce qu'ils nomment les sous-ontologies (« *sub-ontologies* » ou « *Representation ontology* ») qui sont des expressions simplifiées des ontologies d'origine. L'emploi de ces sous-ontologies permet de simplifier le processus d'alignement d'ontologies et d'augmenter sa précision. À partir de ce travail d'extraction et d'alignement, les auteurs considèrent qu'ils sont capables de générer dynamiquement des adaptateurs pour garantir les compatibilités d'interfaces.
– Dans [KKS07], les auteurs proposent une méthode de composition de services qui cible les problèmes d'incompatibilités de messages. Leur approche traite ce problème en amont au niveau de la composition en se basant sur une organisation particulière des services disponibles. En fait, les services sont organisés dans un graphe qui représente l'ensemble des compositions possibles. Lors d'incompatibilités de données entre les services alternatifs, le système utilise le graphe des compositions pour identifier une succession de services capable d'assurer les modifications nécessaires sur les données a priori incompatibles.
– Les auteurs dans [KSPBC06] proposent un cadre basé sur la technique de programmation par aspect pour l'adaptation d'interfaces de services. Ils fournissent une taxonomie des types d'inadéquation potentiels entre les spécifications et les services disponibles. Ce cadre comprend une base de modèles d'aspects prédéfinis (aspect templates) pour automatiser la tâche de gestion d'incompatibilité. Les types d'incompatibilités identifiés sont principalement liés aux signatures de services et aux messages échangés.
– Dans [BP08, PY10], les auteurs proposent des méthodes basées sur la combinaison de techniques de planification des graphes [GNT04, CBB07] avec des descriptions sémantiques attachées aux données et aux fonctionnalités fournies par les services. Ces méthodes supportent que les auteurs appellent les adaptations horizontales (sur les données échangées entre services) et verticales (les incompatibilités entre les besoins et les fonctionnalités).

4.3.3 La variabilité sur la couche de composants

Cette couche concerne l'ensemble des composants qui implémentent les fonctionnalités des services publiés. Ils maintiennent ainsi leur qualité et assurent la conformité aux accords de niveaux de services (SLA). Ces composants sont supportés par une couche d'infrastructure qui détient les ressources de données traitées par l'application. Cette couche gère un seul type de variabilité appelé la *variabilité de la logique*. Ce type de variabilité correspond à l'adaptation de la logique d'implémentation d'un composant. Cela peut être effectué à travers le support de différents algorithmes ou méthodes de traitement de données. Il concerne les adaptations à petit grain qui sont mieux gérées à ce niveau et ne requiert pas un remplacement du service. Il faut noter qu'une telle adaptation est uniquement applicable sur les composants de services *contrôlés* par le fournisseur de l'application, c.à.d les composants qui sont développés et déployés sur sa propre infrastructure et qui lui appartiennent juridiquement. L'adaptation d'un composant du service *non-contrôlé* nécessite la négociation de cette adaptation avec son fournisseur. Par la suite, nous allons fournir un ensemble des travaux de gestion de la variabilité sur cette couche.

Travaux existants :

– Dans [KKKS11], les auteurs proposent un ensemble de patrons de conception pour implémenter la variabilité dans les composants de services. Chaque patron peut être utilisé dans une situation de variabilité différente. Les auteurs proposent un tableau comparatif des différentes solutions en

terme de respect de certaines propriétés (évolution, flexibilité, passage à l'échelle, risque, maintenance, responsabilité). Ce type d'approche est similaire au principe des patrons de conception proposés pour les langages de programmation orientés objets.
- Dans [JB09], les auteurs proposent une méthode basée sur les principes de programmation par aspects pour modulariser les préoccupations de variabilité qui affectent l'implémentation de services. Ils classifient les variantes comme étant légères ou lourdes en fonction de la nature du point de variation. Leur approche consiste à créer un noyau de service (service kernel) qui forme la partie commune entre les clients, et modulariser séparément chaque variante sous forme d'un aspect. Au moment de l'exécution, les aspects spécifiques à chaque client sont dynamiquement composés avec le noyau à travers un mécanisme de tissage d'aspects. L'adaptation concerne le comportement ainsi que les messages d'entrées et de sorties des composants.
- Les auteurs dans [SM10] proposent une approche plus générale en fournissant un méta-modèle de variabilité du service. Les clients peuvent adapter le service en résolvant son modèle de variabilité qui est conforme au méta-modèle proposé. Plus précisément, cette technique de modélisation permet de définir des éléments fixes et variables dans le service (i.e. opérations, types de messages, propriétés etc.). Chaque client doit se faire attribuer les éléments fixes (fondamental pour le bon fonctionnement de service) et ensuite adapter la structure du service en choisissant parmi les éléments variables ceux qui correspondent à ses besoins et d'ignorer le reste des éléments de sorte qu'ils ne fassent plus partie de sa configuration de service.

En résumé, la gestion de la variabilité dans les architectures orientées services est extrêmement complexe et pose un grand nombre de problématiques différentes. Nous avons détaillé un simple échantillon représentatif des approches existantes qui présentent une grande diversité dans la manière de gestion. Cette diversité des approches ne constitue pas une limitation des solutions proposées mais elle est intrinsèquement liée à la nature du défi. En fait, chacun des travaux présentés abordent des problèmes particuliers qui peuvent être prises en compte pour supporter la gestion de la variabilité dans les applications SaaS mutualisées. En plus du type de variabilité, nous faisons une comparaison de toutes ces approches en fonction d'un ensemble de critères supplémentaires (résumé dans la table 4.1). Ces critères sont les suivants :
- *Objectif :* il indique les exigences et les besoins que l'application à base de services devrait atteindre par la gestion de la variabilité.
- *Stratégie :* il indique les voies possibles pour atteindre l'objectif de variabilité compte tenu les mécanismes d'adaptation disponibles.
- *Mécanisme :* il indique les outils prévus par la plateforme sous-jacente sur les différentes couches de l'application qui permettent l'implémentation de la stratégie de variabilité adoptée.
- *Modèle de variabilité :* il indique la technique de modélisation utilisée pour décrire la variabilité de l'application sous forme d'un modèle abstrait.

Une classification et une comparaison exhaustive de toutes les approches de gestion de la variabilité dans les architectures orientées services est hors du cadre de cette thèse.

4.4 Problématique dégagée et approche suivie

La problématique de variabilité de différents types de systèmes logiciels a acquis beaucoup d'intérêt à la fois dans le milieu de la recherche et dans l'industrie. Ces dernières années, cet intérêt a été de plus en plus dirigé vers les applications AOS grâce au succès de ce modèle de construction et le manque d'un support natif de variabilité en termes de gestion. Cependant, les résultats et les orientations sont encore

insuffisants. L'une des principales limitations des approches présentées est celle de leur fragmentation : la majorité visent uniquement des problèmes spécifiques propres à un aspect particulier et sur une couche particulière de l'AOS, tandis que les relations entre les couches sont ignorées par l'isolation des solutions et leur diversité. En conséquence, cela pourra provoquer un manque d'uniformité qui risque de compliquer la réflexion sur la variabilité et de conduire à la prise de mauvaises décisions où les modifications sur une couche pouvant endommager les fonctionnalités d'une autre couche.

Au lieu de travailler sur une couche particulière en lui cherchant une solution de variabilité « ultime », nous essayons de nous focaliser sur la proposition d'une solution globale. Cette solution considère deux activités de gestion essentiels ; la *modélisation* et la *résolution* de la variabilité.

4.4.1 La modélisation de la variabilité

La modélisation de la variabilité consiste à définir un formalisme permettant d'explicitement documenter la variabilité dans l'architecture et de faciliter son traitement et sa communication. La technique de modélisation de variabilité initialement adoptée dans cette thèse est celle d'OVM. Ce choix était lié à notre besoin d'une modélisation de variabilité orthogonale indépendamment des artéfacts de l'application et de son architecture interne. Ce choix est aussi lié à la spécificité de l'AOS, comme un modèle de construction qui n'assume aucun langage d'implémentation spécifique et qui ne peut donc pas compter sur le fait que tous les langages d'implémentation éventuels peuvent offrir un mécanisme de variabilité intégré. En outre, la description des dépendances entre la variabilité sur les différentes couches de l'application est nécessaire, puisque les variations sont mieux implémentées sur une couche, voire même projetées sur plusieurs couches à la fois. Par exemple, une variation dans le processus métier de l'application peut parfois affecter son ergonomie et même nécessite le développement des nouveaux services.

D'ailleurs, les variations de l'application son habituellement exprimées d'une manière informelle et décrites selon les expériences des locataires potentiels et leur propre terminologie du domaine. Ceci peut être différent d'un locataire à un autre et ainsi différent de ce que les développeurs comprennent en termes de concepts et des couches de l'application concernées. Le manque de formalisation et l'absence d'un modèle uniforme et global de variabilité fait du choix des mécanismes et des couches les plus appropriées pour implémenter les variations, une tache non-triviale et particulièrement risquée [GTA12]. Ainsi, l'importance d'utiliser OVM dans notre contexte est de décrire les dépendances de variabilité à travers les différentes couches de l'application et d'uniformiser son expression et son traitement. Dans [GTA11], nous avons montré qu'OVM peut en principe être utilisé pour modéliser la variabilité dans les applications SaaS mutualisées développées à base de services. Nous avons profité du concept de *dépendance avec les artéfacts* (voir section 4.2.2.3) pour établir des liens explicites entre la variabilité modélisée séparément et les couches de l'application affectées par cette variabilité.

Problème : avec l'avancement sur la problématique nous avons rencontrés certaines limites d'OVM pour modéliser des nouvelles situations de variabilité qui sont principalement liées à la différence entre la structure organisationnelle d'une application SaaS mutualisé et celle d'une LDP (voir figure 4.8). Dans [Bos01, PBVDL05], différentes structures organisationnelles pour les LDP ont été étudiées et évaluées. Toutes ces structures partagent le fait que l'ingénierie d'une application doit être uniquement réalisée par la même organisation qui a réalisé l'ingénierie du domaine. Cette contrainte est due aux mécanismes de dérivation fournis par les approches de LDP, qui nécessitent généralement une intervention humaine pour assembler les composants de chaque nouveau produit et ensuite de le déployer dans l'environnement d'exécution du client. Cela est différent de la structure organisationnelle du modèle SaaS mutualisé qui sépare entre la construction du logiciel effectuée par son fournisseur d'une part, et sa configuration effectuée directement par les locataires à n'importe quel moment d'autre part. De ce fait, des

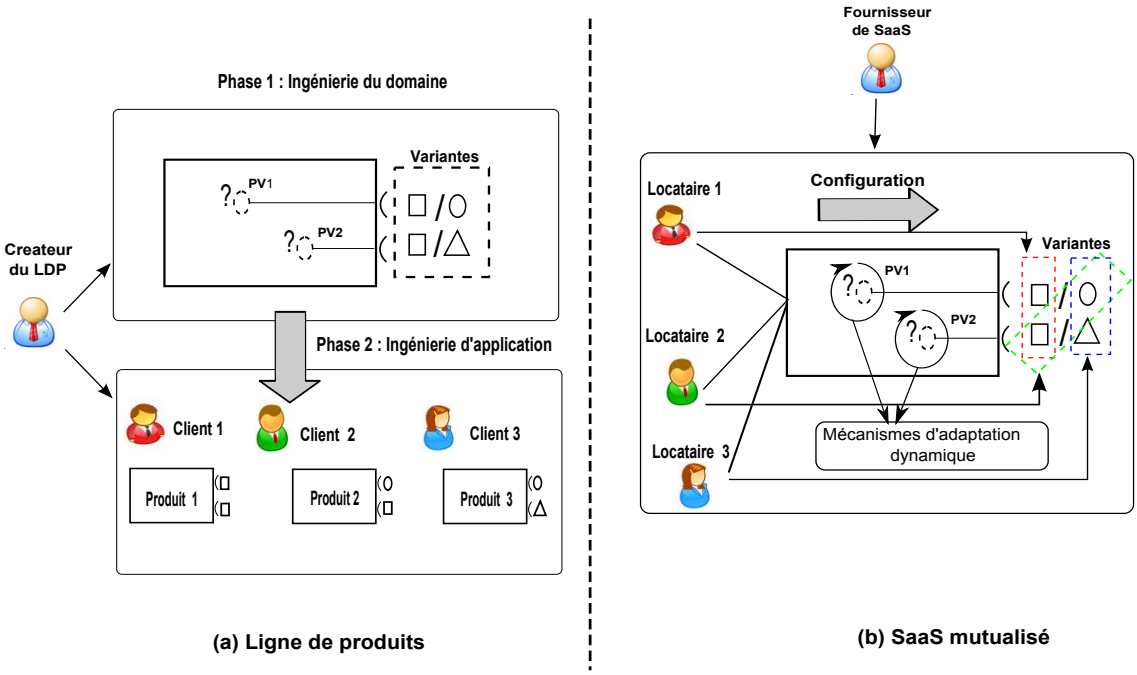


Figure 4.8 – La variabilité dans les LDP Vs. la variabilité dans les applications SaaS mutualisées

préoccupations supplémentaires doivent être gérées à travers une approche de modélisation telles que le *contrôle d'accès* au modèle de variabilité (par exemple, l'interdiction de sélection de certaines variantes par les locataires non autorisés) et son *auto-description* pour permettre sa manipulation directement par les locataires.

D'ailleurs, la configuration de l'application exige non seulement de choisir parmi des variantes prédéfinies, mais également des variantes avec des valeurs libres pouvant être saisies. Il s'agit par exemple de donner la possibilité aux locataires d'indiquer eux-même le titre de l'application ou bien sa couleur de l'arrière plan (variabilité de l'ergonomie) sans être toujours obligés de choisir parmi des variantes prédéfinies. Ces différentes préoccupations (et bien d'autres que nous détaillons plus tard dans ce manuscrit, voir section 6.2.4) n'étaient pas prises en compte par OVM, étant donnée que sa conception part du principe que la dérivation d'un produit ne pourra être effectuée que par les membres de l'organisation qui a construit la LDP.

Approche suivie : pour couvrir ces limites d'OVM, un nouveau modèle de variabilité orthogonale est introduit dans cette thèse (voir section 6.3). Ce modèle étend OVM avec de nouveaux concepts de modélisation lui permettant de répondre aux besoins émergents des applications SaaS mutualisées, tout en conservant la même méthodologie de réflexion sur la modélisation de la variabilité principalement établie par OVM. Cela faciliterai la transition vers le cloud et le support de ses caractéristiques essentielles telles que la mutualisation et le self-service pour les applications existantes.

4.4.2 La résolution de la variabilité

La résolution de la variabilité est une activité essentielle à réaliser lors de la gestion de la variabilité. Dans un contexte SaaS mutualisé, cette activité consiste à concevoir et mettre en œuvre les stratégies

et les mécanismes nécessaires pour dynamiquement adapter l'application sur ses différentes couches d'abstraction en fonction du modèle de variabilité prédéfini et l'identité du locataire connecté. Ceci dit, nous précisions que les approches déjà présentées sur les différentes couches de l'AOS gèrent une variabilité *explicite*, c.à.d. elles définissent avec précision un moment donné dans l'exécution où les changements d'adaptation doivent avoir lieu, et ce à travers la sélection des variantes du comportement prédéfinies et applicables lorsque le changement est nécessaire. D'autres approches d'adaptation de services [FMF05, CT07, BGP07, PR07] considérées comme *implicite* gèrent une variabilité liée aux changement dans l'environnement d'exécution où des anomalies d'exécution (par exemple, l'indisponibilité de services en raison des problèmes des réseaux ou des serveurs) peuvent avoir lieu. Étant donnée que ces anomalies peuvent se produire assez souvent et de façon inattendue dans toute application développée à base de services sans aucune relation avec la notion de différence entres les exigences de clients, ils ne sont pas considérées dans le cadre de cette thèse. Notre travail traite uniquement une variabilité des exigences explicitement modélisée (au moment de la conception de l'application), et une adaptation dynamique de services en fonction du modèle de variabilité et des configurations des locataires connues a priori.

Problème : un défi à notre sens pour la résolution de la variabilité est l'hétérogénéité croissante des plateformes et des technologies utilisées pour implémenter les services, et ainsi leur composition dans une application donnée. Bien que le langage BPEL et sa technologie associée sont actuellement les plus utilisés pour implémenter et exécuter une telle composition, de nouveaux langages commencent à être de plus en plus acceptés dans le monde industriel et scientifique tels que Orc [KQCM09], JOLIE [MGZ07] et BPML [TSPB02]. De ce fait, une solution de résolution de variabilité pourra avoir tendance à exploiter ces différentes technologies et pourra être élaborée en fonction pour profiter de leur caractéristiques et leur opportunités [GER08]. Cela est généralement préférable surtout lors de l'utilisation d'une technologie avancée car elle permet de réduire les efforts et d'aboutir à une solution de résolution sûre.

Cela dit, cette démarche crée un couplage fort avec la technologie et rend impossible de réutiliser les mêmes solutions de variabilité en dehors du contexte dans lequel elles ont étés initialement développées. Cela explique leurs différence d'une couche à une autre et ainsi d'une application à une autre. La situation peut devenir plus compliquée à gérer si les services de l'application sont distribués à travers différents fournisseurs, adoptant chacun une solution de variabilité différente, non seulement au niveau des mécanismes et stratégies d'adaptation, mais aussi au niveau du modèle de variabilité. En effet, la composition des services variables exige de composer leur modèles de variabilité sous forme d'un modèle unique devant être exposé à tous les locataires. Dans le cas où les techniques de modélisation sont différentes entre les services, un tel modèle est difficile à former puisqu'il nécessite d'unifier des modèles de variabilité sémantiquement incompatibles. Par conséquent, il devient très difficile de contrôler la variabilité dans toute l'application et de veiller à ce que les résultats de l'adaptation fonctionnent comme prévu. Cela est le plus souvent remarqué lorsque les variabilités des services composés dans une application présentent des contraintes de dépendance qui nécessitent d'être explicitement modélisés pour éviter une configuration incorrecte et ainsi un comportement erroné.

Approche suivie : pour contrôler la variabilité face à l'hétérogénéité, nous devons adopter une solution de gestion standardisée. Ainsi, les mécanismes d'adaptation doivent être découplés de ce que nous appelons le *cadre de variabilité*. Ce dernier gère l'infrastructure nécessaire au bon fonctionnement des mécanismes et fournit un ensemble de composants génériques et réutilisables quelque soit les technologies utilisées. Parmi ces composants nous pouvons citer ceux qui sont dédiés à la modélisation de la variabilité, la persistance des choix de variantes effectués par les locataires, la création de méta-données nécessaires à l'invocation dynamique des artéfacts variables, la gestion des incompatibilités entre les

services, etc. De notre point de vue, la raison principale des limites des approches existantes réside dans l'absence d'un tel découplage. Si nous poussons ce raisonnement plus loin, nous pouvons réaliser qu'il n'y a aucune obligation à ce que le cadre de variabilité soit déployé dans la même infrastructure que celle utilisée pour déployer l'application. Ce cadre peut être déployé sur une infrastructure externe et utilisé à travers le Web non seulement par une seule application, mais aussi par plusieurs applications en même temps, et plus particulièrement, sous forme d'un service. De ce fait, le concept de variabilité sous forme d'un service (VaaS) a été élaborée dans cette thèse (voir section 6.4).

4.5 Conclusion

Les applications SaaS mutualisées présentent un défi sur l'applicabilité et l'adaptabilité de leur services. Étant donnée que ces applications ne visent pas seulement un ensemble de locataires prédéfinis (mais plutôt de nombreux locataires potentiellement inconnus), elles doivent être hautement adaptables pour satisfaire les exigences différents et variables de chaque locataire. Par conséquent, la variabilité doit être soigneusement gérée dans ce type d'applications en considérant les spécifications de l'architecture orientée services comme un modèle de construction d'application bien adapté à SaaS.

Dans ce chapitre, nous nous sommes basés dans nos activités de gestion de la variabilité sur une nouvelle voie de recherche qui a émergé dans la communauté de SaaS, consistant à réutiliser des concepts de modélisation et de gestion employés avec succès dans le domaine d'ingénierie des lignes de produits logiciels pour soutenir la variabilité dans les applications SaaS mutualisées. Nous avons exploité cette voie de recherche en fournissant tout d'abord une explication globale des concepts fournis par le domaine de LDP, et ensuite en montrant et classifiant un ensemble des approches qui réutilisent ces concepts pour adapter les applications construites à base de services. Les limites des approches existantes concernant la modélisation et la résolution de la variabilité ont étés identifiées et brièvement documentées. Ces limites sont la base de développement de notre nouvelle approche d'externalisation de gestion de la variabilité sous forme d'un service, et la proposition d'un nouveau méta modèle de variabilité pour supporter cette externalisation.

Table 4.1 – Classification des approches de gestion de la variabilité dans les applications à base de services

	Types de variabilité				Objectif	Stratégie	Mécanisme	Modèle de variabilité
	W	C	I	L				
[KSSA09]	X	X			Adaptation du Comportement	Substitution de services	Extension du langage BPEL (VxBPEL)	COVAMOF
[SM10]			X	X	Adaptation du Comportement	Modification de la structure du service	Génération du code	Modèle spécifique
[ML08]	X	X			Adaptation du comportement	Modification du fichier contenant le code BPEL	accès réflexif au fichier du code à travers le langage XPath	Variability Descriptors
[SRS+10]		X			Adaptation du comportement	Substitution de services	Extension du modèle BPMN à travers les stéréotypes, et Génération du code VxBPEL	COVAMOF
[KKS07]		X	X		Gestion d'incompatibilité sémantique	Identification d'une succession de services qui remplacent le service incompatible	Non-précisé	-
[SLLW09]	X	X			Adaptation du comportement	Séparation entre service abstrait et service concret	Non-précisé	Modèle spécifique non-formalisé
[JB09]				X	Adaptation du comportement	Séparation entre le cœur du service (Service Kernel) et la partie variable du service	Composition et tissage des aspects	-

	Types de variabilité				Objectif	Stratégie	Mécanisme	Modèle de variabilité
	W	C	I	L				
[KKKS11]				X	Adaptation du comportement	Un patron de conception pour chaque situation de variabilité	Non-précisé	-
[AV07]	X				Adaptation du comportement	Table de décisions	Utilisation du langage RuleML	-
[JWY+10]			X		Gestion d'incompatibilité sémantique	Extraction de sous-ontologies	Génération dynamique des adaptateurs	-
[SBNH05]		X			Qualité (QoS)	Substitution de services	Non-précisé	OVM
[HBR08]	X				Adaptation du comportement	Modification de la structure du processus	Opérations de changement	PROVOP
[MRD08]		X	X		Adaptation du comportement	Substitution de services	Interception des appels de services à travers des aspects	-
[BP08]			X		Gestion d'incompatibilité sémantique	Description de sémantiques et planification de graphe	Non-précisé	-

Deuxième Défi abordé : l'Isolation des Données de Locataires

5.1 Introduction

Nous abordons dans ce chapitre un autre défi de la mutualisation de SaaS représenté par le besoin d'isoler les données de locataires. Ce défi concerne principalement la construction d'un système d'accès aux données qui soit *sécurisé* et offrant une *flexibilité de personnalisation de données* maximale [CCW06], car il sera responsable de gérer l'accès aux données privées et hautement sensibles de locataires. Le système doit être capable de supporter ces exigences, tout en étant aussi efficace et rentable à administrer et maintenir. C'est seulement dans ce cas que les locataires seront satisfaits de confier le contrôle de leurs données vitales à un fournisseur tiers.

Dans ce chapitre, nous présentons un ensemble des techniques fondamentales de mise en œuvre typique d'une base de données mutualisée, celles que nous avons identifié à travers l'analyse d'un ensemble de travaux de recherches traitant ce défi. Ce chapitre est organisé de la manière suivante : la section 5.2 présente les différents alternatives de conception d'une base de donnée mutualisée qui traitent les exigences principales de la mutualisation à ce niveau. Ensuite, la section 5.3 présente des services de gestion de données actuellement disponibles sur le marché de services du cloud qui répondent aux différentes exigences de la mutualisation de données et sont largement répandus et utilisés par les fournisseurs de SaaS. La section 5.4 montre les problèmes dégagés d'après l'analyse de ces services et prépare le terrain pour notre contribution. Enfin, la section 5.5 conclut le chapitre.

5.2 Isolation des données de locataires dans une base de données mutualisée

Un défi particulièrement important dans une application SaaS mutualisée est celui d'assurer la mise en place de cette mutualisation sur la couche de données [LHX12]. En d'autres termes, le défi consiste à consolider les données de multiples locataires sur une ressource de donnée unique en les isolant entre elles de telle sorte qu'un locataire ne pourra jamais avoir l'accès aux données des autres locataires. La nature de cette mutualisation diffère de celle généralement adoptée par les fournisseurs de bases de données sous forme des services (DaaS, voir sous-section 2.2.6) [HIM02]. Selon ce concept, le partage entre les clients (des fournisseurs de SaaS dans ce cas) est limité aux ressources matérielles (serveur, disque, mémoire, etc.) où chacun a son propre système de gestion de base de données (SGBD) isolé (physiquement ou virtuellement) des autres clients. La nature de mutualisation traitée dans ce chapitre suppose que tous les locataires de l'application partage le même SGBD et probablement une instance unique de base de données sur ce SGBD car les modes d'accès et les structures de données sont les mêmes pour tous locataires de l'application.

En fait, la capacité de gérer nativement la mutualisation n'est pas une caractéristique commune des SGBD classiques. Le vrai défi pour la création et la mise en œuvre d'une telle capacité consiste en la conception et l'implémentation d'un système d'isolation de données qui soit suffisamment sécurisé et robuste, et offrant une flexibilité de personnalisation de données maximale [CCW06]. Cela revient principalement à (i) séparer les données de locataires, (ii) personnaliser le schéma de la base de données *complètement* partagée entre eux et (iii) sécuriser l'accès à leurs données. Dans la suite, nous allons nous concentrer sur la première exigence (la séparation de données de locataires) et présenter les différents modèles de séparation que nous avons identifié dans l'état de l'art et les différentes considérations qui influent le choix d'un modèle de séparation : cela constitue une première étape à effectuer avant d'aller plus loin dans la mise en place d'un système d'isolation de données de locataires dans les applications SaaS mutualisées.

5.2.1 La séparation des données de locataires

La question de passage à l'échelle est celle la plus préoccupante dans ce contexte. D'ailleurs, même si la valeur fondamentale de la mutualisation est celle de réduire les coûts de maintenance par la consolidation de tous les locataires dans une instance de base de données unique, la pratique récente montre qu'une telle consolidation dégrade la performance de l'application et pose de vrais problèmes de sécurité et d'extensibilité [AGJ+08]. De ce fait, il existe des divers degrés de séparation de données qui varient d'un environnement complètement séparé vers un environnement complètement partagé [HJLZ09, JA+07, WGG+08]. Les manières de mise en œuvre le long de ce spectre, incluent principalement trois modèles de séparation illustrés dans la figure 5.1. Nous allons expliquer ces différents modèles dans la suite.

5.2.1.1 Les différents modèles de séparation

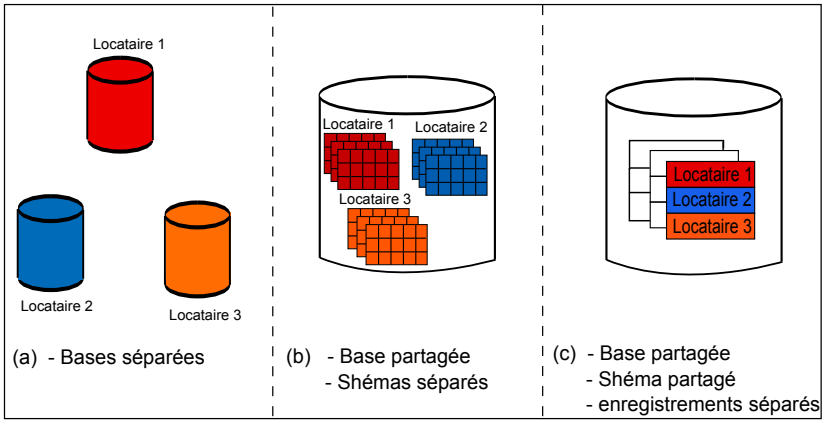


Figure 5.1 – Modèles de séparation de données

1. **Le modèle de bases séparées :** dans ce modèle (voir Figure 5.1 (a)), chaque locataire a sa propre base de données. Ce modèle est le plus facile à réaliser, et dans lequel tous les locataires partagent les mêmes ressources du SGBD. Il est important de noter à ce niveau que même s'il s'agit d'une

solution de séparation de données physique, cela n'empêche pas de maintenir la mutualisation du code de l'application. Des méta-données (définies et stockées dans une base de données spécifique) sont associées à ce modèle qui ont pour objectif de faire la correspondance entre chaque locataire et la bases de données qui lui a été réservée (ex : Locataire1⇒Base1, Locataire2⇒Base3, Locataire3⇒Base2). Ce modèle peut être entièrement construit à l'aide d'un SGBD classique sans avoir besoin d'effectuer des modifications de l'application. Il est ainsi relativement facile d'étendre le modèle de données pour chaque locataire quand les différentes bases de données sont entièrement séparées. De plus, le rétablissement après une défaillance pourra être effectuée avec les techniques classiques de restauration de données.

Cependant, cette approche entraine un coût de maintenance énorme. 1000 locataires provoquent 1000 instances de bases de données à maintenir et à mettre à jour séparément. Cela implique aussi un grand travail de configuration à effectuer pour chaque nouvelle base, où le fournisseur de SaaS doit allouer de l'espace disque, de la mémoire et de la bande passante du réseau et il doit aussi spécifier les paramètres de connexion et la liste de contrôle d'accès (si il en existe). De ce fait, le passage à l'échelle dans cette approche est plutôt difficile puisqu'une consommation inutile de ressources humaines et matérielles pourrait avoir lieu. Les coûts de matériels sont également les plus élevés, étant donné que le nombre de locataires qui peuvent être hébergés est limité par le nombre de bases de données qu'un SGBD peut prendre en charge. Lors de dépassement de ce nombre, un nouveau SGBD doit être installé.

Cette approche est la plus pertinente pour les locataires qui sont prêts à payer plus pour plus de sécurité et de personnalisation de données. Par exemple, les clients dans des domaines tels que la banque ou la gestion des dossiers médicaux ont souvent de très fortes exigences relatives aux données et ne peuvent pas envisager une application qui ne fournit pas à chacun d'entre eux sa base de données spécifique.

2. **Le modèle de base de données partagée et schémas séparés :** dans ce modèle (voir figure 5.1 (b)), toutes les données des locataires sont hébergées dans une instance de base de données unique, tandis que chaque locataire a ses propres tables et son propre schéma au sein de cette base de données unique. Dans ce cas, le fournisseur de service maintient une seule base de données en lui associant toutes les capacités dont il dispose en termes d'espace disque, de mémoire et de bande passante du réseau. Lors de l'inscription d'un nouveau locataire, le système crée un ensemble de tables formant son schéma de données privé. De ce fait, si un locataire décide de changer son modèle de données pour des besoins spécifiques, les autres locataires ne seront pas touchés parce qu'ils ont leurs schémas séparés. Ce modèle offre un meilleur passage à l'échelle que celui du modèle précédent et réduit l'énorme coût de maintenance provoqué par la gestion des bases de données distinctes. Chaque nom de table dans ce modèle doit joindre l'identifiant de son locataire afin de pouvoir différencier les schémas. L'application à son tour devra implémenter un mécanisme de transformation de requêtes pour rectifier les noms des tables dans chaque requête émise et assurer l'interaction avec les bonnes tables. Par exemple, pour obtenir tous les enregistrements de la table `SalesOrder`, la requête saisie par un développeur est la suivante :

```
SELECT * FROM SalesOrder
```

Pour interagir avec la table du locataire connecté (le locataire 123 dans ce cas), la transformation doit générer la requête suivante :

```
SELECT * FROM SalesOrder-L123
```

Cette transformation devra être effectuée d'une manière dynamique à travers des fonctions de réécriture de requêtes. L'identifiant du locataire pourra être récupéré à partir de la session associée à l'utilisateur connecté, dans lequel les informations liées à son locataire sont généralement sauvegardées. La séparation des sessions des utilisateurs peut être gérée par le serveur de l'application qui permet des connexions multiples.

À l'instar de l'approche précédente, cette approche est relativement facile à mettre en œuvre, et le fournisseur peut étendre le modèle de données de chaque locataire aussi facilement. Les tables sont créées à partir d'un schéma par défaut, mais une fois qu'elles sont créées, elles n'ont plus besoin de se conformer à ce schéma. Cette approche offre un degré modéré de séparation logique de données pour les locataires soucieux de la sécurité, mais pas autant que le modèle de bases séparées le serait.

Toutefois, le nombre de tables privées dans ce modèle augmente de façon linéaire avec le nombre de locataires. Par conséquent, le passage à l'échelle est limité par le nombre de tables qu'une base de données peut supporter, nombre qui est lié à sa mémoire disponible. Par exemple, la version 5 de *MySQL* (un SGBD assez utilisé) alloue 9 K octets de mémoire pour chaque table créée. Ainsi, 100.000 tables occupent 900 M octets de mémoire, ce qui est un volume assez considérable. Ainsi, il a été mis en avant qu'une dégradation significative de la performance de ce SGBD lorsque le nombre de tables s'élève au-delà de 50.000 [AGJ+08]. Cela veut dire que pour une application SaaS qui gère environnement 200 tables, le nombre maximal de locataires ne pourra pas dépasser le nombre de 250. Plus de tables à ajouter implique une baisse du nombre de locataires que le système peut supporter.

Un autre inconvénient important de cette approche réside dans la difficulté de restaurer les données de locataires en cas de panne. Si chaque locataire a sa propre base de données, la restauration des données d'un seul locataire signifie tout simplement la restauration de la base de données à partir de sauvegarde la plus récente. Dans ce modèle, la restauration de la base de données entière signifie un écrasement de données de tous les locataires sur cette base indépendamment de ce qu'ils ont connu comme perte de données. Par conséquent, pour restaurer les données d'un locataire unique, l'administrateur de la base de données doit restaurer la base de données à un serveur temporaire puis importer les tables de ce locataire dans le serveur de production, une tâche compliquée et chronophage.

Ce modèle est approprié pour les applications qui utilisent un nombre relativement restreint de tables de bases de données, de l'ordre d'environ 100 tables ou moins. Cela permet généralement d'accueillir plus de locataires par SGBD que l'approche précédente et d'offrir ainsi le service à moindre coût aussi longtemps que les locataires acceptent que leurs données soient partagées dans une base de données unique.

3. **Le modèle de base de données partagée, schéma partagé et enregistrements séparés :** dans ce modèle (voir figure 5.1 (c)), tous les locataires partagent la même base de données et ils partagent également leurs données dans le même schéma. Pour séparer leurs données, chaque table doit être étendue avec une colonne supplémentaire qui détient l'identité du locataire sur chaque enregistrement ajouté. Cette approche ne pose aucune limite sur le nombre de locataires que le système peut supporter et offre le coût de maintenance le plus bas. De plus, les opérations administratives supplémentaires ou la modification des tables peuvent être exécutées et testées pour l'ensemble des locataires à la fois. Cela est nettement plus rapide par rapport aux modèles précédents et, donc il est clair que ce modèle assure le meilleur passage à l'échelle.

Au niveau de restauration des données pour un locataire, ce modèle exige une méthode de restauration similaire à celle du modèle précédent, avec une complication supplémentaire celle des enregistrements spécifiques à ce locataire dans la base de données de production, qui doivent être tout d'abord supprimés et puis réinsérés à partir de la base de données temporaire. Toutefois, s'il y a un très grand nombre d'enregistrements dans les tables concernées par la restauration, la performance de la base de données sera impactée, ce qui risque d'affecter tous les locataires hébergés. Ce modèle est approprié quand il est important que l'application soit capable de servir un grand nombre de locataires avec le minimum de ressources possibles et que les locataires potentiels sont prêts à accepter une sécurité « moins garantie » en échange avec de moindres coûts.

Malheureusement, ce modèle soulève principalement trois problématiques. (i) Premièrement, le logiciel et sa base de données sont en évolution continue. Dans certains cas, cette évolution peut être destinée à uniquement satisfaire les besoins spécifiques d'un locataire précis et ne impacter pas l'ensemble de locataires hébergés. Donc, la question qui se pose ainsi est comment procéder pour faire évoluer un schéma de données partagé selon les besoins de chaque locataire ? (ii) Deuxièmement, comment sécuriser l'accès aux données et assurer qu'un locataire ne pourra jamais accéder aux données des autres, les requêtes à ce niveau doivent subir des transformations plus complexes qui sont difficiles à gérer et à vérifier. (iii) Finalement, comment garantir que la dégradation des performances du système est dans les limites tolérables et que finalement la consolidation de toutes les données de locataires dans les mêmes tables ne ralentie pas considérablement le temps d'exécution.

5.2.1.2 Le choix d'un modèle de séparation

Chacun des trois modèles décrits précédement offre ses propres avantages et les compromis qui en font un modèle approprié à suivre dans certains cas mais pas dans d'autres. Cela est déterminé par un certain nombre de considérations économiques et techniques. Ces considérations sont plus détaillés ci-dessous :

Considérations économiques : les applications utilisant un modèle de données complètement partagées (modèle 3) exigent un effort de développement plus important que les applications conçues à l'aide d'une approche de données complètement séparée (modèle 1 ou 2), ce qui entraîne une hausse des coûts initiaux. Mais vu que le partage permet de supporter plus de locataires par base de données, les coûts de maintenance permanents pour le modèle complètement partagé (modèle 3) ont tendance à être plus faible avec le temps (voir figure 5.2).

L'effort en développement peut être limité par des facteurs économiques qui peuvent influencer le choix de modèle. L'approche du modèle complètement partagé (modèle 3) peut finir par être moins coûteuse sur le long terme, mais elle exige un effort plus large de développement initial avant de pouvoir commencer à générer des revenus. Si le fournisseur de SaaS ne parvient pas à financer un effort de développement de la taille nécessaire pour construire ce modèle, ou s'il y a une obligation de présenter l'application sur le marché plus rapidement que le temps nécessaire pour un développement à grande échelle le permettrait, le choix du modèle complètement séparé (modèle 1) devient plus pertinent.

Considérations de sécurité : vu que l'application doit stocker des données sensibles des locataires, ces derniers auront légitimement de grandes attentes quant à la sécurité. Une méprise commune veut que seule l'approche du modèle complètement séparé peut fournir le niveau de sécurité approprié. En fait, les données stockées à l'aide d'un modèle complètement partagé peuvent également être

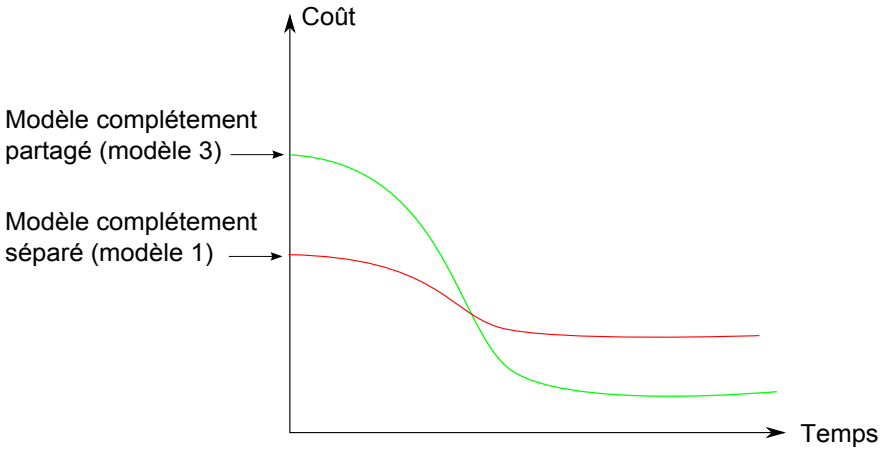


Figure 5.2 – Coût de gestion au fil du temps de la base de données pour une paire hypothétique d'applications SaaS en fonction du modèle de séparation de données adopté [CCW06]

sécurisées, mais cela nécessite l'utilisation de mécanismes de sécurisation plus sophistiqués et donc cette approche est plus coûteuses. Toutefois, les organisations de locataires sont souvent soumises à des lois et des réglementations qui peuvent affecter le niveau de sécurité requis celui qui favorise le modèle complètement séparé à celui complètement partagé. Une investigation sur les environnements réglementaires qui concernent les locataires potentiels dans les marchés sur lesquels un fournisseur de SaaS souhaite fonctionner, peut influer la décision.

Considérations de compétence : la mise en place d'un modèle de séparation de données complètement partagé est une nécessité très récente, dont l'expertise et la maitrise peuvent être difficiles à trouver. Si les développeurs n'ont pas beaucoup d'expérience dans la manière de gérer ce modèle, ils ont besoin d'acquérir les connaissances nécessaires, ou d'embaucher des personnes ayant déjà développé de telles compétences. Dans certains cas, une approche de bases séparées peut permettre au développeurs de profiter de leurs connaissances actuelles.

Après avoir présenter les différents modèles de séparation et les considérations qui influent le choix d'un modèle, nous pouvons remarquer que ce choix dépend non seulement de l'application mais aussi d'autres facteurs économiques et techniques qui varient selon la situation actuelle du fournisseur et celle du marché. Dans la suite, nous allons détailler les exigences supplémentaires à gérer dans un système d'isolation, découlant du choix du modèle complètement partagé (modèle 3) puisqu'il est souvent considéré comme le meilleur choix de modèle de séparation de données dans une application SaaS mutualisée [CCW06, WGG+08]. Ces exigences reviennent essentiellement à gérer la personnalisation du schéma partagé de la base de données entre les différents locataires et la sécurisation d'accès à leurs données. Nous allons détailler chacune de ses deux exigences dans les sous-sections suivantes.

5.2.2 La personnalisation du schéma partagé de la base de données

L'évolution d'une base de données dans un environnement complètement partagé nécessite la mise en place d'une méthode de personnalisation du modèle de données pour assurer la flexibilité du schéma. De toute évidence, pour les deux premiers modèles de séparation présentés (modèles 1 et 2), la personnalisation ne pose pas un problème puisque les locataires ont leurs données dans des schémas séparés.

La modification du modèle de données pour un locataire peut être directement effectuée dans sa propre base de données ou dans son propre schéma sans impacter les autres locataires. Toutefois, lorsque tous les locataires partage le même schéma de données, il est nécessaire de mettre en place une méthode de personnalisation de ce schéma en fonction des besoins spécifiques de chaque locataire. Dans la section précédente, nous avons expliqué que la mutualisation des données et la personnalisation du schéma qu'elle exige ne sont pas des fonctions nativement supportées dans les SGBD traditionnels. Plus difficile encore, les SGBD traditionnels interdisent les modifications fréquentes des schémas car les opérations de modification peuvent négativement influer sur la disponibilité du système. Pour ces raison, et à titre d'exemple, la plateforme force.com a construit sa propre solution de personnalisation de données qui est entièrement gérée à partir de l'application (voir section 5.3.1 pour plus de détails). Malgré son sucées, cette approche réduit les capacités du SGBD à un support de stockage simple et muet sans connaissance de la sémantique des relations entre les données. Ce choix complique aussi le développement puisque de nombreuses fonctionnalités de SGBD telles que l'optimisation des requêtes et l'indexation de données ne sont plus exploitables. Ainsi, des travaux de recherche récents [Aul11, DT10] ont conclu que la nouvelle génération de SGBD doit nativement gérer, à travers la fourniture d'un support spécifique, la personnalisation du schéma et même le concept de locataire pour éviter une telle complexité.

Dans l'attente, différentes méthodes de personnalisation faisant face à cette préoccupation ont été fournis dans l'état de l'art [GSK+08, FDFI09, HJLZ09, XQL10]. Ces solutions consistent généralement à réaliser une sorte de correspondance entre les besoins spécifiques des locataires et les techniques permises dans le monde relationnel. Bien évidemment, ces approches ne perdent pas de vue que l'objectif final d'un SaaS mutualisé consiste à profiter de l'existence des points communs entre les locataires, et que le nombre de ces points reste suffisamment élevé pour justifier le partage. Pour cela, ces approches proposent différentes méthodes de personnalisation tout en gardant, dans les plupart des cas, un schéma de base qui maintient une structure de données commune. Il est donc important de différencier à ce niveau entre le *schéma physique*, celui qui caractérise l'état réel du schéma de données, et le *schéma logique* qui est spécifique à chaque locataire et qui est reconstruit dynamiquement à partir du schéma physique lors de l'accès à leurs données. Toutefois, il existe divers degrés de personnalisation de données qui varient d'un simple ajout de colonnes supplémentaires jusqu'à la personnalisation complexe de l'ensemble du schéma. Dans la suite, nous allons détailler les méthodes de personnalisation qui sont le plus fréquemment utilisées dans une approche de schéma de données partagé (modèle 3) :

Les tables d'extension : dans cette approche, les données spécifiques aux locataires sont verticalement divisées dans des tables séparées qui sont reliées avec les tables communes à travers la colonne `Row`. Cette colonne détient une identité spécifique à chaque enregistrement pour récupérer les données de personnalisation du locataire sur cet enregistrement (voir figure 5.3). La table de métadonnées nécessaires à cette approche définissent les types de données d'extension (ou de personnalisation) et gèrent une colonne supplémentaire `ExtensionID` car plusieurs locataires peuvent utiliser les mêmes types d'extension. Par exemple, le *locataire1* et le *locataire2* utilisent le même type d'extension *CustomerName*. Toutefois, la reconstruction des tables logiques à travers cette approche porte la surcharge des jointures supplémentaires ainsi que des Entrées/Sorties pour récupérer les fragments des enregistrements dispersés à travers les différentes tables (communes et extensions). Cette approche a été initialement développée à partir du modèle de stockage décomposé décrit dans [CK85], et généralement utilisée pour la mise en correspondance d'un modèle orienté objet contenant de l'héritage avec un schéma de base de données relationnelle.

Les champs de réservation : dans cette approche, chaque table du schéma physique est composée d'une colonne réservée à l'identité du locataire, d'un ensemble de colonnes communes entre tous les lo-

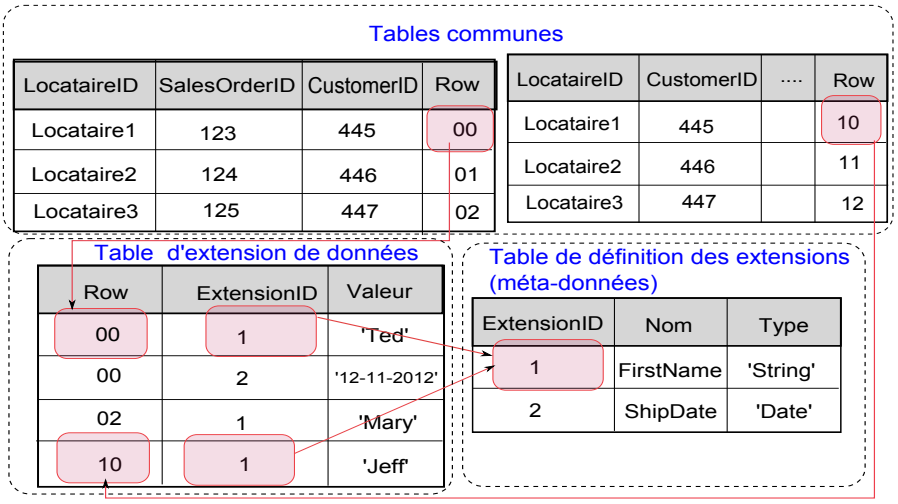


Figure 5.3 – Personnalisation de données à travers les tables d'extension

cataires et d'un ensemble de colonnes spécifiques (voir figure 5.4). Ces dernières ont un type de données *flexible*, tel que `VARCHAR`, dans lequel d'autres types de données peuvent être convertis.

LocataireID	SalesOrderID	....	C1	C2	C3	C4
Locataire1	123		'12-11-2012'	'1.8'	'Ted'	*Null*
Locataire2	124		'False'	*Null*	*Null*	*Null*
Locataire3	125		'Mary'	*Null*	*Null*	*Null*
Locataire1	126		'13-11-2012'	'2.1'	'Alex'	*Null*

Colonnes supplémentaires

Figure 5.4 – Personnalisation de données à travers les champs de réservation

Selon cette approche, chaque donnée spécifique pour un locataire est stockée dans le premier champ de données disponible dans les colonnes supplémentaires, et par conséquent, les différents locataires peuvent étendre la même table de différentes manières. Le fait de maintenir l'ensemble des données communes et spécifiques dans les mêmes tables en même temps permet de faciliter la reconstruction des tables logiques d'un locataire pour les traiter dans son contexte d'exécution. Cette approche est la plus simple à réaliser mais elle présente un inconvénient évident celui que les enregistrements doivent être très larges et la base de données doit gérer de nombreuses valeurs `NULL`. Malgré que les SGBD gèrent les valeurs `NULL` de façon assez efficace, ils occupent néanmoins un espace mémoire supplémentaire et nuisent à l'efficacité du traitement des requêtes [XQL10]. L'application doit effectuer une projection explicite sur les colonnes spécifiques qui intéressent le locataire (plutôt que de faire un `SELECT *`) afin de s'assurer que les valeurs `NULL` ne font pas partie des résultats.

Les tables croisées : dans cette approche, tous les schémas logiques de locataires sont mis en correspondance avec une seule structure générique dans la base de données. Chaque champ de chaque enregistrement dans une table logique est représenté par un enregistrement dans la table croisée (table physique). Celle-ci gère uniquement des méta-données et possède une colonne `Col` qui spécifie la colonne de la table logique à laquelle l'information de l'enregistrement physique est associée (voir figure 5.5).

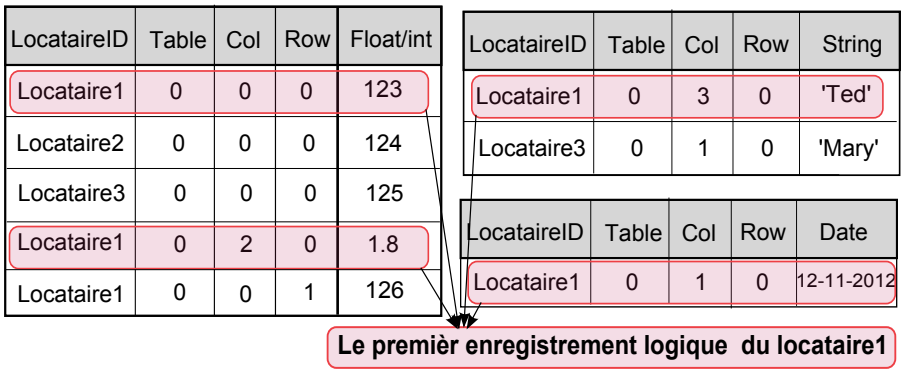


Figure 5.5 – Personnalisation de données à travers les tables croisées

De multiples tables croisées peuvent exister chacune réservée à un type de donnée différent. Pour soutenir efficacement l'indexation, cette approche peut être étendue en mettant en place deux tables croisées pour chaque type de données : l'une avec un index et l'autre sans index. Chaque donnée d'un locataire sera placée dans une de ces deux tables selon la manière dont ce locataire souhaite indexer ses propres données. Cette approche fournit un degré de flexibilité assez élevé vu qu'elle peut supporter n'importe quelle structure du schéma logique. Cependant, cette approche traite plus de méta-données que des données réelles. La reconstruction des schémas logiques exige un nombre important de jointures le long de la colonne `Row` à travers l'ensemble des tables physiques. Cela conduit à une surcharge d'exécution pour interpréter les méta-données qui est relativement plus considérable par rapport aux approches précédentes.

Personnalisation par XML : cette approche exploite les nouvelles fonctionnalités de certains SGBD qui supportent nativement des données structurées suivant le langage XML (Extensible Markup Language). Ce support a été principalement analysé dans [FK99] et mis en œuvre plus tard dans de nombreux SGBD commerciaux tels que *Microsoft® SQL Server 2005* [Rys05], *IBM DB2 PureXML* [SCA06] et *Oracle* [Zeh10]. Selon cette approche, les données spécifiques à chaque locataire sont sauvegardées sous forme d'un document XML associé avec chaque enregistrement ajouté. Ces documents sont stockés dans des attributs qui complètent les tables communes à travers une seule colonne supplémentaire ; *Ext-XML* (voir figure 5.6).

Étant donné que ces documents varient entre les locataires, ils sont faiblement typés. Cette approche présente comme avantage celui de conserver les personnalisation aussi légères que possible, ce qui est une considération importante pour la maintenance. Cependant, la récupération d'un document XML nécessite son extraction du SGBD en tant qu'un objet entier et ensuite de le traiter dans l'application sans aucune possibilité de rechercher uniquement les parties impor-

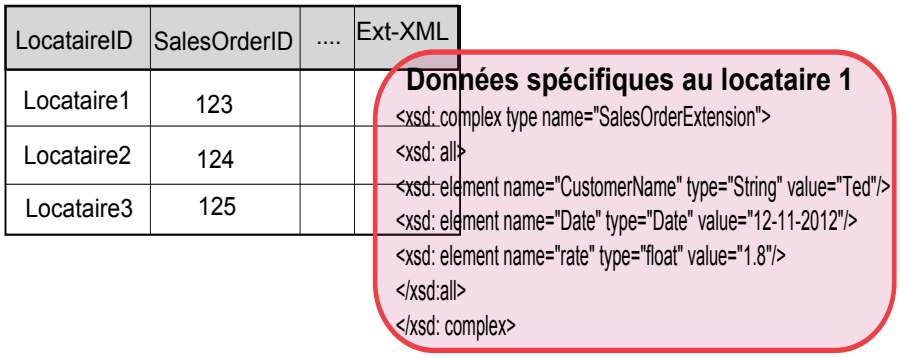


Figure 5.6 – Personnalisation de données à travers les documents XML

tantes du document pour le locataire qui est en cours d'utiliser l'application. Tandis que certains SGBD fournissent le langage *X-Query* [BCF+03] spécifique à la manipulation de ces documents, les études montrent que ce langage souffre d'une inefficacité et peut avoir un temps d'exécution exponentiel à la taille du document interrogé [Par09].

Pour résumer, nous avons présenté quatre méthodes de personnalisation du schéma d'une base de données complètement partagée qui permettent de personnaliser le modèle de données de l'application en fonction de chaque locataire. Cependant, chacune de ces méthodes présente des inconvénients ou des limites qui ne lui permettent pas d'être la plus efficace à utiliser dans certaines conditions. Par exemple, la méthode qui consiste à utiliser des *tables d'extensions* porte la surcharge des jointures supplémentaires pour récupérer les fragments des enregistrements des locataires dispersés à travers ces tables. Elles est donc limitée à une utilisation dans des applications supportant un nombre de locataires relativement petit. La méthode qui consiste à utiliser *Les champs de réservation* permet aux locataires d'étendre le modèle de données de différentes manières en fonction de leurs besoins. Toutefois, elle a l'inconvénient évident d'être dans l'obligation d'avoir des enregistrements très large et laissant au SGBD la gestion de nombreuses valeurs nulles. De plus, les indexations ne sont pas prises en charge dans cette méthode car les colonnes supplémentaires partagées entre les locataires peuvent avoir une structure différente et des types de données différents. Cette limitation conduit à la nécessité d'ajouter d'autres structures pour supporter les indexations de données lors de la personnalisation du modèle. La méthode qui consiste à utiliser des *tables croisées* peut éliminer les valeurs nulles et peut lire d'une manière sélective à partir d'un moindre nombre de colonnes. En revanche, cette méthode gère plus de colonnes de méta-données que des données réelles et la reconstruction d'une table logique de n colonnes exige un nombre très élevé de jointures, ce qui conduit à un temps d'exécution beaucoup plus élevé pour interpréter les méta-données. Pour cette raison, les auteurs dans [AGJ+08] proposent une nouvelle méthode de personnalisation appelée *« Chunk Folding »* (inspirée des tables croisées) qui consiste à verticalement diviser les tables logiques de locataires en plusieurs morceaux (*Chunks*). Ces *Chunks* seront ensuite repliées l'une sur l'autre en différentes tables physiques qui se joignent selon les besoins. Cependant, cette méthode reste toujours dans la phase de la recherche théorique, et les auteurs ne proposent pas un algorithme de reconstruction de requêtes efficace pour obtenir les résultats appropriés. La dernière méthode consiste à utiliser les documents XML pour rendre le modèle de données personnalisable arbitrairement sans introduire beaucoup de change-

ment au schéma physique (une seule colonne supplémentaire). Si les locataires n'exigent pas un degré trop élevé de personnalisation du schéma, *nous pensons que la personnalisation par XML est la meilleure méthode de personnalisation à adopter* et que ce format de données est déjà supporté par la majorité des SGBD commerciales. Cependant, si la personnalisation nécessaire pour répondre aux besoins spécifiques de locataires est assez considérable à un degré où le schéma logique d'un locataire peut être très différent d'autres locataires, l'utilisation massive de méta-données avec un moteur de génération des requêtes optimisées devient le seul choix possible (voir section 5.3.1).

5.2.3 La sécurisation d'accès aux données

Cette sous-section met l'accent sur la sécurisation d'accès aux données de locataires, celle également décrite comme « *la prévention d'un locataire d'obtenir les privilèges d'accès aux données appartenant à d'autres locataires* » [WGG+08]. Cela revient à protéger les données de chaque locataire à des niveaux de sécurité plus élevés que ceux qui existent dans les applications à locataire unique. Selon [WGG+08], il existe deux approches principales de sécurisation (voir figure 5.7) :

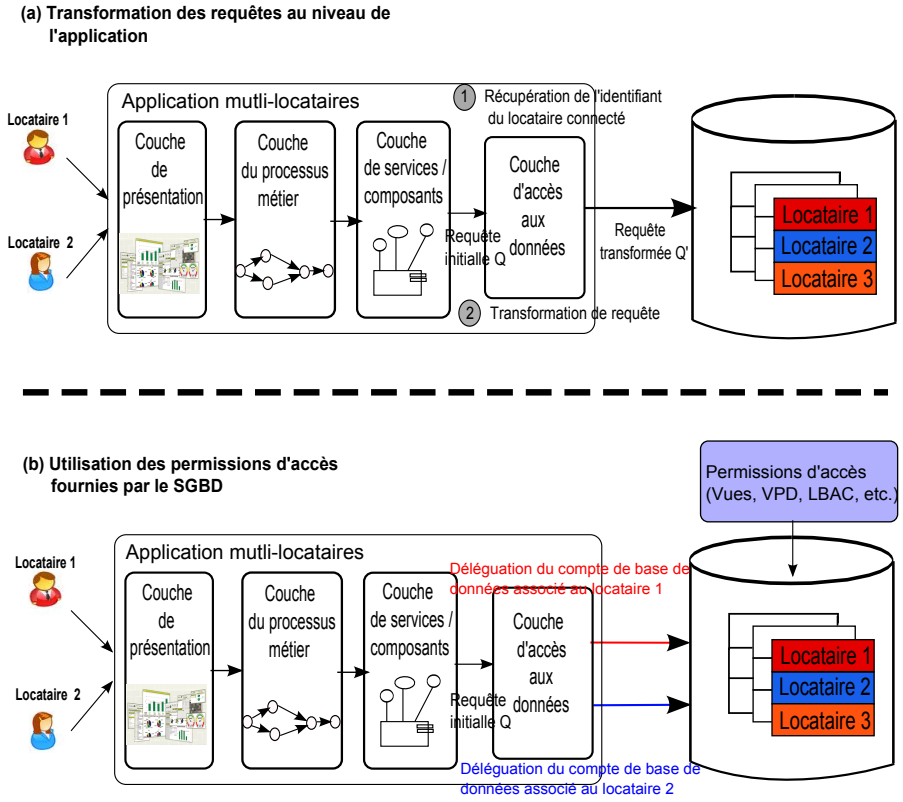


Figure 5.7 – Les différentes solutions de sécurisation d'accès aux données de locataires

Transformation de requêtes au niveau de l'application : cette approche consiste à transformer les requêtes générées par la couche d'accès aux données (voir figure 5.7 (a)), à travers l'injection dynamique de *filtres* qui modifient la structure de ces requêtes. Cette transformation est réalisée en

ajoutant un ou plusieurs critères supplémentaires à la clause `WHERE` de chaque requête pour considérer l'identité du locataire connecté. Les codes SQL 5.1 et 5.2 montrent l'exemple de transformation d'une requête initiale saisie par un développeur pour récupérer tous les bons de commande associés à un client du magasin (un magasin est un locataire de l'application).

Listing 5.1 – Requête initiale

```
1 SELECT o.*, c.CustomerID FROM Customer c INNER JOIN SalesOrder o
2 ON c.CustomerID = o.CustomerID
3 WHERE c.CustomerName='Ted'
```

Pour sécuriser les données et uniquement récupérer celles qui correspondent au locataire connecté (le locataire '123' dans ce cas), la transformation doit générer la requête suivante :

Listing 5.2 – Requête transformée

```
1 SELECT o.*, c.CustomerID FROM Customer c INNER JOIN SalesOrder o
2 ON c.CustomerID = o.CustomerID
3 WHERE c.CustomerName='Ted'
4 //Ajout du filtre
5 AND c.LocataireID=123 AND o.LocataireID=123
```

Cette approche nécessite l'implémentation d'un algorithme de transformation avec une complexité linéaire, qui parcours l'arbre syntaxique de chaque requête et injecte les filtres dans les endroits appropriés. Bien que cette approche soit fréquemment utilisée [BZP+10, BZ10b, GTA11], elle comporte des risques potentiels de sécurité. Vu que tous les locataires partagent un compte de base de données unique, un locataire malveillant peut accéder aux données d'autres locataires en exploitant certaines failles de sécurité qui lui permettent d'injecter du code SQL dans l'application et modifier par conséquent les résultats à obtenir [FDFI10]. Ainsi, des mesures de sécurité supplémentaires doivent être prises pour interdire ce genre d'attaques et rendre cette approche plus fiable.

Utilisation de permissions d'accès fournies par le SGBD : cette approche consiste à déléguer au SGBD la responsabilité de sécurisation de données. Ainsi, chaque locataire doit être différencié au niveau du SGBD par un compte utilisateur différent auquel des permissions d'accès seront attachées. L'exemple le plus connus et le plus simple pour réaliser cette approche est celui de l'utilisation des vues. Une vue de la base de données représente une table virtuelle qui permet d'accéder uniquement à un ensemble de données lié à un contexte d'exécution ou une fonctionnalité particulière. Pour sécuriser les données des locataires, le système doit créer une vue pour chaque table du schéma partagé. Ces vues ajoutent les mêmes critères de filtrage décrites dans l'approche précédente, sauf que la gestion se fait au niveau du SGBD et donc il empêche les risques potentiels d'attaques. Le code 5.3 montre la création d'une vue pour la table `SalesOrder` :

Listing 5.3 – Sécurisation d'accès aux données de locataires à travers l'utilisation des vues

```
1 CREATE VIEW VUE_LOCATAIRE_SalesOrder
2 AS SELECT *
3 FROM SalesOrder o
4 WHERE o.LocataireID = CURRENT_USER
```

Lorsque une requête de type `SELECT` est reçue, le SGBD évalue chaque enregistrement des tables concernées et détermine son appartenance aux résultats de cette requête selon les critères de filtrage définies dans la vue. La couche d'accès aux données doit maintenir une correspondance (via des méta-données) entre un locataire de l'application et un compte utilisateur du SGBD pour pouvoir

basculer dynamiquement entre les différents comptes lors de l'envoie des requêtes (voir figure 5.7 (b)). Cette approche présente une sécurisation plus élevée que celle de l'approche précédente car elle utilise des fonctions fournies et validées par le SGBD. Par ailleurs, des mécanismes de sécurité plus avancés qui fournissent un contrôle d'accès assez granulaire (niveau enregistrement) (ex : Federation de SQL Azure, Virtual Private Database (VPD) d'Oracle, Label-Based Access Control (LBAC) de IBM DB2, etc.), ont étés récemment exploités et ont montré leur capacité à supporter la mise en œuvre d'une base de données mutualisée et assurer la sécurisation requise [Sha11].

Cependant, d'après notre analyse de ces différents mécanismes nous avons constaté leur limitation face à la gestion de la collaboration entre les locataires. En effet, la collaboration est une valeur ajoutée du logiciel qui nécessite un niveau de partage de données métiers et une flexibilité de sécurisation qui oblige une redéfinition continuelle des permissions d'accès attribuées aux locataires. Celle-ci ne peut pas être automatisée à travers un SGBD [GAS+11]. Notre retour d'expérience montre que cette caractéristique de collaboration est plus simple à implémenter dans l'approche précédente sur une base d'extension de l'algorithme de transformation des requêtes.

5.3 Base de données mutualisée : approches d'isolation de données existantes

Contrairement aux travaux de recherches scientifiques gérant la variabilité dans les applications mutualisées, celles traitant le défi d'isolation de données de locataires n'ont toujours pas atteint le même niveau de maturité. La majorité analysent ce défi d'une manière générale (équivalente à l'étude effectuée dans la section précédente) sans proposer des solutions d'isolation concrètes. Nous allons brièvement citer quelques travaux :

Les auteurs dans [JA+07] ont étudié les bases de données relationnelles et ont conclu qu'il n'existe pas un support intrinsèque disponibles pour la mutualisation dans les SGBD actuellement disponibles. Ils ont exploité la possibilité d'une mise en correspondance du contexte de locataire (ou identifiant de locataire) avec les modèles existants de la sémantique de base de données pour isoler les données de locataires. Cela nécessite essentiellement l'introduction d'un cadre administratif dans le schéma de base de données qui devrait maintenir des méta-données liées à chaque locataire. De plus, ce cadre doit permettre la gestion des utilisateurs, le contrôle d'accès, la gestion des quotas de disque qui prend en charge dûment l'exécution des opérations administratives en vrac.

Les auteurs dans [CC06, CCW06] ont exposé de divers modèles qui peuvent être utilisés au niveau de la base de données pour mettre en œuvre la mutualisation, tandis que les auteurs dans [GSK+08] suggèrent que les SGBD actuels et leur mise en œuvre ne fournissent pas la sémantique nécessaire pour soutenir nativement la mutualisation. La solution proposée est d'introduire le concept de locataire au niveau de l'application et d'employer des stratégies de partitionnement de données basées en fonction de ce concept.

Les auteurs dans [BZ10b, BZ10a] ont abordés divers défis de la mutualisation et le rôle dont ce principe peut jouer dans la fourniture de solutions abordables pour les petites et moyennes entreprise. Les défis de mutualisation identifiés par ces auteurs comprennent : la performance, le passage à l'échelle, la maintenance. L'isolation de données est également identifiée comme une exigence de base pour la mutualisation et les auteurs ont proposé d'utiliser une couche d'accès aux données entre la logique métier de l'application et la base de données qui devrait être responsable de la création de nouveaux locataires et la transformation de requêtes. Cependant, leur architecture porte un inconvénient car elle nécessite la

création manuelle des fonctionnalités qui permettent à une application à locataire unique d'être convertie en une application mutualisée.

En abordant le sujet de conversion ou de *réingenierie* d'une application vers la mutualisation, les auteurs dans [BZP+10] identifient la base de données comme l'une des les trois zones qui doivent être restructurées afin de permettre la mutualisation d'une application SaaS. Conformément à ces auteurs, les SGBD commerciaux (« *off-the-shelf* ») ne sont pas intrinsèquement équipés pour gérer le principe de mutualisation. Par conséquent, ce principe doit être introduit dans l'application à travers une couche supplémentaire. La tâche principale de cette couche, en termes d'isolation des données, devrait être la transformation de requêtes. Cette couche doit s'assurer que toutes les requêtes sont articulées d'une manière selon laquelle chaque locataire est en mesure d'accéder uniquement à ses propres données.

Les auteurs dans [DWY10] ont discuté et exploité le support XML natif dans les SGBD actuels pour personnaliser le schéma partagé de la base de données en fonction de chaque locataire, ainsi que l'implémentation de requêtes de lecture et d'écriture. Les auteurs conclu que la méthode de personnalisation par XML est la plus efficace que les autres méthodes de personnalisation existantes (voir section 5.2.2).

En raison de la complexité supplémentaire que ces approches imposent pour implémenter et gérer un système d'isolation de données, nous allons expliquer par la suite deux approches industrielles actuellement présentes sur le marché de services du cloud : la *plateforme force.com* et le *service SQL Azure* qui proposent des services de gestion de base de données avec des capacités d'isolation de données intégrées. Nous précisons que ces deux approches supposent l'utilisation d'un schéma de données partagé entre les locataires, et gèrent plus particulièrement les exigences qui en découlent telles que la personnalisation de ce schéma et la sécurisation d'accès aux données.

5.3.1 La plateforme force.com de l'entreprise Salesforce

L'architecture de données mise en place par l'entreprise Salesforce est principalement appliquée à un système CRM (Client Relation Management) à la demande qui a récemment annoncé l'hébergement de plus de 70.000 locataires payants tous servis à partir de la même instance d'application et de base de données. Saleforce se base dans cette démarche sur sa plateforme de développement prioritaire Force.com et une infrastructure composée de milliers de serveurs.

Les principaux éléments de leur plateforme sont un schéma de base de données dirigé par les méta-données, une méthode de partitionnement de cette base et un cadre de développement d'application intégré. La figure 5.8 montre les couches de l'architecture de Salesforce.

Dans cette sous-section, nous allons uniquement nous concentrer sur la couche de la plateforme de développement et plus particulièrement sur le modèle de données mis en place par cette plateforme pour prendre en charge plusieurs locataires. Ce modèle est basé sur l'utilisation massive de méta-données. En effet, tous les artéfacts logiciels de chaque application développée sur cette plateforme tels que les workflows, les privilèges, les configurations, les règles métiers, les tables de données, etc. sont sauvegardés sous forme de méta-données. La plateforme sépare ces méta-données décrivant les applications de l'environnement d'exécution et de données réelles. Cette séparation permet la mise à jour du noyau de la plateforme et des applications de façon indépendante. Tous les objets qui sont connus par la plateforme Force.com seulement existent virtuellement lors de l'exécution. Les schémas logiques de locataires sont liés avec un seul schéma physique du SGBD par une correspondance du schéma sophistiqué qui combine de multiple techniques de représentation de données discutées en section 5.2.2. Force.com se base sur le système SGBD d'Oracle et utilise sa mémoire rapide pour réduire le nombre d'entré/sortie aux méta-données. Ceci dit, celles-ci ne sont pas disponibles pour le SGBD, et ainsi, plutôt que de les gérer, il les stocke simplement. Pour chaque accès aux données, l'environnement d'exécution récupère les

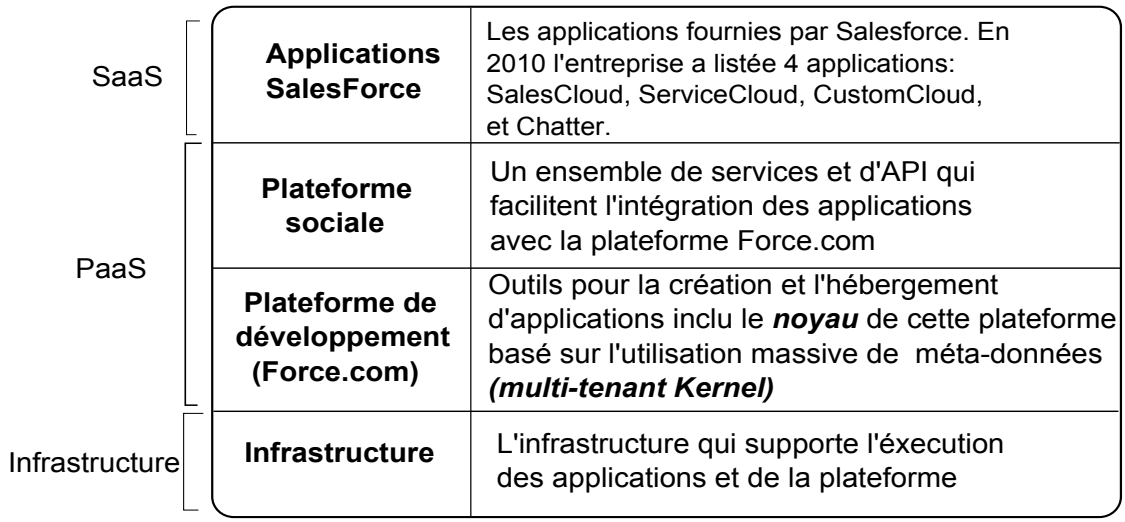


Figure 5.8 – Les couches de l'architecture de Salesforce

méta-données qui représentent physiquement une application et effectue une transformation vers sa représentation logique. L'environnement d'exécution a un optimiseur de requêtes interne qui s'appuie sur la connaissance des méta-données afin de générer la requête SQL optimale.

La figure 5.9 compare un schéma de données traditionnel avec celle de Force.com. Dans le schéma de données traditionnel (voir figure 5.9(a)), les objets et les champs définis représentent une abstraction du monde réel. Ce schéma contient une table de données distincte pour chaque type d'objet représenté. Les attributs spécifiques à chaque objet sont représentés par des champs dans les tables. Les instances des objets sont représentées par des enregistrements insérés comme des données réelles dans les tables de la base de données. Les relations sont représentées par des champs d'une table faisant référence à un champ clé dans une autre table.

Le modèle de données de Force.com est dirigé par les méta-données (voir figure 5.9(b)). Les objets et les champs de données qui définissent le modèle de données de l'application sont mis en correspondance avec les tables de méta-données. Par exemple, la table spécifique à la gestion de méta-données sur les objets réels (*Object Meta-Data*) stocke les définitions de ces objets. Cela est réalisé à travers un champ qui stocke l'unique ID d'un objet, le nom de cet objet et l'ID de l'organisation du locataire qui a crée l'objet. De même, Force.com propose une table pour gérer les méta-données sur les champs (*Feild Meta-Data*). Celle-ci stocke les définitions des champs de chaque objet. Elle contient l'identifiant unique du champ, le nom du champ, le type de données à stocker dans ce champ, l'ID de l'objet concerné, le numéro de séquence de ce champ dans l'objet, l'ID de l'organisation du locataire qui a créé le champ et finalement une information indiquant la nécessité d'indexer les données sur ce champ.

D'ailleur, la plateforme Force.com utilise deux autres tables pour indexer les données et structurer les objets : la table *Indexes*, qui a pour rôle de stocker les définitions des index, et la table *Relationships*, qui stocke les informations sur les relations entre les objets.

Les index dans Force.com ont pour rôle d'optimiser l'accès aux données. La plateforme gère ces index en copiant d'une manière synchronisée les données à indexer dans la table *Indexes*. Celle-ci contient des colonnes fortement typées et nativement indexées (*StringValue*, *NumericValue* et *Date*). Quand les données d'un locataire sont accédées, le générateur interne de requêtes interroge en premier cette table,

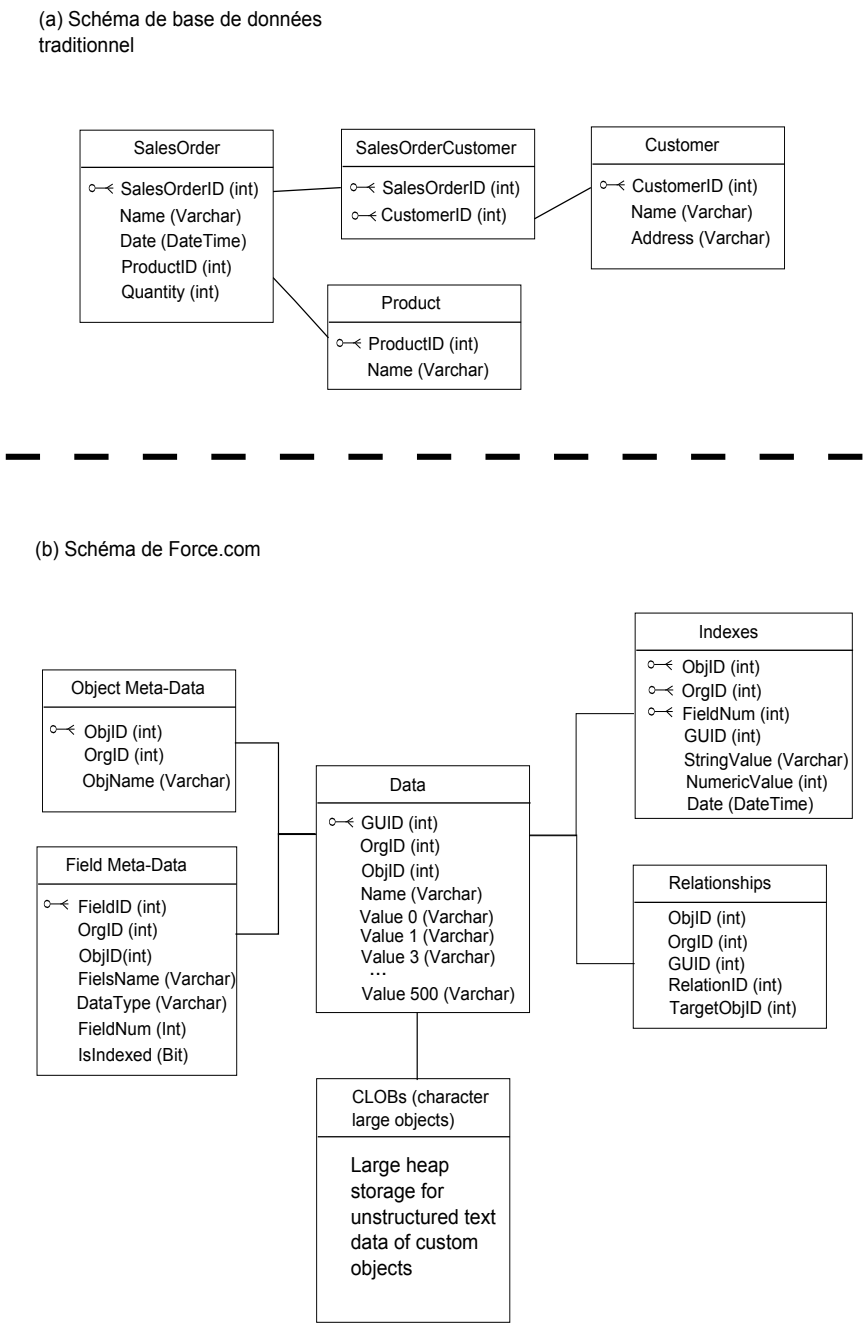


Figure 5.9 – Comparaison entre un schéma de données traditionnel et celui de Force.com dirigé par le méta-données [WB09]

et aligne le résultat avec les enregistrement trouvés. Par souci d'optimisation, la plateforme Force.com emploie des mécanismes de partitionnement fournis par le SGBD qui supporte cette architecture de données.

La table *Relationships* stocke les informations sur les relations et identifie les objets et leurs relations. Celle-ci est généralement utilisée pour mettre en œuvre des opérations de jointure.

La table de données (*Data*) est une table qui stocke les données réelles pour tous les locataires. Chaque enregistrement dans cette table est identifié par un identificateur unique global (GUID) qui sert de clé primaire. Elle contient aussi l'ID de locataire, l'ID de l'objet, le nom naturel de l'objet et les données réelles. Celles-ci sont stockées dans des colonnes de longueur variable avec des noms génériques *Val0, Val1,..., Val500*. Chaque valeur parmi les 500 est mise en correspondance avec un champ de l'objet de données concerné à travers le colonne *FieldNum*. Ces colonnes sont de type `VARCHAR` pour supporter des types arbitraires de données. Outre la table de données, cette plateforme prévoit une table spécifique pour stocker des données non-structurées sous forme des *CLOBS* (character large object storage).

Un inconvénient potentiel de cette architecture est que tous les locataires partagent les tables de méta-données et de données, de sorte que la charge sur la base de données peut devenir trop lourde. Force.com résout ce problème en utilisant une technique de partitionnement. Dans cette technique, tous les serveurs de l'ensemble de l'infrastructure de Force.com réservée aux données partagent le déploiement de la seule base de données mutualisée. Donc, la même base de données est partitionnée sur plusieurs serveurs, et chaque partition possède ses propres mémoire, CPU et dispositifs de stockage. Les demandes des locataires sont automatiquement acheminées d'une manière pré-configurée vers la partition qui stocke leurs données, celui qui accélérer l'exécution des requêtes et passer ainsi à une grande échelle.

Force.com permet aussi aux locataires de personnaliser et de concevoir leurs propres applications. Un outil qu'elle fournit est une application Web qui permet aux locataires de définir de nouveaux objets d'extension. Le schéma de base de données dirigée par les méta-données est un facteur clé de cette plateforme de développement. Celle-ci a également fournit d'autres moyens pour créer des extensions personnalisées. Parmi ces moyens, nous pouvons cité *Apex*, son propre langage de développement, un *plug-in Eclipse*, et un ensemble de services Web que les locataires peuvent utiliser pour créer des extensions spécifiques. Cette architecture fournit un excellent regard sur les questions qui se posent par les fournisseurs du cloud. La base de données dirigée par les méta-données et la méthode de partitionnement se combinent pour offrir une plateforme mutualisée et évolutive. Les méta-données permettent également un développement basé entièrement sur le Web (à travers un simple navigateur), que tous les fournisseurs de SaaS peuvent utiliser pour créer leurs propres applications à la demande.

5.3.2 Le service SQL Azure de la plateforme Windows Azure

SQL Azure est un service de gestion de base de données fourni par la plateforme de développement Windows Azure comme étant l'un de ses composants principaux hébergés dans les centres de données de Microsoft. Il permet aux clients l'accès à un SGBD à la demande pour créer et gérer des bases de données relationnelles avec des capacités de synchronisation, de reporting et d'analyse. Ce service respecte les caractéristiques principales du Cloud Computing et propose aux clients de payer uniquement pour ce qu'ils utilisent en termes de volume de données stockées et de bande passante du réseau. SQL Azure est basé sur la technologie de Microsoft® SQL Server et offre un environnement de gestion de données classique avec la possibilité de créer des index, des vues, des procédures stockées, des triggers, etc. Toutes les données stockées sur ce service sont périodiquement répliquées pour la tolérance aux pannes.

Une application utilisant SQL Azure peut être exécutée sur des infrastructures différentes, On-Permise ou sur le cloud. Indépendamment de l'infrastructure d'exécution, l'application accède à ses données hébergées sur ce service via le protocole *« Tabular Data Stream » (TDS)*, qui est le même protocole utilisé pour accéder à la version locale de Microsoft® SQL Server, et ainsi, l'application peut réutiliser les bibliothèque et les interface d'accès supportées par la version locale. L'administration de

ce service est gérée par Microsoft, et donc n'est pas exposé physiquement pour effectuer des fonctions administratives. Un client ne peut pas l'arrêter ou interagir directement avec le matériel sur lequel il est exécuté.

La taille maximale d'une base de données dans SQL Azure est de 10 gigaoctets. Une application dont les données sont à l'intérieur de cette limite peut utiliser une seule base de données, tandis qu'une application qui utilise un plus grand volume de données doit créer des bases de données multiples. Avec une base de données unique, l'application ne peut accéder qu'à ses propres données et les requêtes SQL peuvent être utilisées d'une manière ordinaire. Avec de multiples bases, l'application doit répartir ses données entre elles et ne peut plus émettre une seule requête SQL qui accède à toutes les données de toutes les bases. Ainsi, l'application doit être consciente de la façon dont les données sont divisées. Les applications ayant un petit volume de données peuvent également choisir d'utiliser plusieurs bases de données. La mutualisation de données sous SQL Azure peut être ainsi supportée suivant cette approche en adoptant le modèle de bases de données séparées, et acheminer les requêtes SQL vers la bonne base (en fonction du locataire connecté) à travers l'utilisation d'un mécanisme spécifique fourni par ce service appelé *Fédération*.

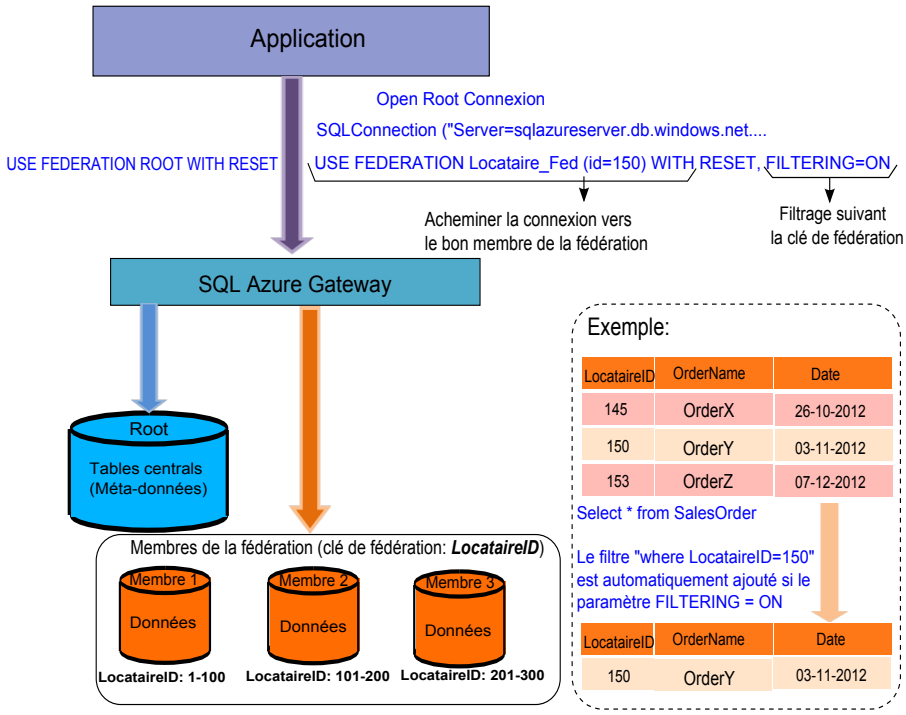


Figure 5.10 – Sécurisation d'accès aux données de locataires à travers le mécanisme de fédération [DBET11]

Une fédération sous SQL Azure est une collection de partitions de base de données connue sous le nom de schéma de fédération. Ce dernier supporte également la mutualisation de données dans une instance de base de données unique (modèle du schéma partagé) et propose une sorte de combinaison entre les deux modèles de séparation (modèle 1 et 3) pour un meilleur passage à l'échelle [Biy11].

La figure 5.10 montre l'architecture de ce mécanisme. Celui-ci sécurise l'accès aux données de locataires d'une façon indirecte à travers l'utilisation des *clés de fédération*. Celles-ci (qui correspondent indirectement aux identifiants de locataires) déterminent la façon dont les données sont distribuées dans les partitions au sein de la fédération. Chaque enregistrement ajouté dans une table fédérée sera associé avec une clé de fédération précise, afin de pouvoir différencier les données de locataire. Les partitions de base de données au sein d'une fédération sont appelées *membres de fédération*, et sont physiquement séparées dans des base de données distinctes. Les tables fédérées sont ainsi les tables réparties entre les membres d'une fédération. Chaque membre de la fédération possède son propre schéma et contient les enregistrements des tables fédérées qui correspondent aux locataires dans les valeurs de leurs clés de fédération sont associées à ce membre. La récupération de toutes les enregistrements contenus dans un membre de fédération correspondant à une valeur de clé de fédération spécifique s'appelle une unité atomique de fédération (les données d'un locataire dans ce cas).

Un membre de fédération établit une séparation physique entre les données qu'il contient et celles qui sont stockées dans les autres membres. Chaque membre de fédération possède son propre schéma, qui peut provisoirement diverger du schéma des autres membres en raison d'un traitement spécifique au membre. Les locataires qui exigent les mêmes modifications du schèma sont mieux placés dans le même membre de fédération. Bien que les membres de fédération soient physiquement implémentés en tant que bases de données, ils sont logiquement référencés au niveau de l'application en tant que plage de valeurs de clé de fédération. Par exemple, pour accéder logiquement à une base de données membre de fédération contenant des enregistrements associées à la plage de valeurs de clé de fédération 50-100, il faudrait spécifier une valeur clé comprise dans cette plage plutôt que le nom de la base de données. L'accès aux fédérations se fait par le biais d'une base de données racine de fédération (Root), qui représente la limite de la fédération pour une application. Celle-ci joue le rôle de point de terminaison logique de sorte que les applications se connectent à une fédération en routant les connexions vers le membre de fédération approprié (celui qui dépend de la valeur de clé de fédération spécifiée). Chaque base de données Root peut contenir plusieurs fédérations, chacune avec son propre schéma de fédération.

Pour utiliser ce mécanisme, chaque application cliente doit tout d'abord créer une fédération et spécifier les tables qu'elle souhaite fédérer, tout en indiquant les clés de fédération appropriées.

Listing 5.4 – Création d'une fédération sous SQL Azure

```
1  //Définition de la fédération et de type de données des clés de fédération (INT)
2  CREATE FEDERATION Locataire_Fed (id INT RANGE)
3
4  //Connexion au premier (et unique pour l'instant) membre de la fédération
5  USE FEDERATION Locataire_Fed (id = 0) WITH FILTERING = OFF, RESET
6  GO
7
8  //Creation d'une table de données dans ce membre de fédération
9  CREATE TABLE SalesOrder(
10 OrderID Int not null
11 OrderName Varchar(500) not null
12 OrderDate DateTime not null
13 LocataireID Int not null
14 primary key (OrderID)
15 )
16
17 //Définir la colonne LocataireID comme la clé de fédération
18 FEDERATED ON (id=LocataireID)
19 GO
```

Après avoir définir la fédération et les tables fédérées, l'application doit indiquer la valeur de la clé de fédération pour chaque requête émise. Cette valeur sera utilisée par le service SQL Azure pour filtrer les données.

Listing 5.5 – Envoie d'une requête avec sécurisation d'accès sous SQL Azure

```
1 USE FEDERATION Locataire_Fed (id = 150) WITH FILTERING = ON, RESET
2 GO
3 SELECT * FROM SalesOrder
4 GO
```

Pour l'instant, la fédération créée ne contient qu'un seul membre (base de données) dans lequel les données de tous les locataires seront stockées. SQL Azure filtre les données par rapport à la valeur de la clé de fédération envoyée avec chaque requête. Pour créer un nouveau membre de fédération, SQL Azure propose la commande `SPLIT`.

Listing 5.6 – Création d'un nouveau membre de fédération sous SQL Azure avec la commande SPLIT

```
1 USE FEDERATION ROOT WITH RESET
2 GO
3 ALTER FEDERATION Locataire_Fed SPLIT AT (id = 101)
4 GO
```

Cette commande consiste à fractionner un membre de fédération existant en deux sur la base d'une valeur de clé de fédération à spécifier, et à partir de laquelle le fractionnement commence. Par exemple, depuis que la clé de fédération est le LocataireID, toutes les tables fédérées seront fractionnées en fonction de la valeur de cette clé indiquée dans la commande (101 dans ce cas). Cela signifie que tous les enregistrements dont l'ID de locataire est inférieur à 101 resteront dans le premier membre de fédération, tandis que les autres seront placés dans le deuxième membre. Afin d'effectuer ce fractionnement, il faut tout d'abord se connecter à la racine de fédération (Root), pour qu'elle modifie les méta-données qui décrivent l'organisation actuelle de la fédération. Pour chaque requête, SQL Azure évalue la valeur de la clé de fédération pour indiquer le membre dans lequel les enregistrements ayant cette valeur se trouvent, et ensuite de transformer la requête en injectant les filtres appropriés (si le paramètre FILTERING est mis à ON). Cette transformation se fait au niveau de l'*algebrizer* [Fri09] de Microsoft SQL Server, celui responsable d'optimiser les requêtes avant leur exécution.

5.4 Problématique dégagée et approche suivie

L'isolation de données de locataires est un défi principal à relever dans toute approche de mutualisation au niveau applicatif du cloud. Malgré les limitations des recherches actuelles, leur direction commencent à être de plus en plus évidente et tendent vers un support natif de ce principe au niveau des SGBD. Entre-temps, nous avons remarqué l'apparition des services qui couvrent certaines exigences de ce défi, et qui gagnent actuellement du terrain et qui s'améliorent pour faciliter aux applications la gestion de la mutualisation sur la couche de données.

Pour profiter des services existants et leurs solutions d'isolation, les fournisseurs de SaaS doivent tout d'abord accepter d'héberger les données de leurs applications sur l'infrastructure de ces services. Ils doivent aussi concevoir l'accès aux données d'une manière conforme à leurs techniques proposées et considérer leurs notions uniques (ex : la fédération de SQL Azure, les méta-données de Force.com). Par conséquent, de nouveaux concepts seront introduits pour les applications qui sont propres aux services

utilisés, et il sera ainsi compliqué de migrer la base de données d'un fournisseur de service à un autre. Les données pourraient elles-mêmes être enfermées dans un format propriétaire à chaque service. Ainsi, les applications deviendraient fortement dépendantes d'un service précis et seraient ainsi obligées de suivre son évolution. Parmi les nombreuses problématiques du cloud, celle-ci rejoignent le problème de portabilité et d'interopérabilité (voir section 2.2.5), spécialement connue sous le nom de « *verrouillage* » d'application avec un fournisseur spécifique (*vendor lock-in*) [ALPW10].

Problème 1 : bien que le verrouillage préoccupe les clients de tous les services du cloud, il se manifeste différemment au niveau des données car elles représentent la partie la plus sensible d'une application. Cette problématique est devenue particulièrement préoccupante avec l'arrêt fréquent et parfois brusque dont subissent les services du cloud, à cause de la multiplication des sources d'échec que ce soit pour des raisons fonctionnelles, économiques ou financières. [Mar09] est un exemple d'un fournisseur de service de données dont l'arrêt a obligé les clients à développer à nouveau et dans l'urgence, leurs couches de données sur la base de nouveaux concepts proposés par d'autres fournisseurs de services.

Pour éviter ce verrouillage, les auteurs dans [CGJ+09] argumentent qu'une approche possible sera la standardisation des services proposant la gestion de données à la demande, afin de faciliter leur remplacement en cas d'arrêt. Dans cette direction, certains tentatives de standardisation ont vu le jour (par exemple l'Open Cloud Manifesto [Man09]) ainsi que de nombreux développements de plateformes « *open-source* », garantissant une transparence totale sur leur fonctionnement interne et leur façon de stocker les données. Nous pouvons citer le projet Reservoir [WEBYE09], à l'initiative d'IBM et de l'Union Européenne, qui consiste à développer un standard en matière de plateformes et des applications hébergées dans le cloud afin de permettre l'interopérabilité entre différents fournisseurs de services. Toutefois, nous croyons que cette approche est loin d'être faisable car la standardisation à ce niveau nécessite la modification de l'architecture interne de chaque service (accumulant des années de recherche et de développement) pour se conformer finalement à des normes génériques qui sont à la base très difficiles à définir et modéliser.

Une autre approche peut être adoptée pour éviter un tel verrouillage et qui consiste à fournir le système d'isolation de données à travers un ensemble de composants logiciels à installer dans l'environnement de développement propre à chaque fournisseur de SaaS. Les auteurs dans [GAS+11] proposent un cadre spécifique à la gestion des bases de données mutualisées à travers la fourniture des solutions pré-packagées de personnalisation du schéma partagé et de sécurisation d'accès aux données par les locataires. Cela éviterait la dépendance d'un service précis et donnent aux fournisseurs la liberté de choisir l'infrastructure qu'il souhaitent pour héberger leurs applications et leurs données. Toutefois, cette approche est complexe à réaliser vu la diversité des environnements utilisés pour développer les couches de données. Cela implique soit le développement de plusieurs versions du cadre proposé, ou de se limiter à des environnements de développement précis. Dans le deux cas, le passage à une grande échelle de clients est extrêmement compliqué.

Problème 2 : un autre besoin généralement ignoré par les services proposés et peu abordé par les recherches scientifiques sur la mutualisation de SaaS, concerne la *réingénierie* de la couche de données d'une application existante pour avoir la capacité de supporter plusieurs locataires [BZ10a, ZSTC10]. Ce besoin est devenu récemment très important dans le contexte de transition vers le cloud, où les entreprises cherchent des solutions simples et efficaces pour transformer leurs couches de données, sans qu'elles soient obligées de faire beaucoup de modifications ou bien de passer à un modèle de développement différent.

Approche suivie : la contribution de cette thèse en vue du traitement des problèmes identifiés consiste en la conception d'un système d'isolation de données de locataires sous forme d'un service.

Ce système présente comme objectif principal d'être à la fois indépendant de la couche de données de l'application à mutualiser et n'exige aucune introduction de nouveaux concepts ou investissement dans de nouveaux développements pour les fournisseurs de SaaS. Le système proposé sera entièrement automatisé et externalisé, il garde à l'application ses données et sera utilisable à la fois dans le cadre d'un nouveau développement, ou d'une réingénierie d'application existante (à locataire unique) vers la mutualisation.

5.5 Conclusion

Dans ce chapitre, nous avons présenté le défi d'isolation de données de locataires et exploré différentes façons de mises en œuvre typiques d'une base de donnée mutualisée. Plus précisément, nous nous sommes concentrés sur les aspects de séparation de données de locataires, de sécurisation d'accès à leurs donnés et de personnalisation du schéma de la base de données complètement partagée. Ce travail effectué peut aider le fournisseur de SaaS à avoir une idée plus claire sur les différents compromis. Nous avons déjà appliqué une partie des résultats de l'étude dans la conception et la mise en œuvre d'une véritable application mutualisée que nous allons détailler dans le chapitre suivant. Les expériences pratiques nous ont aidées à bien comprendre ce défi, et ensuite de toucher de plus près des sujets de recherche concernant la performance, l'optimisation et le passage à l'échelle.

Étant donné que le modèle de séparation de données basé sur un schéma complètement partagé est celui le plus efficace à maintenir, il constitue la première priorité pour une livraison économique d'une application SaaS. Toutefois, ce modèle impose l'implémentation d'un mécanisme de sécurisation d'accès aux données et une méthode de personnalisation du schéma de la base de données pour répondre aux besoins spécifiques de chaque locataire. De nombreux fournisseurs de SaaS font ainsi face à la gestion de l'isolation des données de locataires lors de l'adoption de ce modèle de séparation et engagent beaucoup d'efforts pour la manipulation de ses exigences lors du développement des applications ainsi que lors de la migration des applications existantes vers la mutualisation. Dans ce cadre, nous avons étudié un ensemble de solutions académiques et surtout industrielles fournissant des systèmes d'isolation de données à la demande couvrant une ou plusieurs exigences de ce défi et nous avons déduit un ensemble de conclusions et de meilleures pratiques sur la façon de concevoir la mutualisation sur la couche de données. Les limites identifiées dans les approches existantes ont fait la base de notre nouvelle approche de gestion pour relever ce défi.

PARTIE III

Contribution et validation

Externalisation de la Gestion de la Variabilité des besoins de locataires Dans les Applications SaaS Mutualisées

6.1 Introduction

D'une manière semblable aux applications SaaS à locataire unique où celles traditionnelles vendues à des clients et installées sur leurs propres infrastructures, les applications SaaS mutualisées suivent un processus de développement qui inclut l'identification des exigences, la spécification, la conception, la mise en œuvre et le déploiement. Étant donnée la structure organisationnelle de ce type d'applications, son processus de développement doit porter la surcharge de la gestion de la variabilité dans la majorité de ses étapes. Dans le chapitre 4, nous avons présenté une approche de gestion de cette variabilité qui exploite des concepts bien établies dans le domaine de LDP (Lignes de produits), et nous avons analysé ensuite les approches qui proposent une adaptation dynamique de l'architecture en manipulant ces concepts. Dans cette continuité, nous avons identifié certaines limites dans les approches proposées, notamment sur les axes de modélisation et de résolution de la variabilité, qui découlent de la spécificité des applications SaaS mutualisé et de leurs caractéristiques émergentes (voir section 4.4).

Hormis les limites identifiés, l'utilisation de ces approches pour traiter la variabilité (voir section 4.3) pourra toujours aboutir à une solution de gestion de la variabilité qui reste valable tant que l'approche est techniquement faisable indépendamment de sa conception ou de sa manière de fonctionner. Cependant, le véritable problème que nous avons dégagé n'est pas uniquement lié à la conception ni à l'efficacité des approches (qui est un sujet relatif et à débats), mais aussi à leurs coûts additionnels en terme de développement concret qui risque de retarder considérablement la mise de l'application sur le marché. Pour de nombreux fournisseurs de SaaS qui sont financièrement et temporellement contraints, un tel retard ne peut pas être acceptable puisqu'il provoquera un déséquilibre financier auquel ils ne sont souvent pas en mesure de faire face.

Dans le même temps, et pour pouvoir profiter des avantages de la mutualisation, la gestion de la variabilité est nécessaire pour toucher plus de locataires et augmenter la marge de bénéfices. Pour sortir de cette impasse, l'externalisation de cette gestion à un fournisseur plus expérimenté nous semble être un choix idéal. Cela permettrait aux fournisseurs de SaaS de se focaliser uniquement sur le cœur de leurs compétences et leur savoir faire dont la gestion de la variabilité ne fait pas forcément partie. De ce fait, nous avons orienté notre recherche vers une approche de gestion de la variabilité par externalisation et nous avons concrétisé nos travaux à travers la conception et la mise en œuvre d'un système fournissant la gestion de la variabilité sous forme d'un service.

Ainsi, ce chapitre est destiné à expliquer l'évolution qu'a connu notre travail de recherche ainsi que nos contributions concernant la gestion de la variabilité dans les applications SaaS mutualisées. La section 6.2 présente un exemple d'application pour l'industrie alimentaire que nous avons initialement développé en gérant sa variabilité au travers la modélisation avec les concepts d'OVM, et la résolution par l'exploitation des capacités des technologies de développement utilisées. Ensuite, la section 6.3 se focalise sur notre nouvelle approche de méta-modélisation de la variabilité déduite des limites identifiés dans la pratique, tout en expliquant les nouveaux concepts de modélisation que nous proposons et leur formalisation pour fournir une meilleur compréhension du méta-modèle de variabilité. La stabilisation de ces nouveaux concepts et leur vérification dans plusieurs cas réels a été la première étape mais également la plus importante pour aboutir à notre objectif de gestion de la variabilité par externalisation. Le chapitre 6.4 présente ainsi l'architecture, le processus d'externalisation et les composants clés de notre service de variabilité tout en clarifiant le rôle du méta-modèle dans cette architecture. Finalement, la section 6.5 conclut le chapitre.

6.2 Étude de cas : une application pour l'industrie alimentaire

Dans cette section, nous allons détailler notre méthodologie de gestion de la variabilité initialement adoptée pour une simple application de l'industrie alimentaire (Food Industry Application, FIA pour faire court). Celle-ci est une application mutualisée et développée à base de services dont le but principal est de permettre la simulation d'une recette par les utilisateurs avant de la fabriquer. Cette simulation passe à travers l'utilisation d'un service de simulation développé en interne qui permet à l'application d'évaluer le temps et le coût nécessaires à la fabrication des recettes, ainsi que leurs caractéristiques potentielles de qualité (i.e. la valeur nutritionnelle, le goût, l'odeur etc.). L'application utilise aussi deux services externes (services non-contrôlés par nous) pour gérer une chaîne d'approvisionnement alimentaire en se basant sur les résultats des simulations effectuées. Le premier service est celui d'un fournisseur d'ingrédients qui donne l'accès à une base de données contenant une description des ingrédients réels qu'ils fournissent, alors que le deuxième est un service de transport qui propose une livraison des ingrédients entre deux adresses à indiquer. Le rôle de ce dernier dans le processus est de livrer les ingrédients réels utilisés et validés dans une simulation, depuis l'entrepôt du fournisseur d'ingrédients vers les usines des locataires. La figure 6.1 montre le processus métier de l'application. Ce processus a été implémenté à l'aide de WCF (Windows Communication Foundation) [**?**] qui est un outil intégré dans l'environnement de développement .NET [**?**] pour modéliser et exécuter des applications à base de services Web.

L'utilisateur de l'application (appartenant à un locataire) accède aux fonctionnalités de notre application à travers une page Web dédiée qui lui permet de gérer ses propres recettes ou d'en créer de nouvelles. Dans ce dernier cas, l'utilisateur doit specifier les ingrédients qui composent cette nouvelle recette et le pourcentage respectif de chaque ingrédient de la composition. Il doit également décrire sa procédure de fabrication par l'envoi d'une structure de document XML pré-négociée décrivant les différentes étapes de la procédure. Lorsque l'application reçoit la description de la recette, elle analyse ses détails (a) pour extraire la liste des ingrédients qui la composent. L'application invoque ensuite (b) le service du fournisseur d'ingrédients pour récupérer les informations actuelles concernant le prix et la qualité des ingrédients dans la liste. Ces informations sont ensuite remises à disposition du service de simulation (c) pour évaluer les caractéristiques finales de la recette. Une fois la simulation est terminée, l'application renvoie à l'utilisateur un rapport contenant les résultats de la simulation avec une demande de validation des résultats obtenus (d). Dans le cas où l'utilisateur valide, la recette est enregistrée dans la base de données (e) et un ordre de transport des ingrédients est envoyé (f) au service de transport intégré.

Lors de la réception des ingrédients réels, l'usine peut commencer l'exécution d'un véritable processus de fabrication tout en étant sûr que la recette aura les caractéristiques prédites par l'application.

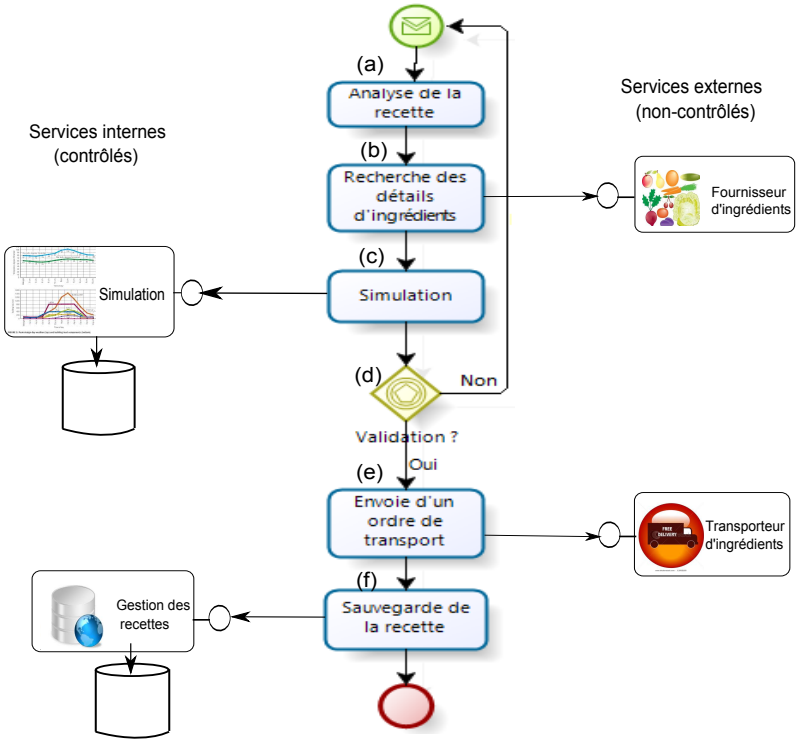


Figure 6.1 – Le processus métier de l'application FIA

6.2.1 Identification de la variabilité dans l'application FIA

Afin de détecter les variations possibles pouvant intéresser les locataires, nous avons présenté les processus métier de l'application à un ensemble de locataires potentiels. Le but était de pouvoir analyser leurs besoins spécifiques et leurs points de divergences pour construire une vue globale sur la variabilité dans ce domaine d'application. Cette étape ressemble à celle de l'analyse du domaine proposée par les LDP (voir section 4.2.1.1). Les variations identifiées sont les suivantes :

1. *Workflow :* cette variation concerne le workflow de validation des résultats d'une simulation (voir figure 6.1 (d)). D'après l'analyse des besoins des locataires potentiels, deux types de workflow ont été identifiés : (i) le *workflow rapide* dans lequel seul l'utilisateur courant valide les résultats des simulations, et (ii) le *workflow lent* où la simulation doit être validée par plusieurs utilisateurs (ex : le directeur, le responsable de production, etc.). Dans ce dernier cas, l'application doit envoyer des e-mails à ces utilisateurs, contenant un lien vers une page Web affichant les résultats de simulation, et sur laquelle ces utilisateurs peuvent la valider ou non. Pour envoyer ces e-mails, nous avons intégré un service de messagerie externe. La sélection du workflow lent permet aussi aux locataires d'inclure un service supplémentaire qui consiste à stocker et gérer les adresses électroniques dans la base de données pour éviter leur saisie à chaque simulation.

2. *Transport et qualité d'ingrédients :* ces deux variations sont principalement liées aux différences entre les modèles économiques adoptés par les locataires. En effet, certains locataires ont demandé un autre transporteur d'ingrédients car ils trouvent que celui proposé par l'application est trop coûteux. De même, certains locataires ont demandé d'avoir une meilleur qualité d'ingrédients fournis pour maintenir leurs recettes dans un niveau de qualité relativement élevé. Pour tenir compte de ces deux variations, nous avons décidé d'ajouter un deuxième service de transport économique avec un lent délai de livraison comme compromis, et un nouveau service de fourniture d'ingrédients de haute qualité avec des prix d'ingrédients forcément plus élevés.
3. *Simulation :* cette variation vise à adapter le mode de fonctionnement du service de simulation pour aboutir à des résultats de simulation précis. En effet, nous avons identifié de différents facteurs qui affectent les résultats de simulation qui sont liés aux propriétés spécifiques de chaque locataire ou d'un ensemble de locataires (ex : la consommation d'énergie à l'usine, la température, la perte d'ingrédients sur la chaîne de production, la qualité des machines de fabrication, etc.). Ces facteurs doivent être pris en compte lors de la simulation d'une recette.
4. *Emplacement de données :* cette variation a été élaborée suite à l'identification d'un ensemble de mesures de sécurité spécifiques à certains locataires qui leur interdisent de sauvegarder leur données dans une base de données externe. Ces locataires voulaient installer la base de données dans leur propres infrastructures informatiques et ils sont prêts à payer les frais supplémentaires pour réaliser et maintenir cette fonctionnalité.
5. *Partage de données métiers :* cette variation est justifiée par le fait que certains locataires appartiennent au même groupe industriel et ils veulent partager les informations de leurs recettes pour tirer réciproquement parti de leurs connaissances et expertises.

6.2.2 Modélisation de la variabilité dans l'application FIA

Notre idée clé pour réagir face au défi de la gestion de la variabilité dans les applications SaaS mutualisées est de modéliser explicitement cette variabilité. Ceci peut être dirigé par trois facteurs :

- *Les exigences de locataires :* les locataires ont des besoins différents et des attentes différentes de l'application qui doivent être formellement exprimés, et ce d'une manière proche de leur compréhension et leur terminologie du domaine pour faciliter la communication de la variabilité de l'application avec eux par le fournisseur de SaaS .
- *La compréhension des développeurs :* les développeurs ont une vue plus technique de la variabilité. Ainsi, nous devons affiner la variabilité dirigée par les exigences des locataires dans un modèle plus orienté vers la compréhension des développeurs et les concepts qu'ils traitent lors du développement. Toutefois, cela n'empêche pas la possibilité d'un consensus sur une partie de modèle de variabilité commune entre les développeurs de SaaS et les locataires dans le cas où la variabilité traite des aspects de l'application communément compris.
- *L'architecture de l'application :* tel que rappelé en début du chapitre, nous considérons l'AOS comme le modèle d'architecture utilisé pour construire nos applications. Ce dernier introduit ses propres artéfacts de développement. Ceux affectés par la variabilité doivent être identifiés et documentés. Cela nous permettra d'envisager les mécanismes et les stratégies d'adaptation les plus appropriées, et en plus, de maintenir une traçabilité de la variabilité entre la variabilité séparément modélisée et sa résolution sur les différentes couches.

En outre, le *binding* de la variabilité (c.à.d. la sélection des variantes souhaitées sur les points de variations, confère définition en section 4.2.2.1) ne dépend pas seulement des exigences d'un locataire,

mais également des choix de variantes déjà effectués par l'ensemble de locataires de l'application. Savoir quels choix ont été effectués par les locataires existants permet d'améliorer la prise de décision et de minimiser les coûts de gestion de l'application. À titre d'exemple, et pour répondre aux différentes exigences des locataires concernant la disponibilité et la performance de l'application, FIA est déployée sur deux serveurs différents. Le premier serveur est loué chez OVH (un fournisseur IaaS) et est caractérisé par une puissance élevée en termes de bande passante et de nombre de CPU. Alors que le deuxième serveur est à puissance normale et loué chez Amazon sous forme d'une image EC2. Un équilibreur de charge est ainsi installé pour acheminer les appels des locataires au serveur qui correspond à leurs exigences. Dans le cas où le serveur à puissance normale devient sur-chargé, un nouveau locataire est mieux acheminé vers le serveur à puissance élevée même s'il ne présente aucune exigence en termes de performance. Cela évitera l'installation d'un nouveau serveur chez Amazon. Par conséquent, le binding de la variabilité dans les applications SaaS mutualisées ne dépend pas toujours des exigences de chaque locataire, mais aussi d'autre locataires et de la situation actuelle du système.

Pour modéliser la variabilité de l'application ainsi que le binding de celle-ci, nous exploitons avant tout les techniques de modélisation proposées par le domaine de LDP, et plus précisément OVM. Mis à part les points de variation et les variantes, OVM propose trois concepts de modélisation que nous trouvons assez pertinents pour notre contexte et pour répondre aux préoccupations de modélisation de la variabilité susmentionnées. Ces concepts sont les suivants : *la variabilité externe*, *la variabilité interne* et *la dépendance avec les artéfacts* (confère section 4.2.2.3). Dans ce qui suit, nous allons décrire leur intérêt par rapport à notre contexte d'application SaaS.

- *La variabilité externe :* en termes simples, la variabilité externe dans OVM est celle communiquée aux clients de la ligne de produits. Dans le contexte de SaaS, la variabilité dirigée par les exigences de locataires correspond bien à ce concept.
- *La variabilité interne :* selon OVM, cette variabilité n'est visible que pour les développeurs de la ligne de produits. En tant que telle, cette variabilité est clairement en corrélation avec la variabilité dirigée par la compréhension des développeurs de SaaS.
- *La dépendance avec les artéfacts :* ce concept d'OVM permet d'établir les liens nécessaires entre les informations de variabilité (points de variation et variantes) et les artéfacts affectés. Dans notre contexte, ces artéfacts correspond aux couches de l'AOS et les éléments architecturaux situés sur chaque couches, tels que les processus, les services, les composants, etc. (confère la section 2.3.5 pour la définition des couches)

Dans la suite, nous suivrons cette distinction entre les concepts d'OVM lors de la modélisation de la variabilité dans notre application mutualisée. Cela nous permet de pouvoir raisonner sur ce modèle afin de soutenir le processus de décision concernant la résolution de celle-ci. La partie supérieure de la figure 6.2 représente le modèle de variabilité de l'application FIA. Dans ce modèle, la variabilité externe concerne généralement le choix d'un fournisseur d'ingrédients (*PV Qualités d'ingrédients*), d'un transporteur (*PV Transporteur*) et de l'un des deux Workflows supportés (*PV Workflow*). Un locataire peut également choisir le niveau de performance requis (*PV Disponibilité et performance*) et s'il souhaite partager les informations de ses recettes (*PV partage d'informations*) avec d'autres locataires à condition que sa base de données ne soit pas installée sur sa propre infrastructure (*PV Emplacement de données*).

La variabilité interne dans ce modèle concerne l'affinement de la variabilité sur les données et sur la performance de l'application étant donné que leur formalisation est différente entre les développeurs et les locataires. Par exemple, une décision à effectuer par un locataire sera de choisir l'emplacement de ses données en étant conforme à ses propres mesures de sécurité. En revanche, du point de vue des développeurs, il s'agit d'un modèle de séparation de données à adopter (*PV Modèle de séparation*). De même, les développeurs préfèrent raisonner sur la nature du serveur d'application utilisé en termes de

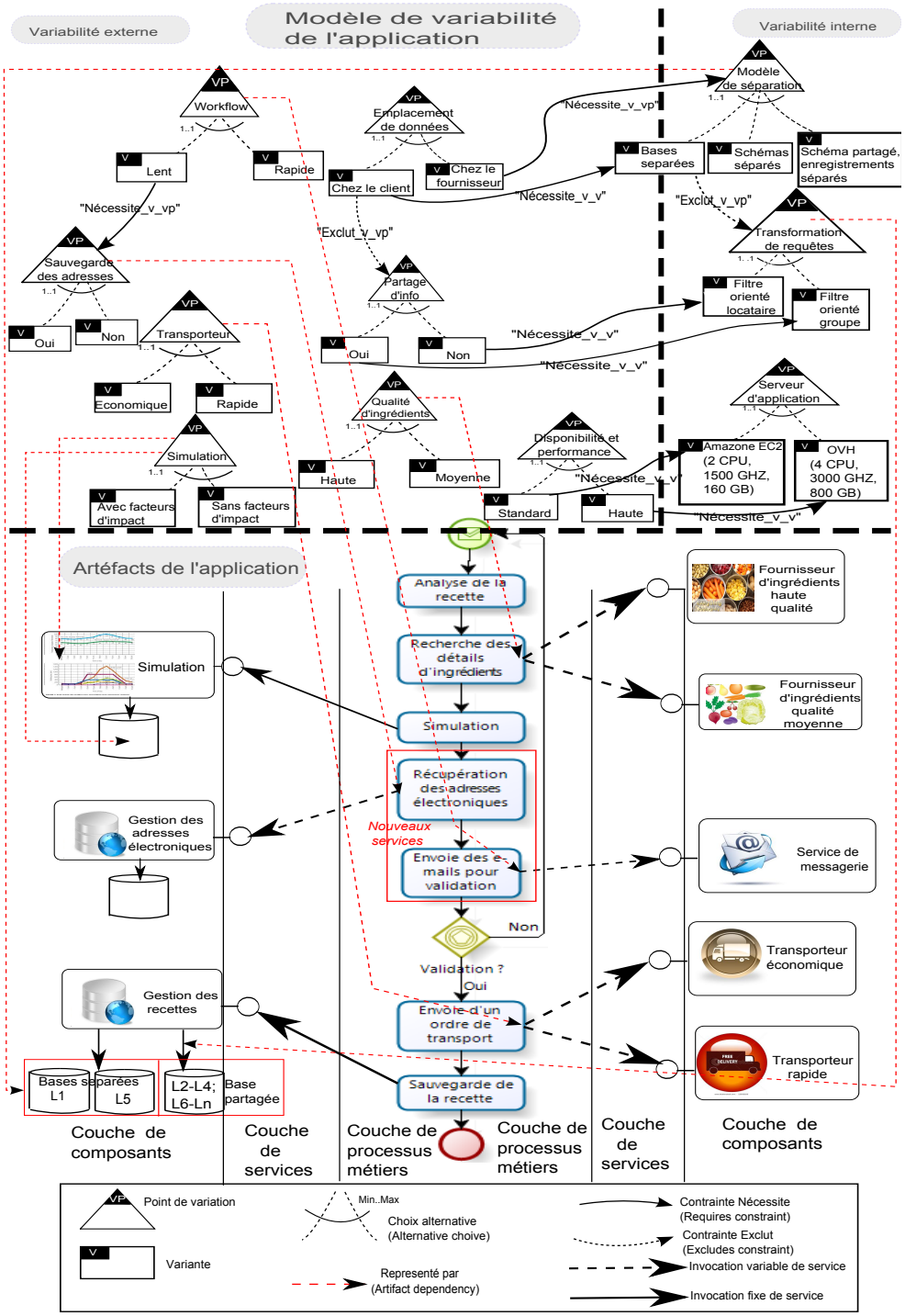


Figure 6.2 – Modélisation de la variabilité dans l'application FIA selon OVM

ses caractéristiques et ses propriétés (*PV Serveur d'application*), alors que les locataires sont concernés par le niveau de performance requis.

6.2.3 Résolution de la variabilité dans l'application FIA

Après avoir modéliser la variabilité, les développeurs doivent prendre la décision concernant la projection de cette variabilité sur les couches de l'application (processus métiers, services, composants et données) tout en identifiant les mécanismes et les stratégies de résolution de la variabilité nécessaire pour dynamiquement adapter l'application sur ces différentes couches. La projection de la variabilité nécessite une étape de réflexion afin de détecter la(les) couche(s) la(les) plus appropriée(s) pour résoudre chaque point de variation. À notre connaissance, cette étape ne peut pas être entièrement automatisée mais doit être guidée à travers l'évaluation de certaines propriétés qui influent la décision. La partie inférieure de la figure 6.2 montre les couches et les artéfacts de l'application FIA ainsi que ceux affectés par la résolution de la variabilité. Les artéfacts affectés sont explicitement modélisés à travers l'association *représenté par* (lignes en pointillé rouge) pour garder trace des décisions de résolution prises. Dans la suite, nous détaillons ces décisions pour résoudre un échantillon des points de variation parmi ceux présentés dans le modèle de variabilité. Les mécanismes de résolution que nous présentons exploitent les capacités des technologies de développement utilisées, et sont tous conçus de manière à interpréter un modèle unique de *fichiers de configurations* contenant les variantes choisies par les locataires.

PV Qualité d'ingrédients : selon la spécification de ce point variation, l'application doit interagir avec le service qui correspond aux besoins de chaque locataire en termes de qualité d'ingrédients requise. Puisqu'il s'agit de services externes dont nous n'avons pas le contrôle, la seule manière de résoudre cette variation est de dynamiquement sélectionner le service à invoquer. Comme nous l'avons déjà évoqué, la résolution de la variabilité dans cette application est basée sur l'exploitation des capacités de la technologie de développement utilisée. Ainsi, nous intégrons une technique fournie par l'outil WCF [**?**] appelée le *proxy dynamique* pour réaliser une invocation dynamique de services.

Dans WCF, un proxy représente une classe générée à partir du fichier WSDL d'un service à utiliser. L'objectif de cette classe est de masquer la complexité des mécanismes de communication avec le service et exposer localement ses méthodes proposées. Une fois que cette classe est *statiquement* générée, elle sera compilé et exécuter avec l'exécution de l'ensemble de services de l'application et ne peut plus être remplacer par un autre service, sauf dans le cas d'arrêt d'exécution et de recompilation (confère composition statique de services en section 2.3.4.2). Toutefois, dans le cas où le proxy d'un service a été *dynamiquement* généré (un proxy dynamique à travers un gestionnaire spécifique fourni par l'outil WCF, voir figure 6.3) il pourra être compilé et accédé par réflexion lors de l'exécution. Par conséquent, son remplacement par le proxy dynamique d'un autre service est possible. La figure 6.3 montre la différence entre la génération d'un proxy « statique » et un proxy dynamique. Ce dernier permet de créer un client de service lors de l'exécution en spécifiant uniquement l'adresse de ce service. À travers cette adresse, l'outil WCF télécharge le fichier WSDL du service et le transforme dynamiquement en une classe déjà compilée et physiquement stockée dans la mémoire vive. Cette classe pourra être ensuite accédée par réflexion pour exécuter la méthode souhaitée. Cette approche nous permettra d'ajouter autant de services alternatifs que nous souhaitons sur ce point de variation, sans arrêter l'exécution de l'application.

PV Simulation : la résolution de ce point de variation a pour but d'assurer une simulation précise à travers la prise en compte des propriétés spécifiques à l'usine de chaque locataire ou un ensemble de locataires. La complexité de ce point de variation réside dans le fait que chaque propriété spécifique (ex : la consommation d'énergie à l'usine, la température, la perte d'ingrédients sur la chaîne de production, la qualité des machines de fabrication, etc.) nécessite un traitement différent pour calculer son effet sur les résultats finaux de la simulation. Une solution rapide pour la résoudre

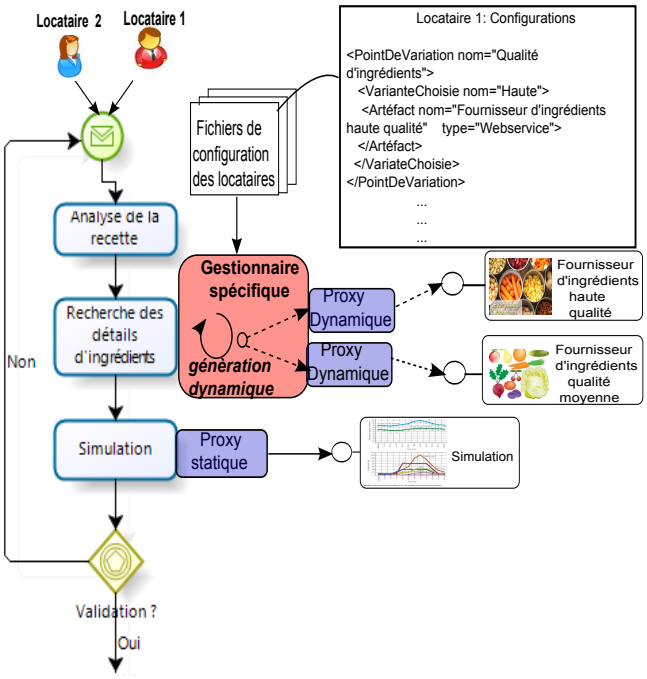
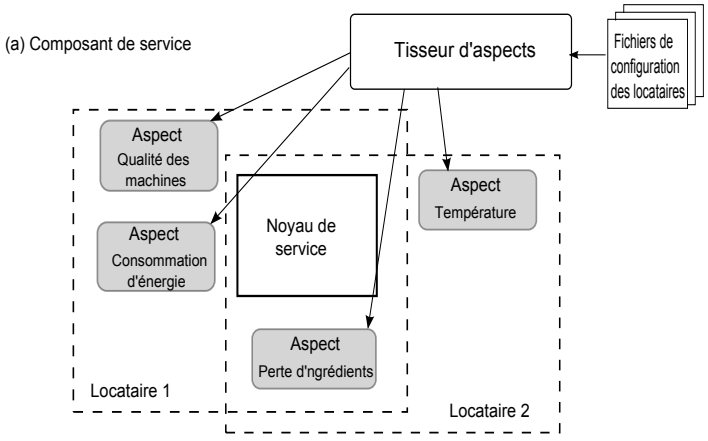


Figure 6.3 – Génération des proxy dynamiques pour adapter la composition de services

consiste à créer une version personnalisée (modification du code) du service pour chaque locataire ou ensemble de locataires dont les propriétés spécifiques ne sont pas déjà supportées par une version existante. Ces différentes versions (considérées comme alternatives du service de simulation) seront ensuite déployées sur le serveur de l'application et dynamiquement sélectionnées à travers la réutilisation du mécanisme d'adaptation existant (le proxy dynamique). Malgré sa contradiction avec le principe de mutualisation, cette solution peut rester valide et maintenable tant que le nombre de services personnalisés (à travers le changement du code) ne dépasse pas une certaine limite. Toutefois, la grande diversité des locataires concernant cette partie de l'application ne garantira pas le non-dépassement de cette limite, et par conséquent, il deviendra vite très compliqué de maintenir un nombre croissant d'instances différentes du service de simulation en cours d'exécution. Pour cette raison, nous avons décidé de résoudre ce point de variation au niveau du composant de service (variabilité de la logique, confère section 4.3.3) en modifiant son architecture interne (voir figure 6.4). La solution adoptée consiste à découper l'implémentation de ce service en deux parties. La première est un composant contenant le noyau du service qui encapsule le comportement commun entre les locataires, alors que la deuxième partie est spécifique à chaque locataire ou ensemble de locataires. Cette dernière récupère de la base de données la propriété correspondante et applique ensuite son effet aux résultats de simulation suivant une logique précise. De ce fait, ce point de variation est un exemple concret d'une projection de la résolution sur plusieurs couches de l'AOS : composant de service et base de données.

La deuxième partie du service est implémentée sous forme d'aspects qui seront injectés dans le noyau du service à travers un processus de tissage d'aspects. La figure 6.4 montre l'architecture interne du service de simulation. Le processus de tissage est généralement destiné à gérer des préoccupations considérées comme transverses (telles que l'application de l'effet des propriétés sur



(b) Base de données

LocataireID	Nom	Extension XML
1	Bel	<Propriétés> <Température code="Temp" Valeur="31° c"/> <PerteIngredients code="PI" Valeur="16%"/> </Propriétés>
2	Danone	<Propriétés> <ConsommationEnergie code="CE" Valeur="1051 kw"/> <PerteIngredients code="PI" Valeur="16%"/> <Qualitémachine code="QM" Valeur="Moyenne"/> </Propriétés>

Figure 6.4 – Architecture interne du service de simulation et utilisation des aspects pour résoudre la variabilité

les résultats de la simulation) exprimés sous formes des aspects à composer avec une décomposition primaire (le noyau de service dans ce cas).

Par ailleurs, un tisseur d'aspects pourrait soutenir le tissage soit au moment de compilation (statique) soit au moment de l'exécution (dynamique) [**?**]. Nous discutons les problèmes de tissage et leurs implications uniquement pour notre cas d'utilisation. Dans le cas d'un tissage statique, à chaque création d'une nouvelle propriété spécifique pour un locataire, un aspect correspondant sera conçu et statiquement tissé en re-compilant le service. Vu que cela cause l'arrêt du service, tous les locataires hébergés seront affectés. Pour une application SaaS mutualisée, cet arrêt ne peut pas être toléré, et par conséquent, une approche de tissage dynamique des aspects doit être adoptée. Dans l'état de l'art, il existe plusieurs approches qui ont été proposées dans le cadre des aspects pour réaliser un tissage dynamique. Plus précisément, les auteurs dans [JB09] discutent d'une approche d'adaptation dynamique des composants de services à travers l'utilisation de ce type de tissage. Nous avons adopté une approche similaire dans notre contexte, et nous sommes rendus capable d'ajouter autant de nouveaux aspects gérant les propriétés spécifiques à chaque locataire, sans arrêter l'exécution du service. La création et le tissage dynamique des aspects est effectué à travers l'utilisation des capacités fournies par le cadre *PostShap* [**?**] spécifique à la définition et le tissage des aspects pour les applications basées sur l'environnement de développement .NET.

PV Partage d'informations : la résolution de ce point de variation permet de partager des données métiers entre les locataires en accord avec le privilège approprié (les locataires appartiennent au même groupe), ou bien de garder leurs données totalement séparées. Logiquement, une telle résolution

nécessite d'étendre le mécanisme existant de sécurisation d'accès aux données (voir section 5.2.3). Celui-ci est implémenté au niveau de l'application à travers un composant générique d'accès aux données utilisé par tous nos services développés en interne. Comme la manipulation des requêtes au niveau de ce composant est simple à implémenter et à maintenir, la méthode adoptée pour résoudre ce point de variation consiste à étendre l'algorithme existant de transformation des requêtes (générées par ce composant) en injectant des filtres spécifiques pour permettre le partage. Cette manière de résolution est transparente aux développeurs de services et améliore ainsi leur productivité. L'architecture que nous avons appliqué pour sécuriser l'accès aux données de locataires est expliquée en détails dans [**?**].

Une raison supplémentaire nous a conduit à utiliser cette approche pour résoudre ce point de variation est l'existence des mesures de sécurité spécifiques à certains locataires qui obligent à l'installation de bases de données séparées. En effet, la prise en compte de ces mesures sera compliquée si les développeurs s'occupent eux-mêmes de la sécurisation de données, tandis que la transformation dynamique des requêtes permet une prise en charge native de cette sécurisation. Lors de l'interaction avec une base de données séparée, il suffit juste de désactiver le mécanisme de transformation et renvoyer la requête dans sa forme initiale.

6.2.4 Manques identifiés

Avant tout, nous précisons que l'application présentée dans ce chapitre est un exemple simplifié du cas réel. Celui-ci implémente plus de fonctionnalités que celles mentionnées et gère un modèle de variabilité plus complexe, qui évolue avec le temps et l'hébergement de nouveaux locataires. Actuellement, nous hébergeons une dizaine de locataires qui sont tous servis depuis deux serveurs sur lesquels la même instance de l'application est installée et synchronisée en permanence. Nous recevons aussi de manière perpétuelle des prospects (locataires potentiels) intéressés par l'application et souhaitant l'utiliser. Pour organiser la gestion de ces derniers, nous avons établi un processus de gestion spécifique (voir figure 6.5) qui diffère par rapport aux processus de gestion classiques en raison de la mutualisation. Les étapes de ce processus sont décrites comme suit : au début, un premier contact s'établit avec le prospect afin d'avoir une idée plus claire sur ses besoins et la raison pour laquelle il souhaite utiliser l'application. Cette étape se termine par la prise d'un rendez-vous pour une démonstration de l'application (soit en se déplaçant chez lui, soit à travers une visioconférence). La deuxième étape consiste à créer un nouveau locataire de l'application avec une configuration de base qui correspond le plus aux besoins identifiés lors du premier contact. Ensuite, la troisième étape consiste à faire la démonstration. Si le prospect s'avère intéressé, la quatrième étape consiste à lui fournir une période de test gratuite de 15 jours durant laquelle son statut passe à locataire de *test*. Cette période lui permet d'apprécier pleinement l'application ou décider de ne pas continuer dans le cas où l'application ne fournit pas la solution recherchée. Toutefois, si le prospect décide de continuer, la cinquième étape sera la signature d'un contrat d'engagement où son statut passe à locataire *actif*. Dans le cas contraire, le locataire de test déjà créé devient désactivé.

Ce processus a été bien établi dans notre contexte pour la gestion des prospects et a été aussi utilisé pour d'autres applications mutualisées. Probablement le niveau élevé de technicité métier requis et présent dans l'application FIA a motivé le processus de gestion de prospects sus mentionné. Ceci dit, le problème principal avec ce processus provient du non-respect de la caractéristique self-service de services du cloud (voir section 2.2.1). En effet, la majorité des étapes présentées dans ce processus et surtout celles dédiées à la création et la préparation de l'espace d'utilisation de chaque nouveau locataire, nécessitaient l'intervention d'un membre de notre équipe pour les exécuter alors que la caractéristique

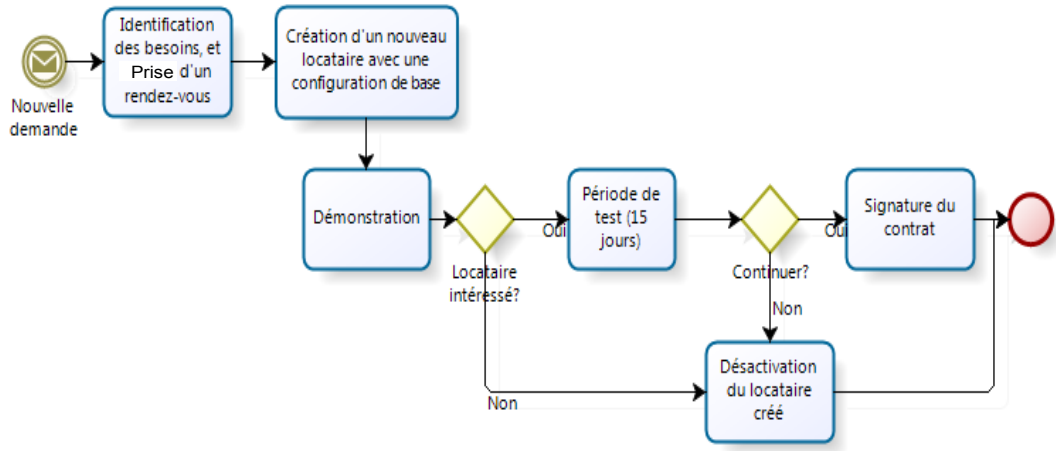


Figure 6.5 – Processus de gestion des prospects

self-service de services du cloud consiste à minimiser autant que possible ces interventions avec un effort minimum d'administration.

Cela ne posait pas des vrais problématiques au départ (début de la phase de commercialisation de l'application) car la fréquence de nouveaux prospects était relativement faible. Au fil du temps, cette fréquence a subit une augmentation importante qui a causé une inefficacité de gestion à travers ce processus et une consommation considérable du temps d'analyse, de vérification et d'accompagnement des locataires lors du changement qui ont dépassé les capacités des ressources humaines existantes.

Pour améliorer ce processus et réduire le temps qu'il consommait, nous devions être capable de l'automatiser. Ainsi, nous devions permettre aux prospects de créer eux-même leur espace d'utilisation, de les configurer et tester pendant la période indiquée sans aucune intervention de notre part ni de la part des revendeurs existants (sauf bien évidement dans le cas urgent). Toutefois, le principal obstacle devant cette amélioration était les limites d'OVM face à l'exposition directe du modèle de variabilité aux locataires (potentiels ou actuels) pour qu'ils effectuent eux-même le binding de variabilité. Une des raisons de ces limites est l'absence de moyens pour contrôler l'accès au modèle de variabilité pour interdire, par exemple, le choix de certaines variantes par les locataires non-autorisés.

À titre d'exemple, et par rapport à la spécification de la variabilité dans notre application, nous devons interdire le choix de la variante *Standard* du PV *Disponibilité et performance* si le nombre de locataires qui l'avaient déjà choisi dépasse 200, car cela indique que le serveur à disponibilité standard est surchargé. Aussi, nous devons être capables d'exprimer que la variante *Rapide* du PV *Transporteur* ne peut pas être choisie par les locataires qui gèrent des usines situées hors du territoire français, car le transporteur rapide travaille uniquement à un niveau local.

D'autres situations ont été ainsi identifiées dans d'autres applications, et qui nous ont aidé à clarifier la nature des limites d'OVM pour des applications SaaS mutualisées et la manière dont nous pouvons les améliorer. En effet, la source de ces limites d'OVM réside essentiellement dans sa définition statique des contraintes de dépendances entre les éléments de la variabilité, celles utilisées pour vérifier la consistance et l'exactitude d'une configuration. En effet, ces contraintes servent à exclure ou nécessiter le choix de certaines variantes ou même la présence de certains points de variations, mais uniquement en fonction

d'autres variantes ou points de variation dans le modèle, tout en ignorant les contextes des locataires principalement concernés par la configuration. Cela est dû à l'hypothèse initiale d'OVM qui part du principe que la dérivation d'un produit ne pourra être effectuée que par les membres de l'organisation qui a construit la ligne de produits.

Dans notre contexte de mutualisation de SaaS, cette hypothèse n'est plus valide. La variabilité de l'application doit absolument considérer des informations sur les locataires pour adapter la structure du modèle en manipulant les contraintes à travers ces informations. Le but est de fournir à chaque locataire une vue différente du même modèle de variabilité tout en gardant à la fois son unicité et uniformité. Dans la section suivante, nous allons présenter notre nouvelle approche de modélisation de la variabilité pour répondre aux manques identifiés.

6.3 Notre méta-modèle de variabilité : concepts et formalisations

6.3.1 Motivations

Notre nouvelle approche de modélisation de la variabilité est conçue de façon à tenir compte de la spécificité des applications SaaS mutualisées. Cette spécificité découle de leur structure organisationnelle dans laquelle une séparation est établie entre la construction du logiciel à effectuer par le fournisseur, et sa configuration à effectuer directement par les locataires à n'importe quel moment. Cela implique l'introduction de nouveaux concepts de modélisation qui permettent de contrôler l'accès aux modèles de variabilité une fois ces derniers sont exposés.

Une autre exigence à ce niveau concerne la résolution de la variabilité une fois modélisée. Par le terme « *résolution* » nous entendons le processus d'exécution des variantes choisies dans l'architecture de l'application. Pour cela, le modèle de variabilité doit contenir les informations nécessaires pour pouvoir soutenir une telle résolution. Par exemple, lorsqu'il s'agit d'une variation sur la couche de composition de services, les différentes variantes doivent fournir les informations nécessaires à l'invocation des services alternatifs qu'elles modélisent. Cela revient à fournir les adresses des services, les opérations à invoquer, les types de messages à échanger et ainsi que d'autres informations nécessaires pour effectuer dynamiquement une telle invocation. De même, les variantes de l'ergonomie doivent fournir les informations nécessaires à l'adaptation des interfaces graphiques. Par conséquent, une approche de modélisation doit être d'un côté assez technique pour qu'elle puisse décrire ces informations, et de l'autre côté, elle doit être suffisamment indépendante de la technologie de développement pour être réutilisable par différents types d'artéfacts. Cela exige que le modèle de variabilité et le code de l'application soient le plus expressifs possible pour qu'une adaptation puisse être réalisée par la combinaison de ces deux. Ainsi, notre méta-modèle de variabilité est dédié aux applications Web basées sur un modèle de construction standard tel que l'AOS, et ses artéfacts de développement communément utilisés.

6.3.2 Méta-modèle

À plusieurs reprises, nous avons discuté de la nécessité d'un modèle de variabilité orthogonal et montré les limites des techniques de modélisation offertes par les lignes de produits logiciels face à la spécificité des application SaaS mutualisées. Dans ce qui suit, les nouveaux concepts de modélisation liées à ce type d'applications et ainsi que les concepts réutilisés à partir d'OVM sont présentés. La figure 6.6 montre notre nouveau méta-modèle de variabilité.

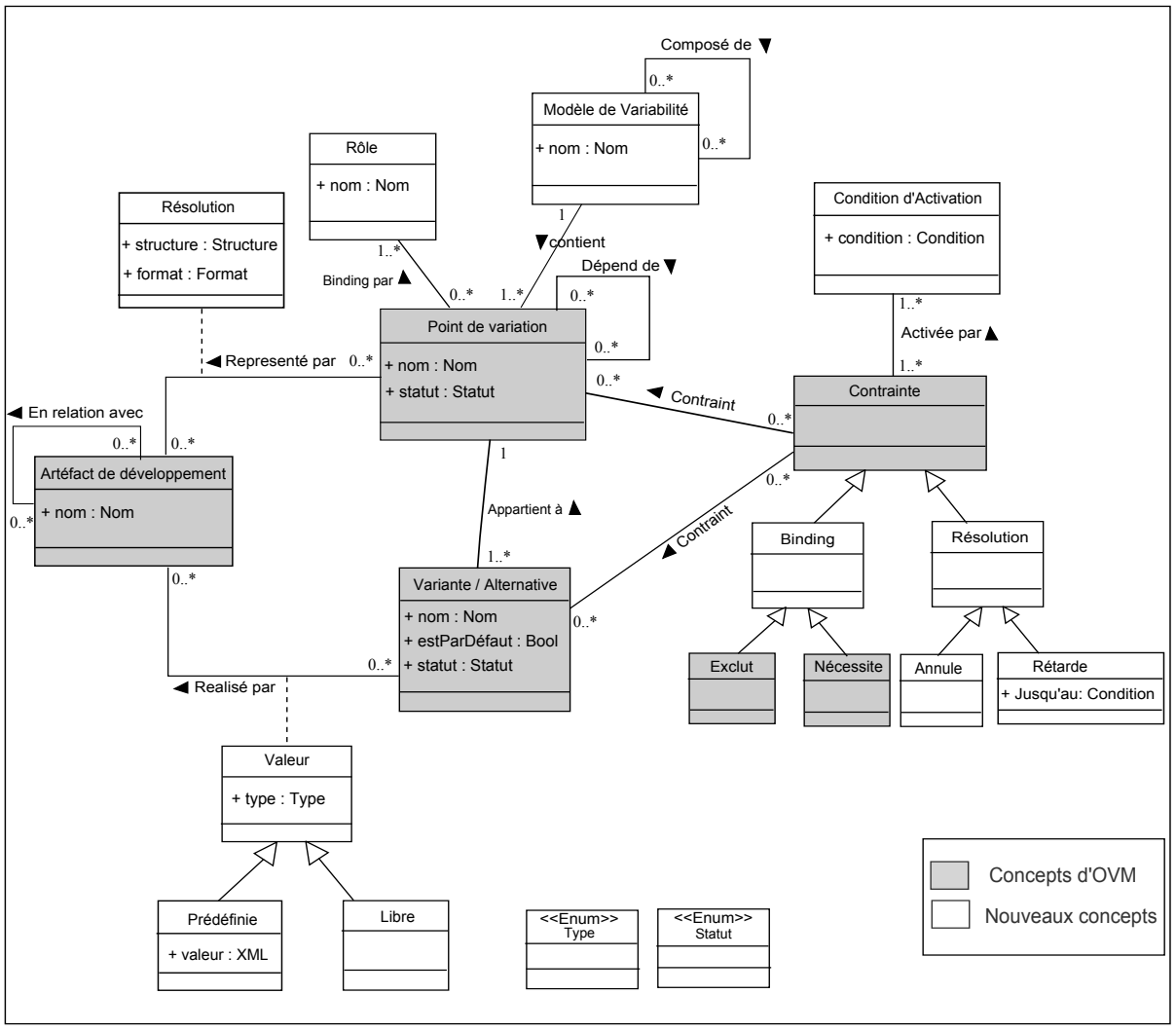


Figure 6.6 – Méta-modèle de variabilité

6.3.2.1 Survol

La variabilité selon ce méta-modèle s'exprime à travers un *modèle de variabilité*. Il est instancié par un fournisseur de service (SaaS ou service AOS) et peut être composé avec d'autres modèles de variabilité appartenant à d'autres services. Son instanciation doit être suivie par la définition de l'ensemble des *points de variation* qui lui appartiennent. Ces derniers représentent les éléments principaux du modèle et sont attachés à des *rôles* pour permettre à plusieurs personnes d'intervenir à la configuration de l'application. Un point de variation définit également un ensemble de *variantes* qui modélisent les différentes options possibles sur ce point. Celles-ci sont associées à des *artéfacts de développement* qui documentent les éléments architecturaaux qui les réalisent. Une classe *valeur* de cette association (variante-artéfact) est introduite pour saisir les informations nécessaires à la résolution de la variabilité. Chaque valeur est soit prédéfinie par le fournisseur, soit libre à saisir directement par les locataires.

D'ailleurs, les points de variation et les variantes peuvent être conditionnellement contraints à travers la définition des *conditions d'activation* attachées aux *contraintes* du modèle. Ces conditions d'activation détiennent chacune une condition à modéliser sous forme d'une expression qui évalue à vrai ou faux pour indiquer la validité des contraintes lors de la configuration.

6.3.2.2 Concepts : présentation détaillée

Modèle de variabilité : lors de la modélisation de la variabilité, le modèle de variabilité est le premier concept à spécifier par instanciation du méta-modèle. Il permet de regrouper tous les éléments de la variabilité appartenant à une application SaaS (ou service de l'AOS). Bien que le méta-modèle en tant que tel représente un modèle de modèles de variabilité, l'utilité de ce concept est d'établir une composition entre plusieurs modèles pour répondre au problème de composition de services variables (confère section 4.4.2). Un service variable est un service composé dans l'application et présente une variabilité indépendamment de cette application qui le compose.

Association* : la composition des modèles de variabilité peut être réalisé à travers l'association réflexive *composé de* sur la classe modèle de variabilité. Cette association permet ainsi de définir la composition de modèles de variabilité en fonction de la composition des services (variables) dans l'application (voir figure 6.7). Ceci est bénéfique dans le cas où l'application est composée de plusieurs services variables appartenant chacun à un fournisseur différent. Ainsi, la variabilité de chaque service peut être séparément modélisée par son fournisseur et ensuite composée avec la variabilité de l'application qui les contient pour avoir une vision globale de la variabilité dans le système entier. La figure 6.7 montre un exemple de composition de modèles.

Dans cet exemple, nous supposons que le service de messagerie (service externe) réutilisé par l'application FIA présente une variabilité sur le type du message à envoyer. Le fournisseur de ce service modélise sa variabilité à travers un seul point de variation *Type de Message* et deux variantes *SMS* et *E-mail* attachés au modèle *MV-Messagerie* qui regroupe la variabilité de ce service. La composition de ce service dans l'application FIA exige la composition de son modèle de variabilité aussi. Selon la sémantique de cette composition, le modèle de variabilité principal importe tous les éléments du modèle composé et est capable de les contrôler à travers l'utilisation des contraintes. Par exemple, nous nécessitons (à travers la contrainte de spécification *nécessite* du méta-modèle) le choix de la variante *E-mail* du modèle *MV-Messagerie* pour tous les locataires qui choisissent le Workflow *Lent* de l'application FIA. L'établissement des contraintes entre plusieurs modèles nécessite tout d'abord leur composition. Ces contraintes suivent toujours le sens de la composition et ne peuvent pas être établies dans le sens inverse sauf si la relation de composition est réciproque entre les modèles. Bien évidement, pour gérer cette composition et veiller à ce que les règles de composition soient respectées, un mécanisme de gestion est nécessaire. Pour cela, nous proposons un cadre de modélisation de la variabilité (voir figure 6.10) en nous basant sur les concepts de ce méta-modèle, tout en intégrant les mécanismes nécessaire pour effectuer cette modélisation d'une manière correcte. Les modèles de variabilité résultants peuvent être ensuite exportés sous formes des fichiers XML pour faciliter leur traitement et leur portabilité.

Formalisation : soit *MV={...,mvi,...}* un ensemble de modèles de variabilité. La fonction *Compose : mv→MV* associe un modèle de variabilité à un ensemble d'autres modèles pour créer une composition entre-eux. Par l'application de cette fonction, le modèle *mv* devient le modèle principal de la composition qui importe automatiquement tous les éléments des autres modèles composés. Pour récupérer l'ensemble des modèles de variabilité composés avec un modèle précis, la

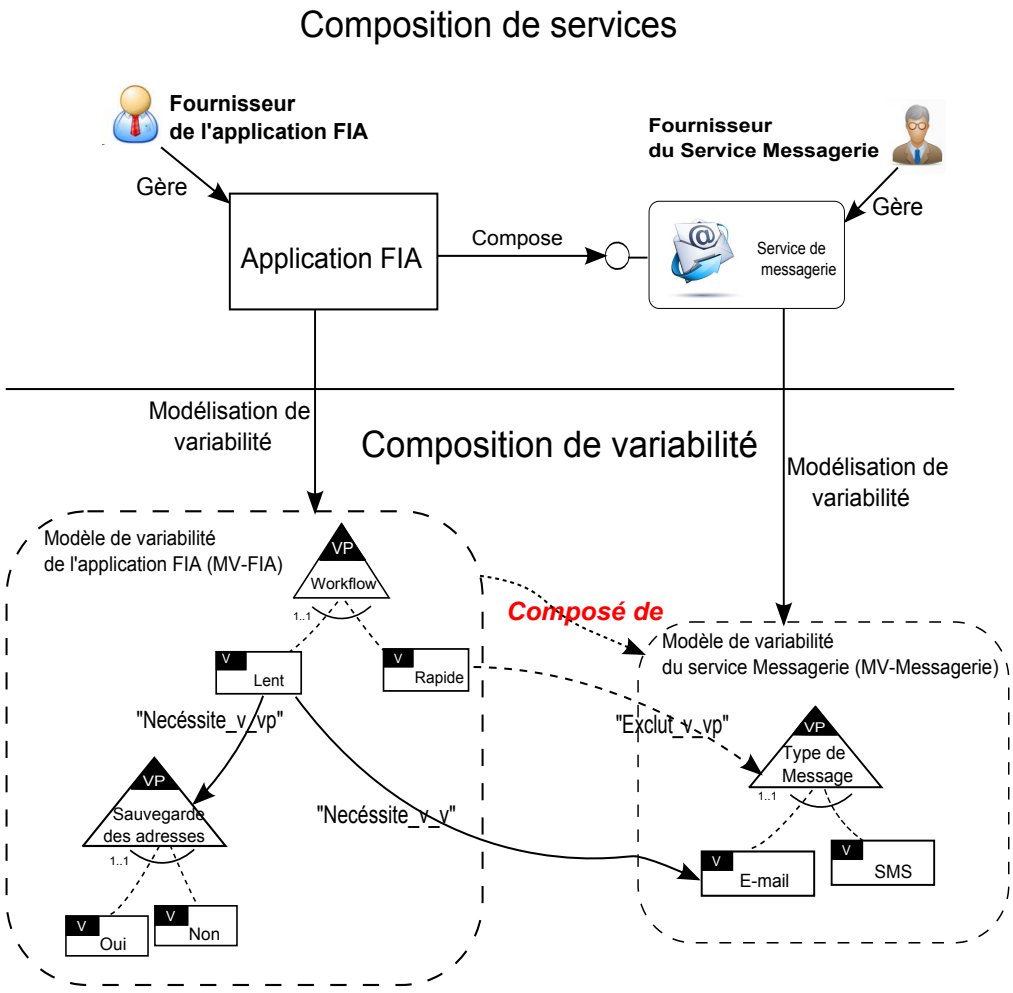


Figure 6.7 – Composition de modèles de variabilité suite à une composition de services

fonction *Compose(mv)* est ainsi utilisée. Cette formalisation est appliqué de la même manière pour le reste des concepts.

Point de variation : ce concept est réutilisé à partir d'OVM. Il constitue le concept central de toute approche de méta-modélisation de variabilité. Dans les lignes de produits logiciels, ce concept est bien connu. Les auteurs dans [**?**] le définissent comme *« un ou plusieurs endroits dans le système logiciel où une variation se produira »* , alors que les auteurs dans [PBVDL05] voient son utilité comme *« une représentation du sujet de la variabilité dans les artéfacts du domaine »* . D'un point de vue général, un point de variation peut être considérée comme un ou plusieurs endroits dans une ligne de produits logiciels qui sont variables et qui doivent être résolus lors de la création d'un nouveau produit. Dans notre contexte, ce concept identifie bien les éléments architecturaux variables et réparties sur les différentes couches de l'application.

Une limitation d'OVM a été identifiée à ce niveau concernant l'ordre dans lequel le binding des points de variation doit être effectué. En effet, un locataire doit parfois faire des choix dépendant d'autres choix déjà effectués. La sémantique d'une dépendance entre deux points de variation indique que le binding du premier doit être absolument effectué avant celui du deuxième.

Dans notre exemple, la saisie d'un titre spécifique de l'application pour un locataire crée instantanément un répertoire nommé avec le titre saisi, et dans lequel les documents appartenant à ce locataire seront sauvegardés, dont son logo spécifique de l'application. De ce fait, un point de variation concernant le logo de l'application dépend de celui du titre et doit être exposés aux locataires après que le binding du deuxième point de variation sera effectué.

Association* : pour modéliser cette dépendance, l'association *Dépend de* est introduite dans notre méta-modèle. Selon cette association, un point de variation peut dépendre d'un ou plusieurs autres points de variation. De même, plusieurs points de variation peuvent dépendre d'un seul autre. Contrairement à la relation de composition entre les modèles de variabilité, la dépendance entre deux points de variation ne peut pas être réciproque (non symétrique).

Les relations de dépendances peuvent également cibler des points de variation importés d'autres modèles de variabilité. Il est également permis d'introduire de nouvelles dépendances entre les points de variation importés de différents modèles de variabilité. Cependant, il n'est pas autorisé de supprimer les dépendances importées d'autres modèles. En effet, la suppression de ces dépendances modifierait la séquence possible dans laquel les points de variation doivent être traités et pourra donc implicitement introduire une mauvaise configuration. Suivant ce raisonnement, nous distinguons deux types de points de variation : les *points de variation indépendants* qui seront les premiers à être exposés aux locataires durant le processus de binding direct de la variabilité, et les *points de variation dépendants* qui seront exposés tout de suite après.

Formalisation : soit *PV = {...,pvi,...}* un ensemble de points de variation et PV_{mv}=π_1(mv) l'ensemble des points de variation appartenant au modèle de variabilité *mv* $\in$ *MV*, incluant les points de variation importés depuis les autres modèles composants. Cette importation des points de variation implique que la condition suivante doit rester toujours valable pour chaque modèle de variabilité *mv* $\in$ *MV* : $\forall$ *w* $\in$ *Compose(mv)* : $\pi_1(w) \subseteq \pi_1(mv)$. Cette condition signifie que l'ensemble des points de variation importés depuis un modèle est toujours un sous-ensemble de l'ensemble de points de variation du modèle principal.

Pour créer des dépendances entre les points de variation, la fonction *Dep : pv*$\rightarrow$*PV* est utilisée. Celle-ci crée une dépendance entre un point de variation et un ensemble d'autre points de variation. Cette création de dépendances nécessite ensuite la différenciation entres les points de variation indépendants et les points de variation dépendants. Pour récupérer les points de variation indépendants, la fonction suivante est utilisée :

$PV^{Independants}(mv) = \{ pv \in PV_{mv} : Dep(pv) = \Phi \}$.

L'ensemble des points de variation dépendants est ainsi déduit par différence entre l'ensemble global de points de variation dans une composition de modèles et l'ensemble des points de variation indépendants :

$PV^{Dependants}(mv) = PV_{mv} - PV^{Independants}(mv)$.

Condition d'activation : ce concept est introduit dans notre approche pour fournir un certain niveau de flexibilité de modélisation, permettant de réagir d'une manière simple face à des situations de modélisation complexes qui exigent d'exclure ou nécessiter (à travers les contraintes de spécification

Exclut et *Nécessite* du méta-modèle) certaines variantes ou points de variations du modèle de variabilité en fonction des informations sur les locataires (confère section 6.2.4). Bien évidemment, cela nécessite la prise en considération de ces informations lors de la modélisation de contraintes et de les évaluer au moment de la configuration de l'application par rapport à l'identité du locataire connecté. Pour ne pas modifier la sémantique des contraintes d'OVM, ce concept de condition d'activation est introduit. Une condition d'activation modélise explicitement une condition sous forme d'une expression à évaluer dynamiquement sur la base des informations sur les locataires et le binding de variabilité déjà effectué. Une contrainte du modèle sera ainsi associée à une ou plusieurs conditions d'activation et sera activée par l'évaluation à vrai de l'ensemble des conditions de ses conditions d'activation. Chacune de ces dernières détient une condition qui est une expression conçue de manière à être évaluée à *vrai* ou *faux* à travers l'analyse des informations spécifiques sur le locataire configurant l'application.

La structure d'une condition

```
1 Condition = [Fonction, Opérateur de comparaison, Valeur]
```

La structure d'une condition est décomposée en trois parties : la première est une fonction à sélectionner parmi un ensemble des fonctions prédéfinies par notre cadre de modélisation, dont chacune est dédiée à retourner une information spécifique sur le locataire. La deuxième partie est un opérateur de comparaison devant comparer la valeur retournée par la fonction avec la valeur saisie dans la troisième partie de la structure. L'évaluation à vrai d'une condition valide sa condition d'activation qui active à son tour une ou plusieurs contraintes. Aussi, la même contrainte peut être activée par plusieurs conditions d'activation. La sémantique de cette relation entre les conditions d'activation et les contraintes précise que celles-ci deviennent activées uniquement dans le cas où l'ensemble des conditions appartenant aux conditions d'activation associées à ces contraintes sont évaluées à *vrai*.

La figure 6.8 montre un exemple d'instanciation du méta-modèle avec l'utilisation de conditions d'activation. Dans cet exemple, nous pouvons remarquer que les contraintes ne sont plus statiquement définies entre les variantes et les points de variation. Elles sont attachées à des conditions d'activation qui sont évaluées à travers les expressions de leurs conditions. La définition des conditions se base, tel que nous l'avons évoqué, s'opère sur l'utilisation d'un ensemble de fonctions prédéfinies par notre cadre de modélisation. Ces fonctions sont les suivantes :

- `VarianteChoisie(PV)` : cette fonction analyse le binding de variabilité déjà effectué par un locataire précis. Elle retourne la variante choisie par le locataire sur le PV indiqué.
- `NbLocataires(Variante)` : cette fonction analyse le binding de variabilité effectué par tous les locataires. Elle retourne le nombre de locataires qui ont déjà choisie la variante indiquée.
- `LocataireInfo(Propriété)` : cette fonction analyse les propriétés du locataire. Elle retourne la valeur de la propriété indiquée pour le locataire courant.

D'autres fonctions peuvent être ajoutées selon les besoins. Nous allons détailler l'utilisation de celle responsable d'analyser les propriétés des locataires et montrer son intérêt pour la modélisation de la variabilité. Dans notre exemple, cette fonction est utilisée dans deux conditions d'activation différentes, la première a pour but d'activer l'exclusion de la variante *Rapide* sur le PV *Transporteur* uniquement pour les locataires situés en dehors du territoire français (`LocataireInfo(« Pays »)!= « France »`), tandis que la deuxième est responsable d'activer l'exclusion de la variante *Lent* sur le PV *Workflow* pour les locataires qui sont en phase de test de l'application (`LocataireInfo(« Statut »)!= « Test »`). L'utilisation de cette

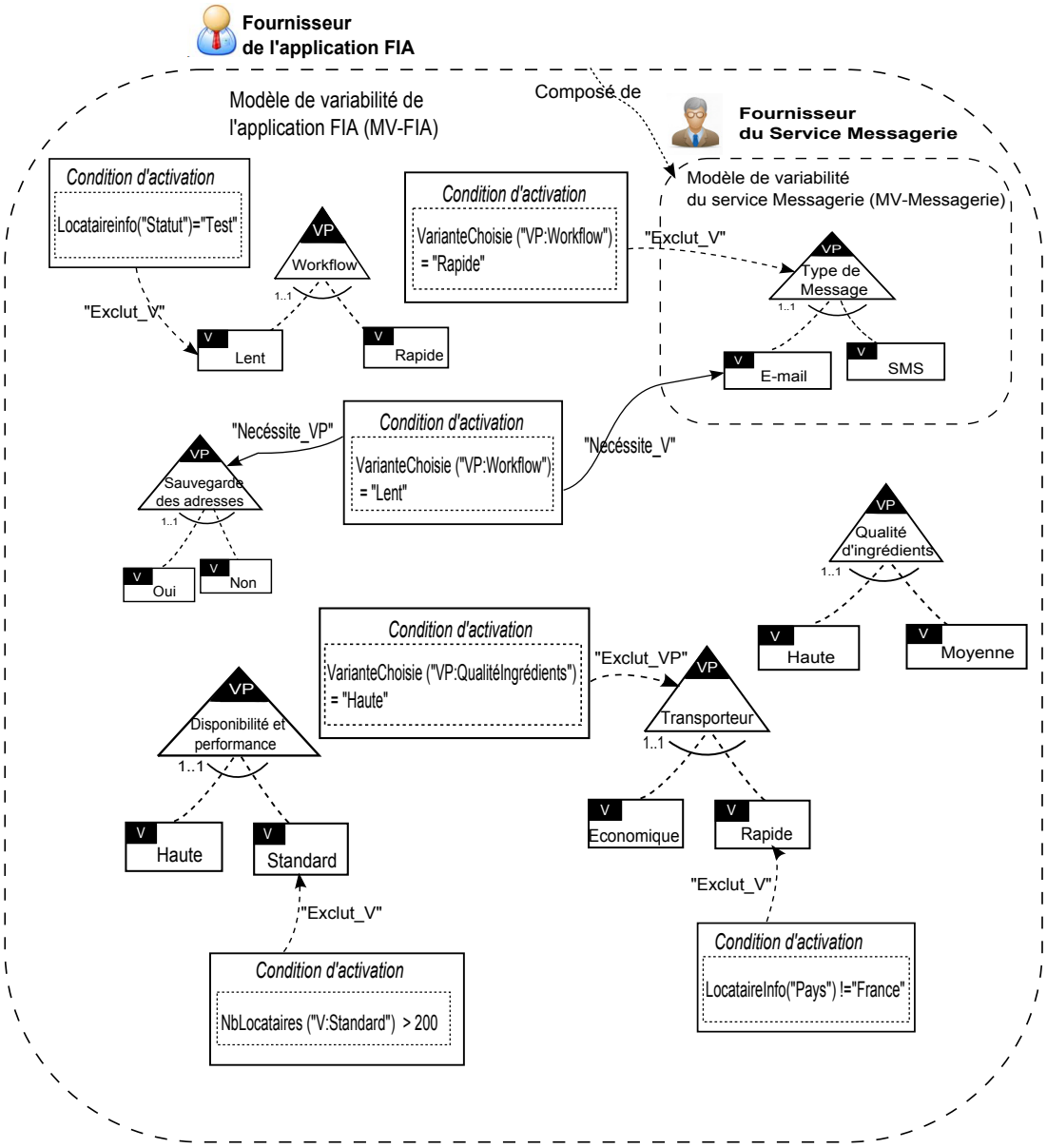


Figure 6.8 – Exemple d'instanciation du méta-modèle

fonction dans la première condition est justifiée par le fait que le transporteur rapide offert par l'application est dans l'incapacité de supporter des livraisons internationales. Ainsi, son utilité est de gérer une situation de variabilité dirigée par une exigence fonctionnelle qui dépend de la localisation géographique des locataires et de leurs usines. Pour la deuxième condition, cette fonction est utilisée pour interdire la sélection du workflow lent par les locataires durant la période de test afin d'empêcher l'interaction avec le service de messagerie qui facture son utilisation selon le nombre de messages envoyés. En effet, les locataires durant la période de test peuvent causer une augmentation du montant de cette facture alors qu'ils ne paient pas de frais d'utilisation. Ainsi, l'utilité

de cette fonction dans cette situation de variabilité est de répondre aux exigences économiques du fournisseur dans le but de réduire certaines dépenses qui ne sont pas compensables.

Formalisation : soit *CA = {...,cai,...}* un ensemble de conditions d'activation et *Co = {...,coi,...}* un ensemble des condition déclarées dans un modèle de variabilité et importées depuis les modèles composants. Une condition *co* est définie comme une expression qui est évalué à vrai ou faux en fonction de ses données d'entrée. L'évaluation d'une condition est effectuée par l'utilisation de la fonction suivante :

$$\textit{Eval (co)}\rightarrow \begin{cases} vrai & \textit{Si l'évaluation retourne vrai} \\ faux & \textit{Autrement} \end{cases}$$

D'ailleurs, la fonction *CondCA : ca→co* $\cup \perp$ est utilisée pour associer une condition ou une $\perp$ à une condition d'activation. L'association de $\perp$ au lieu d'une condition signifie qu'il s'agit d'une condition d'activation qui est toujours valide. Cela est bénéfique dans le cas où un fournisseur de SaaS souhaite poser une contrainte fixe dans son modèle de variabilité qui ne dépend pas forcément de certaines informations à évaluer. Aussi, cela peut être utilisé pour annuler temporairement l'effet d'une condition d'activation sur certaines contraintes sans être obligé de changer la structure du modèle.

Rôle : ce concept est introduit dans notre approche de modélisation pour modifier la binarité d'OVM concernant la classification des points de variation uniquement entre externes et internes. Selon notre expérience, l'échelle de classification est plus large et nécessite l'intervention de plusieurs rôles dans l'application pour intégrer leurs connaissances et compétences. Par exemple, un fournisseur de SaaS peut décider de mettre en place un réseau de revendeurs distribués à travers le monde afin de toucher plus rapidement de nouveaux locataires. Les revendeurs à engager dans cette démarche sont généralement des locataires caractérisés par une expérience dans le domaine de l'application et le besoin du marché au sein duquel ils essayent de revendre l'application. Ainsi, certains choix de variantes peuvent être effectués directement par ces revendeurs et appliqués ensuite aux locataires concernés. Le but est d'accélérer la phase de configuration de l'application pour les locataires tout en leur réduisant le nombre de points de variation qu'ils doivent traiter.

Au niveau des locataires, ce concept de rôle est aussi utile pour repartir les tâches de configuration à plusieurs utilisateurs présentant chacun une connaissance et une compétence différente. Par exemple, un rôle pourrait bien être l'expert de GUI à l'entreprise pour configurer l'ergonomie de l'application, l'expert métier pour configurer la partie fonctionnelle ou bien l'expert QoS qui configure uniquement les propriétés de qualité. Pour modéliser la variabilité interne (uniquement visible aux développeurs), les points de variation peuvent être associés au rôle fournisseur. Les choix à effectuer par le fournisseur affectent directement toute l'hiérarchie sauf dans le cas où la prise en compte de certaines contraintes contredit avec ces choix. Le binding de variabilité par le fournisseur doit toujours précéder celui à effectuer par les locataires. Les variantes choisies par les fournisseurs ne seront pas sauvegardées mais considérées comme celles par défaut sur les points de variation concernés (attribut *estParDéfaut* de la classe variante du méta-modèle).

Cependant, l'intégration de plusieurs rôles nécessite leur gestion de façon à créer une certaine hiérarchie entre eux pour les différencier et préciser la manière dans laquelle le choix d'une variante de l'application sera propagé (voir figure 6.9 pour un exemple d'hiérarchie des rôles).

Pour mieux qualifier et grouper les points de variation d'une manière plus précise, nous pouvons les marquer avec des marqueurs arbitraires (tags en anglais). Ces marqueurs pourraient, par exemple, être *Qualité* pour désigner les points de variations liés à la qualité de l'application, ou *GUI* pour

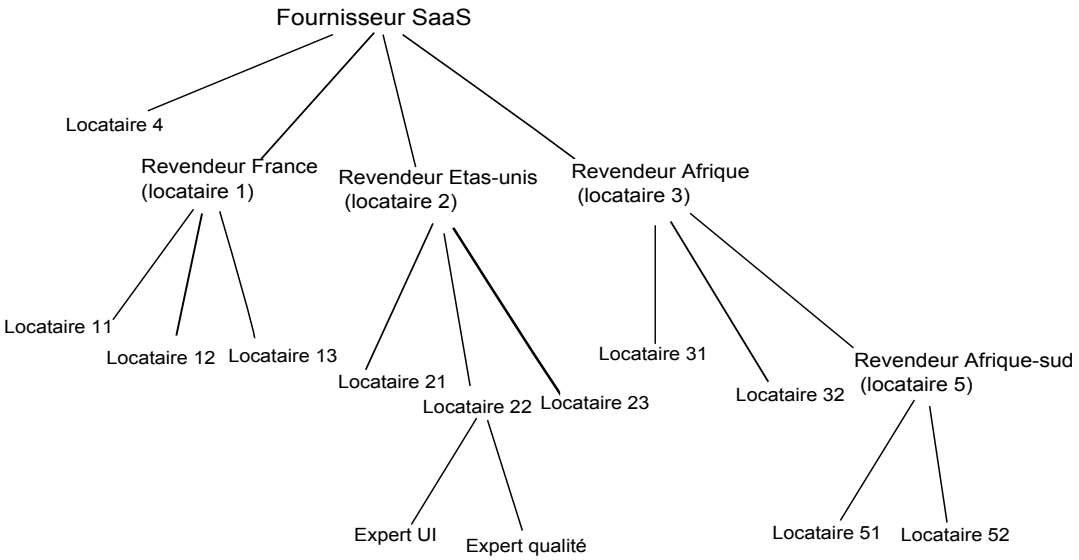


Figure 6.9 – Hiérarchie des rôles

désigner les points liés à l'interface graphique. Les marqueurs servent uniquement à titre indicatif et peuvent être exploités par notre cadre de modélisation pour générer des informations utiles aux fournisseurs comme par exemple pour identifier les locataires qui ont choisi la même ergonomie ou les mêmes paramètres de qualité.

Formalisation : soit *RL = {...,rli,...}* un ensemble de rôles définis dans un modèle de variabilité et ses modèles composés. La fonction *Roles : pv→RL* identifie l'ensemble des rôles autorisés à faire le binding sur un point de variation précis. Lors du binding de variabilité, un rôle rl ∈ RL est uniquement autorisé à faire le binding sur les points de variation qui lui ont été associés. Pour récupérer ces points, la fonction suivante est utilisée :

RolePV(r,mv)={pv $\in PV_{mv}$ *| ∀ rl ∈ Roles(pv) : rl=r }*

D'ailleurs, pour poursuivre la distinction entre les points de variations au niveau des rôles, la fonction suivante est utilisée pour récupérer les points de variation indépendants parmi l'ensemble des points associés à un rôle précis :

$RolePV^{Independants}$*(r,mv)={*$pv_{ind} \in PV^{Dependants}$*(mv) | ∀* pv_r *∈ RolePV(r,mv) :* pv_{ind}*=*pv_r *}*

L'ensemble de points de variation dépendants parmi ceux associés à un rôle est ainsi égal à la différence entre l'ensemble de points de variation associés à ce rôle et l'ensemble de points de variation indépendants :

$RolePV^{Dependants}$*(r,mv) = RolePV(r,mv) -* $RolePV^{Independants}$*(r,mv)*

Contrainte : ce concept est réutilisé à partir d'OVM. Il permet d'assurer la cohérence et l'exactitude lors du binding de la variabilité. La modification apportée par notre approche à ce niveau concerne la séparation entre les contraintes de *binding* et les contraintes de *résolution*.

Les contraintes de binding sont vérifiées par notre cadre de modélisation uniquement lors de la configuration de l'application par un locataire (binding directe) dans le but d'aboutir à une configuration correcte. La sémantique de ce type de contraintes est équivalente à celle proposée par

OVM à l'unique différence que leur validité peut être manipulée à travers la définition des conditions d'activation (voir concept *condition d'activation*).

Les contraintes de résolution sont uniquement vérifiées lors de résolution de la variabilité, où les variantes choisies durant la configuration sont à exécuter dans l'architecture de l'application. L'utilité de ce type de contraintes est d'*annuler* ou de *retarder* l'exécution de certaines variantes en fonction des besoins de la variabilité. Par exemple, la variabilité dans la composition de services exige parfois d'annuler l'invocation de certains services ou même de changer l'ordre de leur exécution en retardant leur invocation jusqu'au l'occurrence d'une condition précise.

Formalisation : soit *C* = $Contraintes^{Exclut} \cup Contraintes^{Ncessite} \cup Contraintes^{Annule} \cup Contraintes^{Retarde}$ l'ensemble de contraintes appartenant à un modèle de variabilité et ses modèles composants. L'établissement de telles contraintes dans un modèle de variabilité exige avant tout la précision du type des contraintes à établir dans le modèle et ensuite l'élément de variabilité concerné (un point de variation ou une variante).

Par exemple, la fonction $ContraintesV^{Exclut}$ *: v→C* est utilisée pour associer un ensemble de contraintes d'exclusion à une variante. De même, la fonction $ContraintesPV^{Exclut}$ *: pv→C* associe un ensemble de contraintes d'exclusion à un point de variation. Les autres types de contraintes (nécessite, annule et retarde) sont associées aux éléments de la variabilité de la même façon.

Pour conditionner l'activation d'une contrainte, la fonction *CondAct : c→CA* est utilisée. Celle-ci associe un ensemble de conditions d'activation à une contrainte dans le but de valider ou non son existence dans le modèle lors de la configuration.

Variante : ce concept est réutilisé à partir d'OVM pour modéliser les options possibles sur un point de variation. À la différence d'OVM et d'autres modèles de variabilité, tels que FORME [**?**] ou FODA qui autorisent la sélection de plusieurs variantes, notre approche de modélisation à ce niveau limite le nombre de variantes qui peuvent être choisies sur un point de variation à une seule variante. Pour rendre cette distinction plus explicite, nous autorisons l'utilisation du terme *alternatives* pour exprimer les variantes, puisqu'elles sont mutuellement exclusives par point de variation. La raison de sélection d'une variante unique a pour but d'empêcher le conflit d'exécution lors de la résolution de la variabilité. En revanche, les modèles de variabilité offerts par les approches de LDP sont exploités à un niveau plus abstrait que le niveau de code sur lequel notre approche de modélisation fonctionne, plusieurs variantes peuvent être sélectionnées.

Formalisation : soit *V = {...,vi,...}* un ensemble de variantes appartenant à un modèle de variabilité et ses modèles composants. Pour associer un ensemble de variantes à un point de variation, la fonction suivante est utilisée : *Variante : pv→V*. Chaque variante dans cet ensemble peut être permise ou non. Cela est déterminé à travers l'évaluation de toutes les conditions des conditions d'activation associées aux contraintes d'exclusion de cette variante. Ainsi, pour récupérer uniquement les variantes permises sur un point de variation, la fonction suivante est utilisée :

$V^{Permises}$*(pv)= Variante(pv) - {v* $\in Variante(pv)$ *| ∀ c* $\in$ $ContraintesV^{Exclut}$*(v) | ∀ ca* $\in$ *CondAct(c) : Eval (CondCA (ca)) = vrai}*. Cette fonction retourne l'ensemble des variantes à travers le calcul de la différence entre l'ensemble des variantes appartenant à ce point et celui des variantes exclues.

D'ailleurs, la sémantique opérationnelle du méta-modèle exige la sélection automatique de chaque variante dont les contraintes de nécessité sont valides à un moment donné (c.à.d. les conditions de conditions d'activation associées à ses contraintes sont toutes évaluées à vrai), à condition qu'elle soit l'unique nécessaire parmi l'ensemble des variantes permises. Pour déterminer avant tout la nécessité d'une variante, la fonction suivante est utilisée :

$$\textit{VarianteNécessaire (v)} \rightarrow \begin{cases} vrai & \forall\, c \in ContraintesV^{Ncessite}(v) \mid \forall\, ca \in \textit{CondAct(c)} : \\ & \textit{Eval (CondCA (ca)) = vrai} \\ faux & \textit{Autrement} \end{cases}$$

Le même raisonnement est appliqué pour déterminer l'exclusion d'une variante, ainsi que pour la nécessité et l'exclusion d'un point de variation, sauf que leur unicité n'a pas d'interprétation spécifique :

$$\textit{VarianteExclue (v)} \rightarrow \begin{cases} vrai & \forall\, c \in ContraintesV^{Exclut}(v) \mid \forall\, ca \in \textit{CondAct(c)} : \\ & \textit{Eval (CondCA (ca)) = vrai} \\ faux & \textit{Autrement} \end{cases}$$

$$\textit{PVNécessaire (pv)} \rightarrow \begin{cases} vrai & \forall\, c \in ContraintesPV^{Ncessite}(pv) \mid \forall\, ca \in \textit{CondAct(c)} : \\ & \textit{Eval (CondCA (ca)) = vrai} \\ faux & \textit{Autrement} \end{cases}$$

$$\textit{PVExclu (pv)} \rightarrow \begin{cases} vrai & \forall\, c \in ContraintesPV^{Exclut}(pv) \mid \forall\, ca \in \textit{CondAct(c)} : \\ & \textit{Eval (CondCA (ca)) = vrai} \\ faux & \textit{Autrement} \end{cases}$$

Pour déterminer qu'une variante est la seule nécessaire parmi un ensemble de variantes, la fonction suivante est utilisée :

$$\textit{VarianteSeuleNécessaire (v,V)} \rightarrow \begin{cases} vrai & \forall\, \textit{vi} \in \textit{V-v : VarianteNécessaire(vi) = faux} \\ faux & \textit{Autrement} \end{cases}$$

Valeur : ce concept est introduit dans notre approche pour indiquer la valeur d'une variante sur un artéfact précis parmi l'ensemble des artéfacts capables de la réaliser. Ce concept permet de saisir les méta-données nécessaires à l'exécution dynamique de la variante dans le but de retourner ses résultats à l'application. La même variante peut avoir différentes valeurs pour différents artéfacts. Chacune de ce dernières doit avoir un type spécifique afin de pouvoir déterminer l'action de résolution nécessaire à effectuer. La spécification d'un type à chaque valeur permettra à notre cadre de modélisation de déterminer l'interface appropriée dans lequel cette valeur sera saisie. Par exemple, si nous créons une instance de valeur pour une variante particulière avec le type *Web-Service*, une interface graphique est automatiquement ouverte pour fournir la description du service (ex : l'adresse d'invocation, l'opération, les paramètres, etc.). De même, si une instance de valeur est créée avec le type *fichier*, l'interface graphique ouverte sera un simple *Input* du fichier.

Toutefois, chaque valeur d'une variante peut être soit prédéfinie par le modèle, soit libre à saisir directement par les locataires lors de la configuration (qu'ils soient des revendeurs ou des locataires simples). L'approche d'OVM ne supporte pas cette différentiation et permet uniquement la modélisation des valeurs prédéfinies en raison de son orientation différente basée sur la composition statique des artéfacts pour générer l'architecture d'un produit. Mais la configuration d'une application en cours d'exécution, tel que c'est le cas dans la mutualisation de SaaS, signifie que l'application est déjà mise en place avant son utilisation et traite dynamiquement des informations pour son auto-adaptation. De ce fait, les locataires peuvent participer à cette adaptation en fournissant les informations nécessaires à travers les valeurs libres, et ce à des niveaux d'abstraction qui correspondent à leur compréhension. Les locataires interviennent généralement sur la variabilité de l'ergonomie en indiquant le titre de l'application qu'il souhaite, son logo et sa couleur d'arrière plan, et parfois à un niveau plus technique tel que la spécification des règles métiers du

processus. Notre cadre de modélisation facilite aux locataires la saisie des valeurs libres à travers des interfaces graphiques simplifiées.

Formalisation : soit *VAL* = $VAL^{Predefinies} \cup VAL^{Libres}$ un ensemble de valeurs appartenant à un modèle de variabilité et ses modèles composants. Ces valeurs sont divisées entre celles prédéfinies par le modèle et celles libres à saisir directement par les locataires. Ainsi, l'identification de la spécialisation exacte d'une valeur est nécessaire pour déterminer comment la traiter. Pour pouvoir faire cette distinction, la fonction suivante est utilisée :

SpécialisationValeur (val)$\rightarrow \begin{cases} Libre \\ Predefinie \end{cases}$

D'ailleurs, pour récupérer la valeur (libre ou prédéfinie) d'une variante réalisée par un artéfact précis, la fonction suivante est utilisée : *Valeur(v,art)*.

Artéfact de développement : ce concept est réutilisé à partir d'OVM. Il est associé aux points de variation et aux variantes pour établir des liens de traçabilité entre la variabilité séparément modélisée et les artéfacts de développement affectés par cette variabilité. L'extension apportée par notre approche de modélisation à ce niveau concerne la modélisation de la manière selon laquelle ces artéfacts sont reliés, à travers l'association *en relation avec*. Cette association sert plus précisément à modéliser la relation entre les artéfacts qui représentent un point de variation et ceux qui réalisent chacune de ses variantes. Cela est utile déjà pour représenter à un niveau abstrait la relation entre les artéfacts affectés par la variabilité, et ensuite pour savoir quel artéfact alternatif sélectionner lors de la résolution d'un point de variation (parmi l'ensemble des artéfacts qui peuvent réaliser la variante choisie sur ce point).

Formalisation : soit *A = {...,ai,...}* un ensemble d'artéfacts variables appartenant à un modèle de variabilité et ses modèles composants. La fonction *Représente : pv→A* associe un point de variation à l'ensemble des artéfacts qui le représentent, tandis que la fonction *Réalise : v→A* associe à une variante l'ensemble des artéfacts qui la réalisent. La fonction *Relie : a→A* crée une relation entre un artéfact et un ensemble d'autres artéfacts. Pour identifier l'artéfact à sélectionner parmi ceux qui réalisent la variante choisie sur le point de variation à résoudre, la fonction suivante est utilisée :

ArtéfactV(pv,$artefact_{pv}$*) = {*$a_v \in$ *Réalise(VarianteChoisie(pv)) |* $\forall$ *a* $\in$ *Relie(*$artefact_{pv}$*) : artefact =* $artefact_v$*}*

Cette fonction prend le point de variation à résoudre comme entrée et l'artéfact variable (parmi ceux qui représentent ce point au niveau du code) pour lequel cette résolution est destinée à adapter.

Dans la section suivante nous détaillerons notre approche de gestion de la variabilité sous forme d'un service et montrerons l'intérêt du méta-modèle présenté pour le fonctionnement de cette approche.

6.4 Variabilité sous forme d'un service

Avoir une approche de modélisation de la variabilité qui conforme avec la spécificité des applications SaaS mutualisées représente uniquement une partie du défi. L'autre partie consiste à résoudre cette variabilité lors de l'exécution de l'application en fonction du locataire connecté et les variantes qu'il avait déjà choisies. Ces deux parties du défi sont complémentaires dans le sens où nous ne pouvons pas résoudre la variabilité sans qu'elle soit modélisée auparavant.

Toutefois, et même si les approches de résolution de la variabilité (de l'état de l'art, confère section 4.3) adoptées sur l'ensemble des couches de l'AOS semblent être efficaces à cette résolution, leur

mise en œuvre concrète est une tâche difficile qui surcharge les développeurs et augmente les coûts opérationnels de l'application [**?**]. Cela risque de consommer un temps de développement additionnel qui est susceptible de retarder la mise de l'application sur le marché et d'imposer un détournement important du cœur des compétences chez le fournisseur. Un tel détournement peut être parfois une source de distraction qui dépasse les capacités du fournisseur à maintenir les délais et qui l'empêche de se focaliser uniquement sur son savoir faire et ses connaissances acquises en termes de technologie de développement et de compréhension du domaine métier de l'application.

Pour éviter une telle complexité et pour être toujours capable de gérer efficacement la variabilité, nous avons décidé d'externaliser cette gestion à un fournisseur plus expérimenté (dans le même esprit des services du cloud). Le système de gestion de *la variabilité sous forme d'un service (VaaS : Variability as a Service)* que nous allons présenter implique la nécessité d'un nouveau type de fournisseur de service : le fournisseur de VaaS. Ce dernier propose à ses clients (c.à.d les fournisseurs de SaaS) de modéliser et de résoudre la variabilité de leurs applications sur sa propre infrastructure. Nous allons expliquer dans la section suivante l'architecture de ce système ainsi que le processus à travers lequel la gestion de la variabilité peut être externalisée.

6.4.1 L'architecture de VaaS et le processus d'externalisation de la gestion de la variabilité

Le système VaaS que nous proposons fournit un ensemble de composants logiciels génériques dédiés à la gestion de la variabilité des applications mutualisées, et ce indépendamment de leurs technologies de développement utilisées. Ces composants permettent aux fournisseurs de SaaS (ou même les fournisseurs des services Web classiques) de gérer la variabilité des applications sur l'infrastructure du système proposé. Cela revient à modéliser dans un premier temps cette variabilité suivant les concepts de modélisation présentés, et ensuite de la résoudre à travers l'envoi de demandes de résolution en temps réel. L'architecture du système VaaS est présentée dans la figure 6.10. Comme le montre cette figure, le système VaaS défini un ensemble d'étapes à suivre par les fournisseurs de SaaS et par les locataires dans le but d'externaliser la gestion de la variabilité. Ces étapes sont divisées en deux catégories principales : la *spécification* et la *résolution*.

6.4.1.1 Étapes de spécification

Ces étapes consistent principalement à modéliser correctement la variabilité de l'application et à la configurer complètement par les locataires afin de préparer le terrain aux étapes de résolution (voir figure 6.10 A). Pour modéliser la variabilité, le fournisseur de SaaS doit tout d'abord instancier (1) le méta-modèle de variabilité (a) que nous avons proposé. Cela lui impose bien évidemment une compréhension des concepts du méta-modèle et de la manière dont laquelle ces concepts peuvent être manipulés. Le modèle de variabilité résultant de cette instanciation sera ensuite sauvegardé dans un répertoire spécifique (b) contenant tous les modèles de variabilité appartenant à toutes les applications clientes du système. Pour différencier ces modèles, chacun sera associé au nom de l'application qu'il représente.

Une fois la variabilité modélisée, le binding de celle-ci peut commencer (2). Celui-ci est effectué par le biais d'un outil de binding (c) mis en œuvre dans notre système. Cet outil permet d'un côté d'aider les locataires à naviguer dans le modèle de variabilité de l'application, et d'un autre côté de contrôler l'accès aux éléments de ce modèle en évaluant les conditions d'activation associées aux contraintes (confère section 6.3.2.2). Une fois le binding effectué, les variantes choisies par chaque locataire seront sauvegardés (3) dans un fichier de configuration (d) qui porte le nom de ce locataire et stocke une représentation

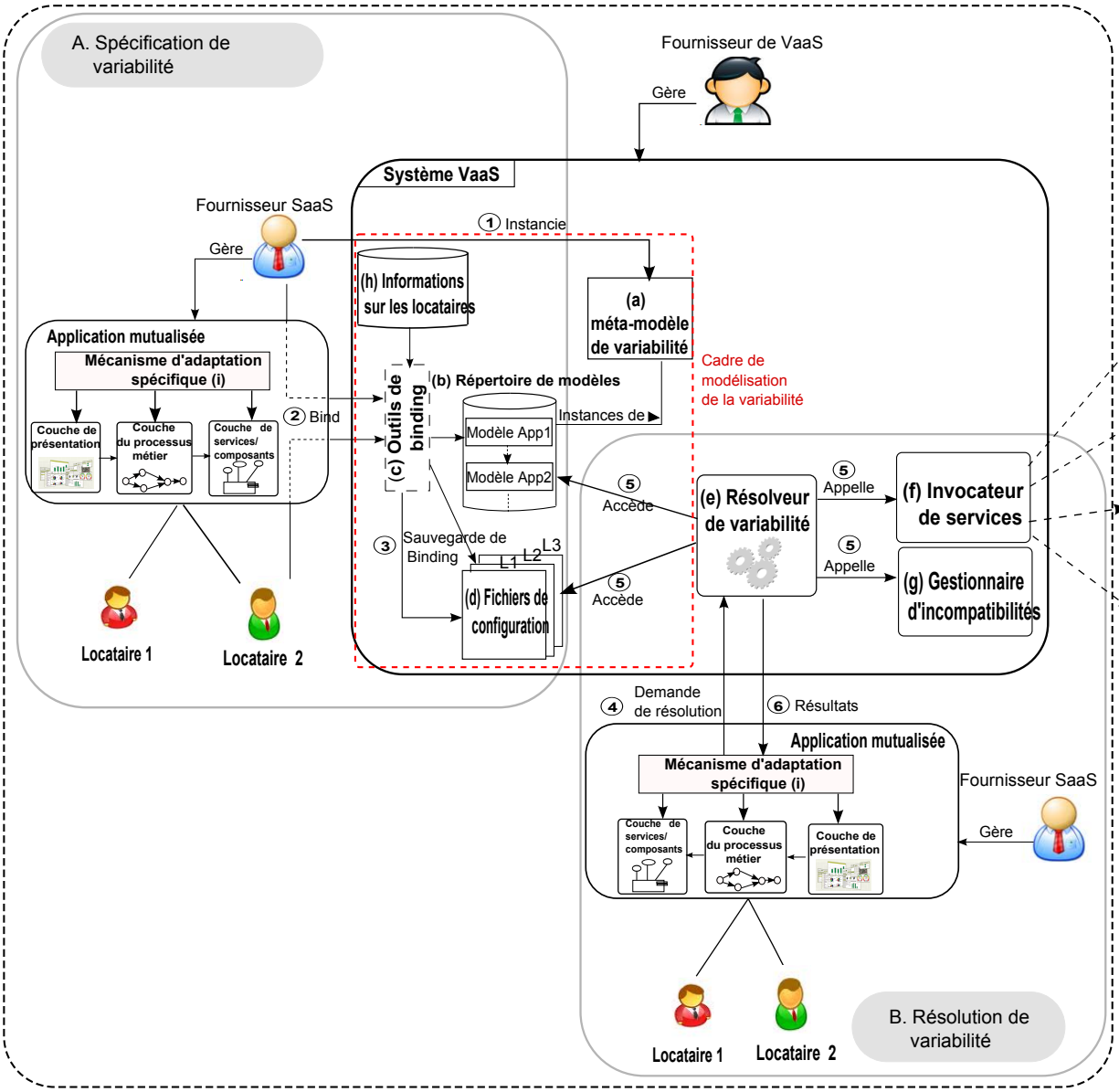


Figure 6.10 – Architecture de VaaS

sérialisée de cette configuration sous forme d'un fichier XML. À ce moment, ce locataire peut commencer à utiliser l'application et s'attendre à une fonctionnalité et une qualité de service équivalentes à la configuration effectuée.

6.4.1.2 Étapes de résolution

Ces étapes consistent à adapter l'application à l'exécution une fois son modèle de variabilité définie et sa configuration effectuée (voir figure 6.10 B). Cette adaptation est effectuée à travers des demandes

de résolution à envoyer en temps réel par l'application. Chaque demande concerne un artéfact de développement précis et vise le point de variation du modèle de variabilité que cet artéfact le représente au niveau du code (voir figure 6.11 pour exemple de résolution). Nous supposons que ces demandes sont déjà codées par le fournisseur de SaaS à travers un mécanisme d'adaptation spécifique (i) à la technologie de développement utilisée pour mettre en œvre son application. Ce mécanisme a pour rôle d'intercepter les endroits variables de l'application à l'exécution, d'interagir avec le système VaaS et de traiter ensuite les résultats de résolution obtenus. L'existence de tels mécanismes est essentielle pour pouvoir traiter la variabilité de l'application comme une préoccupation séparée, sans avoir à interagir avec le système VaaS directement à partir du code métier (une approche intrusive et complexe à maintenir). Pour accélérer le processus d'externalisation, nous supportons actuellement un ensemble de mécanismes « ready-to-use » compatibles avec des différentes technologies de développement. Nous les fournissons sous forme des composants logiciels à installer dans l'environnement de développement de chaque application cliente. Dans le cas où celle-ci utilise une technologie de développement non-supportée, le mécanisme nécessaire sera développé sous la propre responsabilité de son fournisseur. Bien évidemment, cela sera accompagné par un service de conseil et de bonnes pratiques. Ce mécanisme est responsable d'intercepter les endroits variables à résoudre durant l'utilisation de l'application par un locataire (après l'avoir configurée). Pour chaque endroit variable intercepté, une demande de résolution sera envoyée au système VaaS (4) visant le point de variation intercepté à condition que ce point soit représenté au niveau du code de l'application (ex : instruction d'invocation d'un service dans le code du processus métier de l'application). Lorsqu'une demande est reçue, elle sera traitée par un résolveur de variabilité (d) qui accède au modèle de variabilité de l'application et aux fichiers de configuration du locataire pour identifier l'action de résolution appropriée à effectuer (5) (ex : invocation du service qui correspond à la variante choisie sur le PV à résoudre). Cette action génère des résultats de résolution qui seront retournés à l'application pour pouvoir continuer son exécution (6). Ces résultats doivent toujours avoir une structure de données compatible avec celle requise par l'application pour ne pas causer des erreurs d'exécution (ex : incompatibilité de services alternatifs). Le gestionnaire d'incompatibilités (g) au niveau du système VaaS est conçu pour cette raison. Il est responsable de l'ajustement dynamique de données générées à partir d'une résolution pour avoir des structures de données compatibles . Cela est le plus souvent utilisé dans le cas où il existe des services alternatifs qui sont sémantiquement compatibles mais chacun retournant ses données dans une structure de données différente (voir figure 6.12). Les méta-données définissant les structures de données requises par l'application suite à la résolution de chaque point de variation doivent être explicitement définies dans le modèle de variabilité, et ce à travers l'attribut *structure* de la classe d'association *Résolution* (voir figure 6.6) entre le point de variation à résoudre et l'artéfact de développement concerné par cette résolution.

6.4.2 Les composants clés

Après avoir présenté le fonctionnement global du système VaaS, cette sous-section se concentre sur l'explication du fonctionnement de ses composants. Chacun joue un rôle précis dans l'architecture de ce système et contribue à sa faisabilité. Le méta-modèle de variabilité présenté dans la section précédente représente le composant principale de cette architecture. Dans la suite, nous allons nous concentrer sur les autres composants. Le répertoire de modèles de variabilité instances et les fichiers de configuration servent uniquement à stocker des informations et donc leur fonctionnement interne est relativement facile à cerner par rapport aux autres composants qui effectuent des tâches de traitement complexe et doivent être bien conçus pour la réussite de VaaS. Voici ces composants :

Outil de binding : le rôle de cet outil dans notre système VaaS est d'aider les locataires à réaliser le binding du modèle de variabilité une fois que celui-ci est créé et enregistré dans le répertoire des modèles. L'objectif d'utiliser un tel outil est de garantir aux locataires une configuration correcte et complète. Une configuration correcte signifie que seules les variantes parmi celles permises ont été choisies et leurs valeurs libres ont été bien saisies, alors que la complétude signifie que le binding sur tous les points de variation a été bien effectué. Ainsi, l'application sera bien adaptée à chaque locataire locataire seulement si la configuration est correcte et complète. Pour garantir ces deux propriétés, cet outil doit dans un premier temps veiller à ce que le binding du modèle de variabilité suive la sémantique opérationnelle du méta-modèle, surtout au niveau du traitement de la dépendance entre les points de variation et la relation de composition entre les modèles de variabilité (ex : présenter les points de variation dans l'ordre exacte en distinguant les points indépendants et les points dépendants, confère section 6.3.2.2). Cet outil doit ainsi implémenter un algorithme de vérification de configuration parmi deux types d'algorithmes généralement utilisées. Le premier type consiste à vérifier la configuration de l'application après qu'elle soit faite tandis que le deuxième type permet de vérifier le binding sur chaque point de variation.

Le premier type d'algorithmes comporte deux inconvénients potentiels : la complexité possible d'un modèle de variabilité comprenant beaucoup des dépendances ainsi qu'un grand nombre de points de variation complique la vérification de la configuration [SZG+08, **?**]. De plus, il n'est pas très convivial de dire à un locataire, après une relativement longue étape de configuration, qu'une erreur a été produite sur un point de variation et qu'une refonte complète de la configuration est nécessaire en raison des dépendances. Ainsi, cette possibilité est mieux adaptée pour les modèles de variabilité plus simples avec un nombre réduit de dépendances.

Par conséquent, l'approche adoptée par le système VaaS à travers cet outil est de naviguer le modèle de variabilité d'une façon incrémentale en vérifiant le binding effectué sur chaque point de variation. Cela oblige le locataire à effectuer une configuration complète et correcte même s'il mettra plus du temps à le faire, mais nous évitons forcement les inconvénients potentiels de la première possibilité.

l'algorithme de navigation que l'outil de binding utilise pour guider les locataires et aboutir à une configuration correcte et complète est divisé en deux parties

- **Parite 1 : parcourir les points de variation indépendants :**
  dans cette partie, l'algorithme récupère l'ensemble de points de variation indépendants associés à un rôle précis dans un modèle de variabilité (et ses modèles composants) en utilisant la fonction suivante : $RolePV^{Independants}$*(r,mv)*. Cette fonction prend le modèle de variabilité à naviguer *mv* et le rôle *r* actuel comme entré (revendeur, locataire simple, expert métier, etc.). Ensuite, pour chaque point de variation *pvi* non-exclu (*!PVExclu(pvi)*) dans cet ensemble, l'algorithme vérifie parmi l'ensemble de ses variantes l'existence d'une qui est la seule nécessaire (finction : *VarianteSeuleNécessaire(vi,* $V^{Permises}$*(pvi)*). Dans ce cas, cette variante *vi* sera sélectionnée par défaut. Dans le cas contraire, l'algorithme expose au rôle toutes les variantes permises (celles non-exclues par des contraintes d'exclusion actives, fonction : $V^{Permises}$*(pvi)*), pour sélectionner celle souhaitée. Une fois la variante *vi* est sélectionnée (par défaut ou par le rôle) l'algorithme vérifie la définition d'une valeur libre (fonction : *SpécialisationValeur(Valeur(vi,artefact))* pour cette variante afin de la saisir directement par le rôle actuel. L'algorithme passe ensuite au PV suivant du modèle de variabilité à naviguer et répète la même procédure
- **Parite 2 : parcourir les points de variation dépendants :**

dans cette partie, l'algorithme récupère l'ensemble de points de variation dépendants en utilisation la fonction suivante : $RolePV^{Dependants}$*(r,mv)*. L'algorithme exécute la même procédure décrit dans la partie précédente sur chaque PV de cet ensemble.

Résolveur de variabilité : ce composant résout la variabilité des applications clientes en recevant des demandes de résolution en temps réel. Les demandes reçues sont structurées d'une manière prédéfinie (voir figure 6.11) et doivent fournir un ensemble d'informations nécessaires à leur traitement par notre système :

- *L'identifiant de l'application (AppID) :* cette information sert à identifier l'application à l'origine de la demande de résolution afin de charger son modèle de variabilité et vérifier l'existence des contraintes de résolution à considérer.
- *L'identifiant du locataire (LocataireID) :* cette information est l'identifiant unique du locataire qui a provoqué l'envoie de la demande de résolution par l'application suite à son utilisation. Cette information, stockée sur le système VaaS, sert à identifier le locataire et à charger son fichier de configuration.
- *Le point de variation à résoudre (PVR) :* cette information sert à identifier le point de variation concerné par la demande de résolution effectuée, afin de déterminer la variante à exécuter à travers sa valeur et de retourner les résultats de résolution à l'application.
- *L'artéfact de développement concerné (ART) :* cette information identifie l'aréfact variable de l'application dont cette résolution est destinée à adapter. Puisqu'il peut exister plusieurs artéfacts qui représentent le point de variation à résoudre, l'intérêt de cette information est de préciser celui concerné.

  La figure 6.11 montre l'envoie des demandes de résolution depuis l'application FIA (vers le système VaaS) sur deux points de variation de son modèle de variabilité *PV Qualité d'ingrédients* et *PV Transporteur*. L'action de résolution effectué par ce composant dans cet exemple précis consiste à invoquer dynamiquement le service approprié qui correspond à la variante déjà choisie par le locataire (lors du binding de la variabilité) sur le point de variation indiqué dans chaque demande de résolution.

Gestionnaire d'incompatibilités : ce composant implémente des transformateurs qui compense la non-concordance de structure et de formatage de données entre la valeur retournée par la résolution d'un point de variation et celle requise par l'application. Ainsi, seule l'équivalence sémantique est nécessaire entre les variantes pour être considérées comme des solutions possibles sur un point de variation. La figure 6.12 illustre un exemple de gestion d'incompatibilité entre la structure de données récupérée à partir des résultats de résolution du *PV Qualité d'ingrédients* (voir figure 6.11) et la structure de données requise par l'application. Cette gestion est réalisée à travers un mécanisme de transformation qui accomplit la compensation de messages par des substitutions basées sur l'utilisation des règles de transformation codées suivant le langage XSLT 2.0 (Extensible Stylesheet Transformation) [**?**]. Ces règles sont définies par le fournisseur de SaaS sous forme de fichiers à exécuter lors de la détection d'une incompatibilité. Chaque fichier concerne une variante du modèle dont la structure de données retournée à partir de son exécution est incompatible avec celle requise par l'application sur le point de variation concerné à résoudre. Le transformateur XSLT (composant logiciel chargé de la transformation) crée une structure logique arborescente à partir de la structure de données retournée par l'exécution de la variante, et lui fait subir des transformations selon les règles prédéfinies pour produire une structure adaptée.

D'ailleurs, il est possible d'affecter à travers ce gestionnaire une liste de règles de transformation qui permettent au transformateur d'appliquer des règles différentes aux diverses parties de données

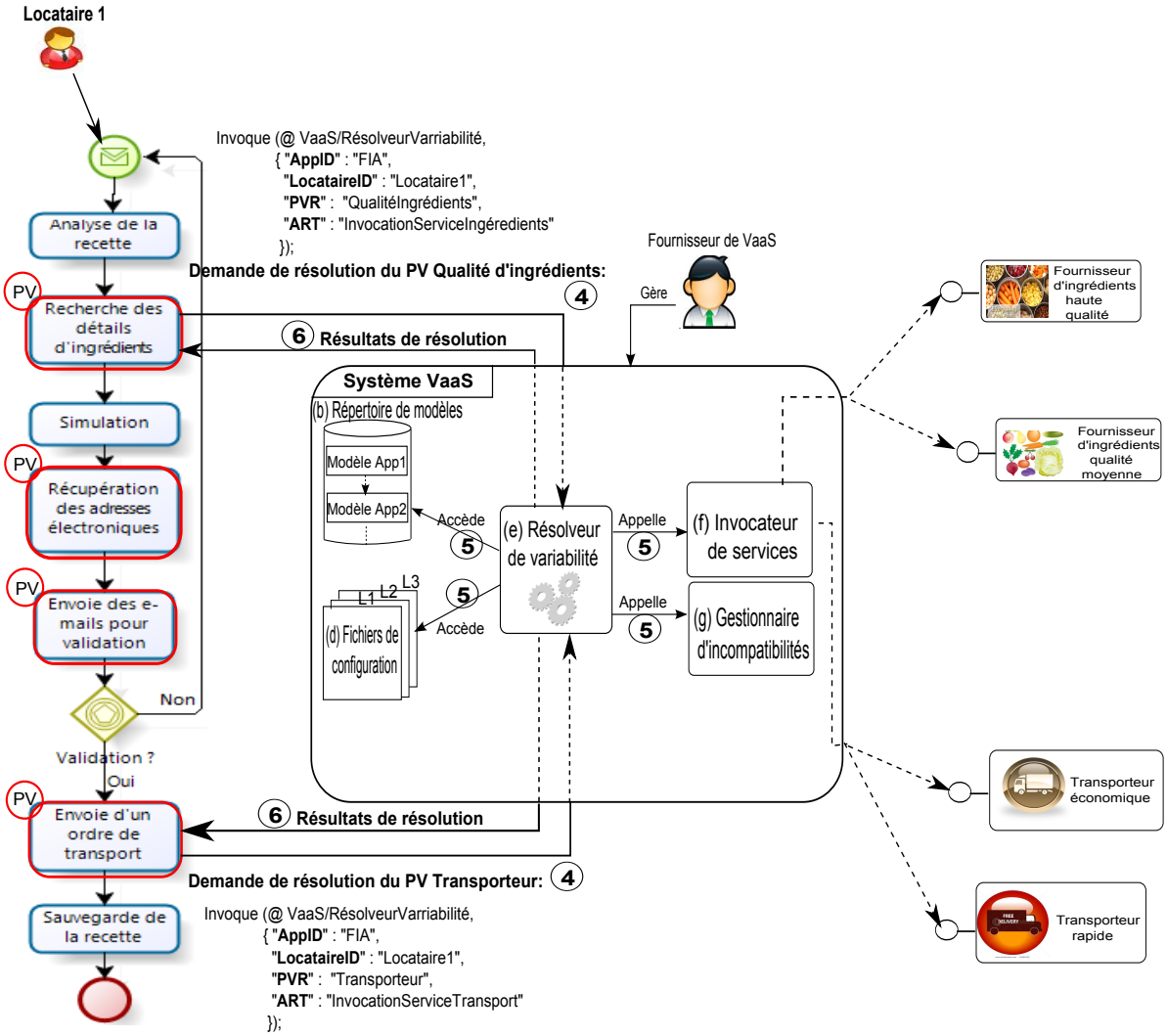


Figure 6.11 – Résolution de la variabilité de l'application FIA à travers l'interaction avec le système VaaS en temps réel

ou mettre en œuvre des transformations enchaînées. Dans les rares cas où les transformations ne sont pas applicables (par exemple, si les exigences ne peuvent pas être respectées par le langage XSLT), le système VaaS permet, à travers ce composant, de configurer un moteur de transformation externe tel que Apache Synapse [**?**].

6.5 Conclusion

Dans ce chapitre, nous avons présenté l'évolution qu'a connu notre travail de recherche concernant la gestion de la variabilité dans les applications SaaS mutualisées. Nous avons commencé par la présentation d'un exemple d'application industrielle (FIA : Food Industry Application) que nous avons développé et géré sa variabilité au travers de la modélisation et de résolution. La modélisation s'appuie sur OVM

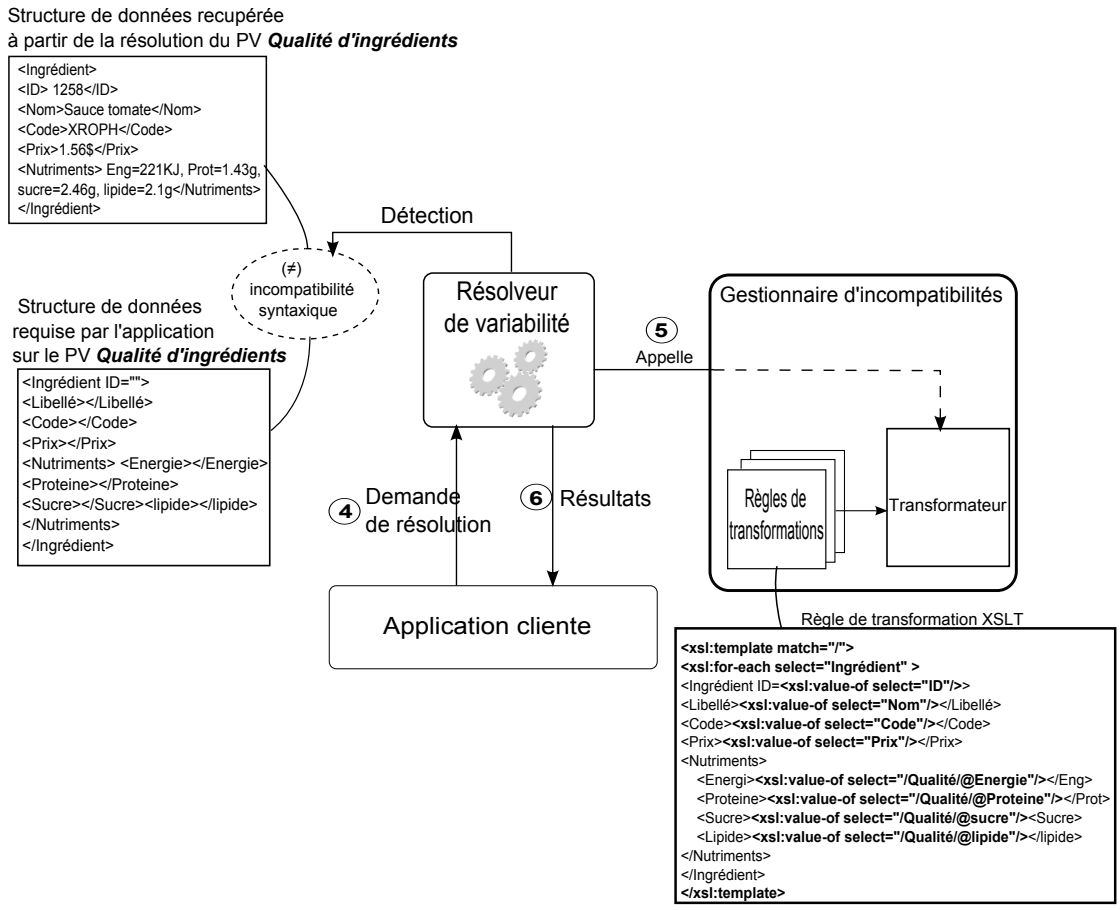


Figure 6.12 – Exemple de gestion d'incompatibilités lors de résolution de la variabilité de l'application FIA

(Orthogonal Variability Model), une technique de modélisation de la variabilité bien établie dans le domaine de lignes de produits logiciels, et la résolution se fait au travers les couches de l'architecture AOS supportant l'application tout en exploitant les capacités de technologies de développement utilisées (la plateforme .NET et ses outils de développement intégrés). Nous avons ensuite présenté les manques identifiés dans l'approche de modélisation d'OVM face à la spécificité des applications SaaS mutualisées et surtout leur structure organisationnelle qui sépare entre la construction du logiciel effectuée par son fournisseur, et sa configuration effectuée directement par les locataires à n'importe quel moment. Nous avons pour cela proposé un nouveaux méta-modèle de variabilité qui étend OVM avec des concepts de modélisation supplémentaires permettant d'un côté de contrôler l'accès aux modèles de variabilité (instances du méta-modèle) après leur exposition directe aux locataires et d'augmenter l'autonomie de ces derniers lors de la configuration de l'application (binding de la variabilité) et la création unilatérale de leur espace d'utilisation d'un autre côté. Une fois que nous avons stabilisé et testé les concepts de ce méta-modèle dans plusieurs applications, la prochaine étape consistera à externaliser la gestion de la variabilité. Nous avons pour cela introduit la notion de *(VaaS : Variability as a Service)* comme un nouveau membre de la famille des services du cloud. La justification de cette contribution réside essentiellement dans notre orientation plus général de gérer la mutualisation de SaaS par externalisation,

ce qui nécessite tout d'abord d'externaliser la gestion de la variabilité. Nous avons ainsi présenté l'architecture du système VaaS et le processus d'externalisation de gestion de la variabilité qui'il propose ainsi que le rôle du méta-modèle dans cette architecture. Dans le chapitre suivant, nous allons prolonger cette politique de gestion par externalisation afin de l'appliquer au niveau de la gestion de l'isolation des données de locataires et montrer ensuite notre approche générale de gestion de la mutualisation de SaaS par externalisation.

Vers l'Externalisation de la Gestion de la Mutualisation de SaaS : Application aux Données

7.1 Introduction

Dans le chapitre précédent, nous avons présenté le système VaaS en détails. Ce dernier présente comme but d'externaliser la gestion de la variabilité des besoins de locataires dans une application SaaS mutualisée à un fournisseur tiers. Avec ce système, un défi extrêmement important de la mutualisation de SaaS est résolu avec un nouveau modèle de gestion qui vise à réduire les investissements dans la gestion de la variabilité et facilite la modélisation et la composition de services variables. Toutefois, la gestion de la variabilité reste à elle seule un défi à relever, étant donné qu'une solution complète de gestion de la mutualisation devrait également prendre en considération l'isolation des données de locataires. Celui-ci nécessite l'exploitation des capacités d'une base de données à gérer les exigences de la mutualisation sur les trois aspects de gestion identifiés : *la séparation de données entre les locataires*, *la sécurisation d'accès à leurs données* et la *personnalisation du schéma partagé* (voir section 5.2). Pour garder l'uniformité de gestion par externalisation initialement adoptée au niveau du système VaaS, nous investiguons la faisabilité d'externaliser l'infrastructure informatique nécessaire à établir un système d'isolation de données. Ceci nous permettra de proposer la gestion complète de la mutualisation de SaaS sous forme d'un service qui pourra être utilisé à la demande, motivant par conséquent l'adoption de ce principe d'architecture logicielle et réduisant au maximum les risques qui peuvent en découler (complexité, coût de développement supplémentaire, retard de mise de l'application sur le marché, etc.) .

Dans ce chapitre, nous allons étendre la politique de gestion par externalisation initialement adoptée pour gérer la variabilité afin de l'appliquer au niveau des données en proposant un système d'isolation de données sous forme d'un service *(DIaaS : Data Isolation as a Service)*. La section 7.2 présente ce système et son architecture ainsi que le processus d'externalisation proposé et les composants clés qui contribuent à cette externalisation. La section 7.3 notre plateforme dédiée à la gestion de la mutualisation de SaaS qui regroupe les systèmes VaaS et DIaaS, ainsi que d'autres services liés à l'administration et à la sécurité des applications mutualisées, afin d'offrir une solution globale de gestion de la mutualisation de SaaS. Par la suite, la section 7.4 classifie et évalue des plateformes actuellement présentes sur le marché qui proposent des services de gestion similaires et montre les différences avec notre approche de gestion de points de vue fonctionnel et économique. Enfin, la section 7.5 conclut le chapitre.

7.2 Isolation de données sous forme d'un service

Pour rendre une base de données mutualisée, nous avons analysé dans la section 5.3 un ensemble d'approches existantes qui proposent des services d'isolation de données permettant à la couche de données d'une application d'accueillir facilement plusieurs locataires (par exemple, les *méta-données* de force.com et les *clés de fédération* de SQL Azure). Nous avons également démontré que ces services introduisent une quantité importante de changement dans le modèle de développement à cause de leurs notions uniques. Ainsi, cela nécessite la conception et la mise en œuvre d'un système d'isolation de données qui peut être configuré de sorte qu'une base de données créée d'une manière non-consciente de la mutualisation puisse devenir facilement mutualisée en l'utilisant. Une base de données non-consciente de la mutualisation est celle qui n'a pas été conçue avec la capacité de différencier un locataire d'un autre. Par conséquent, il devient plus facile de fournir des méthodes de réingénierie d'une base de données existante à locataire unique vers la mutualisation, sans modifier sensiblement l'architecture de l'application et ses mécanismes d'accès aux données existants.

De ce fait, le sujet principal de notre contribution à niveau de la mutualisation d'une base de données consiste à proposer un système d'isolation de données sous forme d'un service (DIaaS). Ce système fournit une capacité de gestion de la mutualisation des données d'une application invoquée par un locataire, de telle sorte que l'application accède uniquement aux données associées à ce locataire sans la nécessité qu'elle soit programmée auparavant, ni de connaître toutes les informations de ce locataire pour fonctionner. Le système d'isolation proposé est complètement externalisé. Il est utilisable à la demande et suppose l'interaction avec des applications initialement non-conscientes de la mutualisation sur la couche de données. Le système reçoit des requêtes d'accès aux données émises par ces applications et renvoie à chacune d'entre elles les résultats appropriés. Du point de vue de l'application, ce système semble être une interface d'accès aux données classique. Cela signifie qu'il accepte des requêtes d'accès aux données et exécute ces requêtes pour finalement retourner les résultats attendus au locataire qui avait principalement invoqué l'application. Tel qu'il est utilisé par la suite, une « *requête spécifique d'accès aux données* » est une requête qui ne contient pas un identificateur unique qui identifie un locataire ou la base de données dans laquelle ses données existent.

7.2.1 La variabilité du système DIaaS

Pour répondre au mieux aux exigences des applications SaaS mutualisées concernant l'isolation de données des locataires, notre système DIaaS couvre tous les aspects d'isolation de données déjà identifiées (voir section 5.2). Ces aspects reviennent principalement à :

- *Séparer les données des locataires :* à travers le choix d'un modèle de séparation parmi les trois modèles présentés. Nous avons évoqué que ce choix devra être effectué suite à l'évaluation d'un ensemble de considérations économiques et techniques.
- *Personnaliser le schéma partagé :* à travers l'ajout d'extensions spécifiques permettant de répondre aux besoins spécifiques de chaque locataire. De même, nous avons présenté plusieurs méthodes de personnalisation, chacune présentant ses propres avantages et inconvénients.
- *Sécuriser l'accès aux données de locataires :* à travers la mise en œuvre d'un mécanisme de sécurisation sophistiqué permettant de garantir qu'un locataire n'aura jamais le privilège d'accès aux données d'autres locataires. Deux types de mécanismes ont été présentés à ce niveau : le premier est complètement géré à partir de l'application sur une base de transformation de requêtes, tandis que le second exploite les capacités d'un SGBD à fournir des mécanismes de contrôle d'accès assez granulaires.

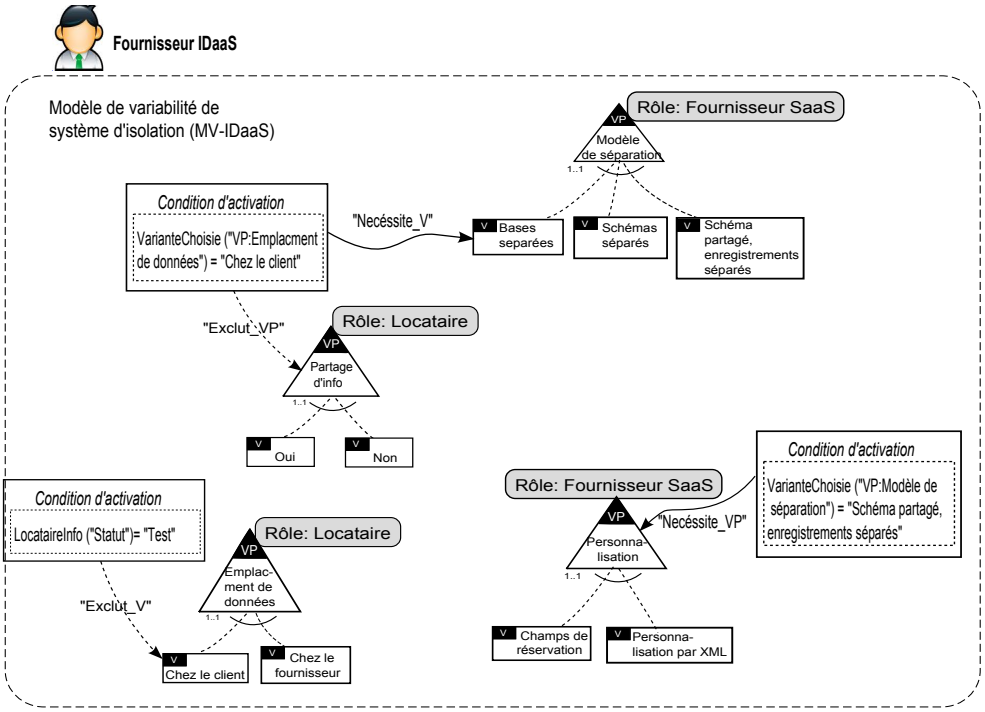


Figure 7.1 – Le modèle de variabilité du système DIaaS

À ce stade, il faut noter que les deux derniers aspects d'isolation (personnaliser le schéma partagé et sécuriser l'accès aux données de locataires) dépendent du modèle de séparation choisi. Ils ne doivent pas être considérés comme des caractéristiques obligatoires de notre système d'isolation si les données des locataires sont déjà séparées dans des bases ou des schémas de données distincts. De plus, le partage d'informations entre les locataires, que doit être selon nous supporté pour permettre leur collaboration, et possible uniquement dans le cas où les locataires partagent le même schéma de données. De ce fait, un système d'isolation à la demande doit explicitement modéliser ces différentes options (modèle de séparation de données, méthode de personnalisation de données, possibilité de partage d'informations, etc.) et capable d'être configuré selon la stratégie d'isolation de données souhaitée par les fournisseurs de SaaS (les clients du système DIaaS). Nous nous basons dans cette modélisation sur une approche de variabilité réutilisant la technique de modélisation présentée dans le chapitre précédent (voir section 6.3.2.2).

La figure 7.1 montre le modèle de variabilité de notre système d'isolation. Il est défini à travers la réutilisation du cadre de modélisation offert par le système VaaS (voir section 6.4). Il faut noter que les applications qui n'exigent pas une persistance de données durant leur utilisation ne sont pas concernées par le défi d'isolation de données, et donc ne présentent pas le besoin d'utiliser ce système. Ainsi, nous insistons sur l'importance de séparer dans cette thèse entre le système VaaS qui gère la mutualisation uniquement au niveau de l'application (représentée par le défi de gestion de la variabilité), et le système DIaaS qui s'occupe uniquement de la gestion de la mutualisation sur la couche de données.

Les points de variation du modèle de variabilité appartenant au système DIaaS sont repartis dans leur binding entre le fournisseur de l'application cliente et les locataires. Tel que les règles du système VaaS

l'oblige, ce binding doit être effectué en premier temps par le fournisseur de l'application sur les points de variation attaché au rôle *fournisseur SaaS*.

- **Rôle fournisseur SaaS :** ce rôle est le premier à définir la stratégie souhaitée concernant l'isolation de données et ce, en faisant le binding de la variabilité du système DIaaS sur les deux points de variation associés à ce rôle (voir figure 7.1). Ceci concerne les points de variation suivants :
  1. *PV Modèle de séparation :* il permet de choisir le modèle de séparation de données parmi les trois modèles supportés (Variantes : *Base de données séparées*, *base de donnée partagée et schéma séparés*, *schéma partagé et enregistrement séparés*).
  2. *PV Personnalisation :* il permet de choisir une méthode de personnalisation du schéma partagé parmi deux méthodes supportées (Variantes : *Champs de réservation*, *Personnalisation par XML*).
- **Rôle Locataire :** les choix des variantes effectués par le fournisseur de SaaS affectent pour l'instant tous les locataires de l'application, qui sont autorisés de leur côté à faire le binding des points de variations suivants :
  1. *PV Partage d'information :* il permet de choisir si le locataire souhaite partager ses données avec d'autres locataires (Variantes : *Oui*, *Non*).
  2. *PV Emplacement de données :* il permet choisir l'emplacement de données du locataire en fonction de ses mesures de sécurité (Variantes : *Chez le client*, *Chez le fournisseur*). En choisissant de placer ses données sur sa propre infrastructure (sélectionner la variante *Chez le client*, le locataire modifiera pour sa configuration le modèle de séparation déjà choisi par le fournisseur (s'il est différent). Cela est en raison de l'existence d'une contrainte de dépendance qui pourra être activée par ce choix de variante et qui nécessitera l'installation d'une base de données séparée pour ce locataire. D'ailleurs, le fournisseur de l'application est toujours capable, depuis son modèle de variabilité principale, de poser une contrainte d'exclusion sur ce PV dans le cas où il ne souhaite pas gérer sa complexité supplémentaire.

Par la suite, nous allons expliquer l'utilisation de ce modèle de variabilité dans le cadre du processus d'externalisation de l'isolation de données à travers l'architecture du système DIaaS.

7.2.2 L'architecture de DIaaS et le processus d'externalisation de l'isolation de données

Les figures 7.2 et 7.3 décrivent l'architecture du système DIaaS et le processus à suivre pour externaliser la gestion de l'isolation de données. Ce processus défini l'ensemble des étapes qui sont divisées en deux catégories principales : *la configuration du système DIaaS* et *l'isolation de données à travers ce système*.

7.2.2.1 Étapes de configuration du système DIaaS

Ces étapes ont pour but de prendre les actions initiales nécessaires pour préparer la couche de données d'une application cliente d'avoir la capacité d'accueillir plusieurs locataires. Cette préparation est réalisée d'une façon complètement automatisée à travers notre système sans distinguer entre un nouveau développement d'application en mode mutualisé, et une application existante à locataire unique à transformer pour mutualiser sa couche de données. Les deux cas peuvent être facilement supportés grâce à l'architecture proposée.

La première étape de cette catégorie consiste à effectuer le binding du modèle de variabilité du système DIaaS (voir figure 7.1) par le fournisseur de l'application cliente, et ce sur les points de variation

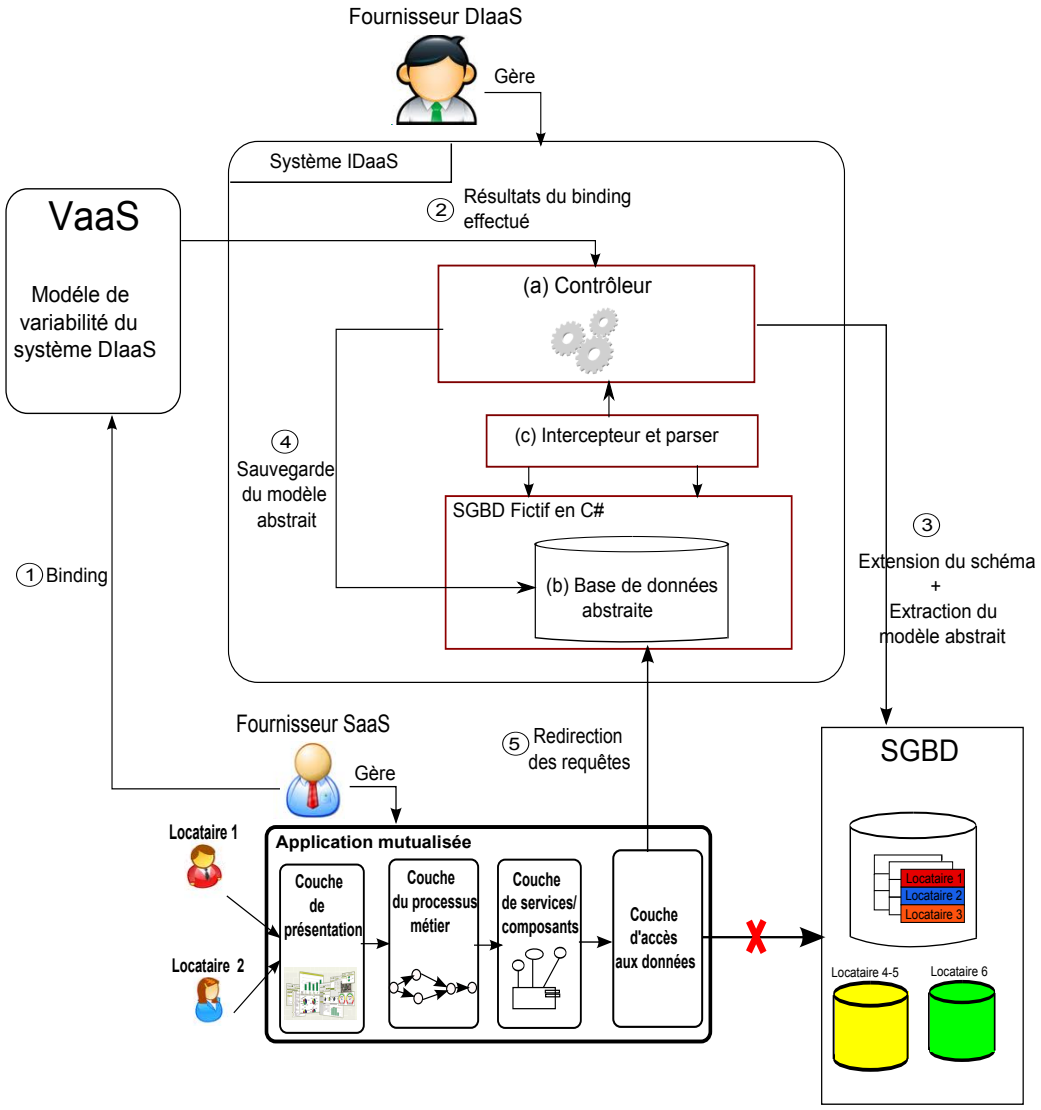


Figure 7.2 – Architecture de DIaaS : étapes de configuration du système

réservés au rôle *fournisseur SaaS* (1). Ce binding est réalisé à travers l'accès au système VaaS. Une fois ce binding effectué, ses résultats seront envoyés au contrôleur (a) pour les analyser et déterminer les premières actions à effectuer pour préparer la base de données de l'application à accueillir plusieurs locataires(2). En fonction de cette analyse, le contrôleur accède au SGBD de l'application et effectue les actions appropriées (3). Par exemple, si le modèle de séparation de données au niveau enregistrements est celui choisi lors du binding de la variabilité (variante *schéma partagé, enregistrements séparés*), ces actions consistent à étendre automatiquement toutes les tables du schéma partagé de l'application par l'ajout d'une colonne supplémentaire : *LocataireID*, et de préparer ce même schéma pour pouvoir être

personnalisé (en fonction de la méthode de personnalisation choisie, par exemple, l'ajout de champs de réservation ou d'extension par XML). Dans le cas où l'un des deux autres modèles de séparation est choisi (*bases de données séparées* ou *schémas séparés*), aucune action ne sera effectuée pour l'instant. Dans les deux cas, le contrôleur du système extrait de la base de données de l'application *son modèle abstrait* (c.à.d. sa structure du schéma en termes des tables, des colonnes et des relations entre les tables) et la sauvegarde (4) dans une *base de données abstraite* spécifique au stockage des schémas de données (mais sans stockage de données réelles) (b) elle-même installée sur un *SGBD fictif* intégré dans ce système DIaaS (voir section 7.2.3 pour l'explication détaillé de ces deux composants : *base de données abstraite* et *SGBD fictif*). Le but principal de cet SGBD fictif est de pouvoir récupérer en temps réel les requêtes d'accès aux données émise par l'application, par une simple re-configuration de la couche d'accès aux données de celle-ci, afin de les transformer dynamiquement en injectant les filtres appropriés et garantir que la sécurisation d'accès aux données par les locataires est toujours strictement maintenue. Ceci nécessite que le fournisseur de l'application SaaS reconfigure sa couche d'accès aux données (5) pour rediriger ses communications vers ce SGBD au lieu de celui d'origine. Cela sera bien évidemment effectué sans avoir besoin d'installer des composants spécifiques à l'application ni de modifier la structure actuelle de sa couche d'accès aux données. Cette approche permet à notre système d'isolation d'être complètement indépendant de la technologie de l'application et ainsi utile dans les deux cas d'utilisation déjà précisés : (i) pour un nouveau développement d'application en mode mutualisé et (ii) pour une réingénierie d'application existante à locataire unique vers la mutualisation de ses données (passe à des locataires multiples).

7.2.2.2 Étapes d'isolation des données de locataires

Ces étapes concernent principalement la prise d'actions qui garantissent l'isolation des données des locataires dès leur inscription à l'application et durant son utilisation. Cela nécessite tout d'abord que chaque locataire complète de son côté le binding du modèle de variabilité appartenant au système DIaaS (voir figure 7.1), et ce sur les points de variation qui ont été réservés au rôle *locataire* (1). Une fois inscrit à l'application, ce locataire sera redirigé vers une interface de binding fournie par le système VaaS où les points de variation en question seront exposés. D'une manière semblable aux étapes de configuration précédentes, le résultat du binding sera envoyé au contrôleur du système DIaaS (2) pour effectuer les actions nécessaires à la création de ce locataire dans la base de données de l'application. Cette création peut se faire de trois manières différentes en fonction du modèle de séparation choisi :

- *Par la création d'une nouvelle base de données* avec le nom du locataire si la décision d'une séparation physique de données a été déjà prise (choix de la variante *bases séparées* du modèle de variabilité du système DIaaS, voir figure 7.1). Cela est réalisée par le contrôleur du système DIaaS qui récupère en premier (3) temps le schéma de données par défaut de l'application précédemment stockée dans la base de données abstraite et crée ensuite une nouvelle base de données pour ce locataire à travers l'exécution d'un script spécifique (4).
- *Par la création d'un nouveau schéma* dans l'unique base de données de l'application si le modèle de schémas séparés est celui choisi (choix de la variante *schémas séparés*). De même, le contrôleur du système DIaaS récupère (3) le schéma de données précédemment stocké dans la base de données abstraite et exécute un script spécifique pour créer un nouveau schéma de données dans la même base de données existante, tout en ajoutant le nom du locataire sur chaque table de ce schéma (4).
- *Par la création d'un nouvel enregistrement* dans une table de données spécifique à la gestion des informations sur les locataires (4) et ce, bien évidemment dans le cas où le modèle de séparation au

niveau enregistrements est choisi (choix de la variante *schéma partagé, enregistrements séparés*). L'étape (3) de cette catégorie d'étapes est ignorée dans ce cas.

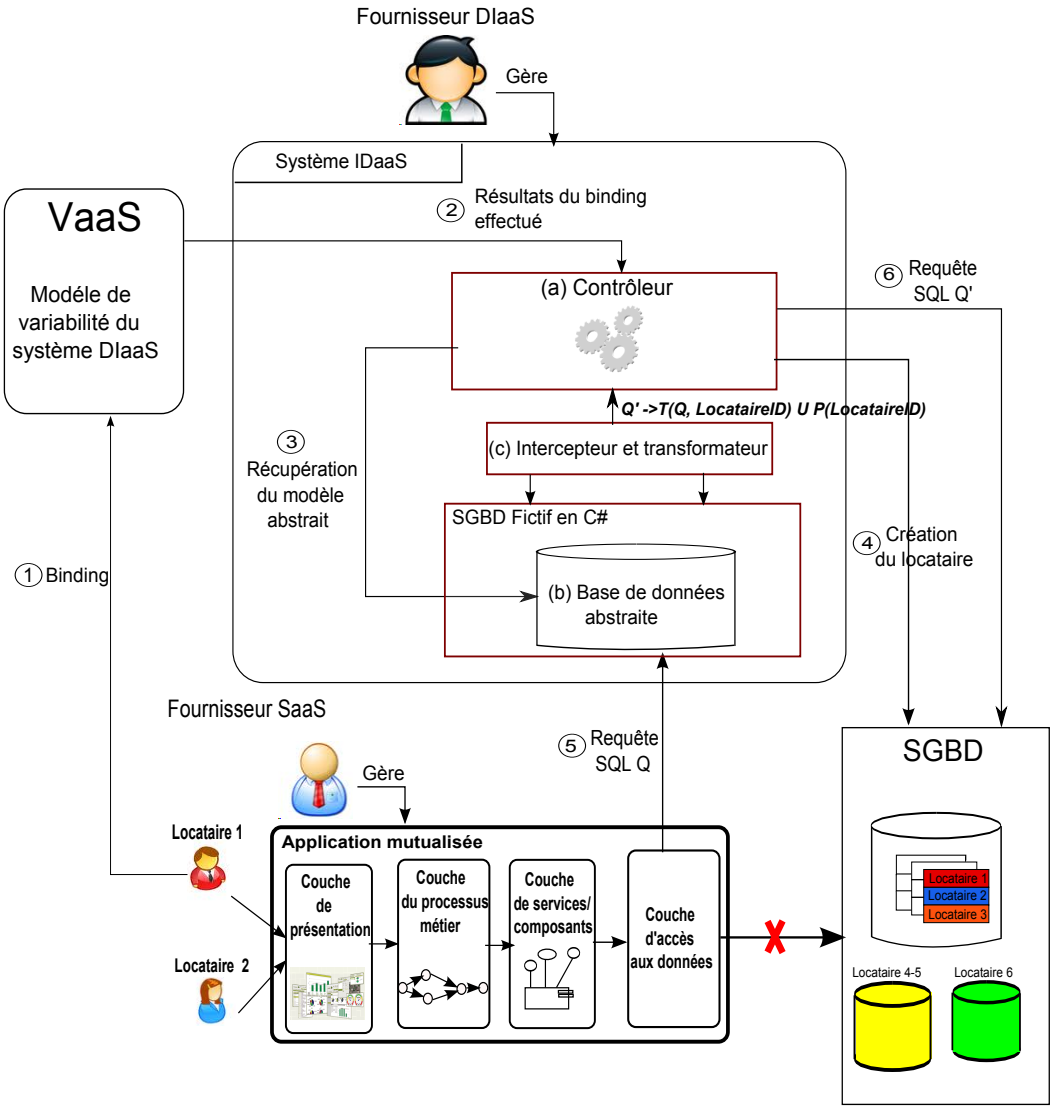


Figure 7.3 – Architecture de DIaaS : étapes d'isolation de données

Une fois que le locataire est créé dans la base de données, il peut commencer à utiliser l'application. Les requêtes d'accès aux données générées à partir d'une telle utilisation seront envoyées vers le SGBD fictif (5), interceptées et transformées par un composant spécifique (c) pour être ensuite renvoyées à la base de données réelle (6). L'objectif final de ces différentes étapes est d'assurer que chaque locataire accède uniquement aux données auxquelles il a le droit d'accéder. Par la suite, nous allons détailler les

composants clés de ce système, ceux qui permettent l'exécution de toutes les étapes du processus et rendent l'externalisation de l'isolation de données possible.

7.2.3 Les composants clés

Notre système DIaaS favorise l'adoption d'une approche mutualisée au niveau des données de l'application, tout en externalisant la gestion de cette mutualisation et déchargeant les fournisseurs de SaaS des coûts de développement et de réingénierie qu'elle implique. Ceci a pu être réalisé grâce à l'architecture de ce système basée sur les trois composants suivants :

Base de données abstraite : ce composant consiste en une base de données classique créée sur un SGBD *« open-source »* [**?**] (voir figure 7.3, (b)). Nous avons dénommé cette base de données *abstraite* parcequ'elle contient uniquement des modèles de données (des schémas) et ne stocke pas de données réelles. Son but est de réaliser une externalisation simple de l'isolation de données à travers la création d'un schéma de données similaire à celui de l'application. Cette similarité permet à l'application de se connecter à cette base de données abstraite sans aucune modification des mécanismes d'accès aux données existants. Ainsi, ce composant représente une couche d'abstraction pour modéliser la base de données de l'application, et vers laquelle les requêtes d'accès aux données seront redirigées. Cela se fait par une simple reconfiguration de la couche d'accès aux données (voir figure 7.2, étape (5)), tout en lui gardant les procédures de gestion de la base de données (réelle) telles qu'elles sont. En plus, ce composant est utile pour créer automatiquement des bases de données séparées ou des schémas séparés lors de l'inscription de nouveaux locataires (si les variantes *bases de données séparées* ou *schémas séparés* sont choisies sur le PV *Modèle de séparation*, voir figure 7.1), puisqu'il détient le schéma de la base de données d'origine.

Intercepteur et transformateur : ce composant est responsable d'intercepter chaque requête envoyée à la base de données abstraite pour la transformer en une nouvelle requête en fonction des informations sur le locataire courant (voir figure 7.3, (c)). L'algorithme de transformation est décrit comme $Q' \rightarrow T(Q, LocataireID) \cup P(LocataireID)$, où T est la fonction de transformation responsable d'injecter les filtres de sécurisation (ex : *Where LocataireID='1'*), et P est la fonction responsable de récupérer les données personnalisées pour le locataire indiqué.

Certaines requêtes n'ont pas besoin d'être transformées avant leur exécution. Par exemple, une requête qui accède aux données disponibles publiquement et accessibles par tous les locataires. Pour mieux traiter les requêtes et minimiser le temps pour effectuer une transaction, ce composant analyse chaque requête envoyée pour vérifier l'existence des tables de données publiques (celles qui ne sont pas étendues par la colonne supplémentaire `LocataireID`).

Contrôleur : le principale rôle de ce composant est de créer les locataires dans la base de données réelle et de maintenir l'emplacement de leur données pour rediriger et exécuter les requêtes au bon endroit (voir figure 7.3, (a)). Vu que toutes les requêtes SQL passent à travers ce composant et que l'identifiant du locataire est connu à ce niveau, des opérations plus complexes au niveau de chaque locataire peuvent être effectuées. Par exemple, un locataire pourrait avoir une action spécifique consistant à restreindre les données qu'il pouvait récupérer par rapport à celles récemment ajoutées, ou interdire le dépassement d'un nombre d'enregistrements dans une table spécifique. Ces extensions peuvent être précisées au niveau de ce composant sans modifier le code de l'application cliente.

7.2.4 Des fonctionnalités supplémentaires

En plus de la séparation de données de locataires et la sécurisation de leurs accès, le système DIaaS fournie des fonctionnalités supplémentaires telles que la personnalisation du schéma partagé et la restauration de données après panne.

Personnalisation du schéma partagé : telle qu'elle est conçue, la base de données de l'application naturellement inclue une configuration standard, des tables par défauts et des relations entre ces tables qui sont appropriées à la nature de l'application. Lors de la mutualisation, les différents locataires vont certainement avoir leurs besoins spécifiques. Tandis qu'une conception rigide et inextensible du schéma de données sera incapable de supporter de tels besoins. Ainsi, il est nécessaire de mettre en œuvre des méthodes de personnalisation à travers lesquelles le schéma de données de l'application puisse être étendu en fonction de chaque locataire. Notre système DIaaS supporte actuellement deux méthodes de personnalisation : *les champs de réservation* et *l'extension par XML* (voir section 5.2.2).

L'approche la plus simple est celle de champs de réservation où le schéma de données est simplement étendu pour créer un nombre prédéfini de champs supplémentaires dans chaque table à étendre. Cependant, les locataires utilisant uniquement un sous-ensemble de champs de personnalisation disponibles, créent des valeurs nulles dans les champs non-utilisés provoquant un gaspillage des colonnes et ainsi de mémoire. En comparant cette approche avec celle qui propose une extension par XML, le problème des valeurs nulles est résolu à l'exception que les documents XML nécessitent un traitement et une transformation qui introduisent plus de complexité de calcul et ralentie l'exécution des requêtes.

Pour choisir entre l'une des deux variantes, les auteurs dans [**?**] proposent la fonction d'évaluation suivante : $\mu \times (\alpha / \beta) + (1-\mu) \times \gamma$, où μ signifie la proportion d'utilisation des champs de personnalisation par les locataires, α signifie le nombre de requêtes d'accès aux tables personnalisées et γ signifie le niveau de service requis. Plus la valeur retournée par cette fonction est élevée, moins la personnalisation par XML est appropriée.

Restauration de données après panne : cette fonctionnalité de notre système permet à l'application de restaurer ses données après panne, et ce en configurant un backup continue des données. En effet, il existe deux types de backup de données, le premier est un backup complet en copiant à chaud le fichier qui représente physiquement la base de données (par exemple *database.bak* pour SQL Server) pour recréer la base de données à un certain stade. Le deuxième type consiste à utiliser les fichiers de log qui conservent en mémoire toutes les requêtes envoyées à la base de données, y compris celles de définition de données telles que la création des tables, des relations, des permissions, etc.

La situation idéale est de restaurer les données à partir des fichiers de log vu qu'ils contiennent toutes les modifications des données effectuées jusqu'au le moment de la panne. Sachant que, cela nécessite une longue durée d'exécution. Notre idée consiste à combiner les deux types de backup, de sorte que nous sauvegardons les fichiers du backup complet d'une manière périodique et nous conservons aussi des copies de tous les log des requêtes au niveau du contrôleur du système DIaaS. En cas de panne, nous effectuons dans un premier temps un backup complet à partir du dernier fichier sauvegardé (backup complet) et nous exécutons toutes les requêtes conservées dans les fichiers de log à partir de la date de ce dernier backup.

Dans le pire des cas, si le dernier fichier du backup complet est corrompu, aucune donnée ne sera pas perdues. Même dans ce cas il est possible de restaurer autant de données que nécessaire, mais

cela prendra plus du temps. Puisque notre idée combine entre un fichier de backup complet et des fichiers de log pour remettre la base de données dans son état, chaque backup est équivalent à celui antérieur et les requêtes de log à partir de la date de ce backup. Cela est exprimé de la manière suivante : *Backup(N) = Backup(N-1) + LogsDepuis(Backup(N-1))*. De ce fait, si le dernier fichier de backup complet est endommagé nous pouvons le substituer par le précédent et élargir ensuite la période des requêtes à exécuter à partir des fichiers de log.

Après avoir présenté le système DIaaS, nous allons présenter dans la section suivante notre approche générale de gestion de la mutualisation de SaaS par externalisation. Cette approche regroupe les systèmes DIaaS et VaaS ainsi que d'autres services liées à l'administration et à la sécurité des applications SaaS mutualisées dans une plateforme dédiée à la gestion de ce type d'applications.

7.3 Mutualisation de SaaS sous forme d'un service

Généralement, chaque fournisseur de SaaS préfère mutualiser ses propres applications. Son objectif à travers cette mutualisation est d'atteindre les bénéfices associés en termes de réduction des coûts opérationnels et infrastructurels de l'application, ainsi que l'exploitation des économies d'échelle et la diminution des coûts d'abonnement pour les locataires. Sur les plans conceptuel et technique, la mutualisation introduit une complexité de conception qui nécessite des efforts supplémentaires et des engagements dans de nouveaux développements pour relever ses défis et de la gérer d'une manière efficace [GSH+07].

Pour beaucoup de fournisseurs de SaaS, la préoccupation essentielle est celle de la maîtrise de la technologie de développement et la connaissance du domaine métier de l'application, la gestion de la mutualisation représente un détournement vers un nouveau domaine dans lequel ils ne sont pas forcement des experts. La situation est encore plus difficile à gérer lorsqu'il s'agit de la transformation d'une application existante à locataire unique en une application mutualisée. Le fournisseur de cette application doit faire face à une tâche de ré-ingénierie relativement complexe [BZ10a]. Tout d'abord, il doit retravailler l'ensemble de ses mécanismes d'accès aux données de sorte que les différents locataires puissent partager une instance de base de données unique. Cela doit être réalisé parfaitement sans manquer un point d'accès, sinon il risque de compromettre la sécurité des données. Le fournisseur doit également étendre le modèle de données de l'application pour inclure des informations spécifiques à chaque locataire et différencier leurs données. Bien évidemment, cela représente beaucoup de travail et ce n'est que le début. L'application doit également inclure des capacités administratives de gestion pour faire face à la notion de locataire vu qu'elle n'était pas conçue dès le début pour fonctionner en mode mutualisé. Cela implique plus d'interfaces de gestion, des rapports et des contrôles d'accès à développer. De même, beaucoup de fonctionnalités qui étaient configurées à un niveau global devront être refaites sur la base de chaque locataire, et ce, dans le but d'une gestion dynamique de la variabilité. Cela représente encore plus de travail à la charge du fournisseur de SaaS, et par conséquent, il est communément connu que la mise en œuvre de la mutualisation de SaaS dépasse souvent le temps et les estimations des coûts initiaux. Ceci pourra probablement distraire l'équipe de développement et reporte le travail sur les fonctionnalités métiers de base qui sont les valeurs clés du marché pour les fournisseurs de SaaS. Ainsi, ces derniers vont certainement essayer d'éviter cette complexité en utilisant des technologies éprouvées et pré-packagées.

Pour cela, nous avons regroupé les systèmes VaaS et DIaaS ainsi que d'autres services d'administration et de sécurité dans une même plateforme dédiée à la gestion de la mutualisation de SaaS. Cette plateforme, que nous avons dénommé *MaaS (Multitenant as a Service)* présente la nouveauté, par rapport à des nombreuses plateformes que nous avons examiné et analysé (voir section 7.4) de fournir les composants logiciels nécessaires pour cette gestion d'une manière faiblement couplée avec les applications

clientes. Cela signifie que les applications clientes ne doivent pas s'occuper des opérations qui se produisent à l'intérieur de cette plateforme, et par conséquent, elles sont indépendantes de son architecture et sa technologie de développement. Cette caractéristique de couplage faible est un des facteurs principaux qui différencient notre plateforme de celles existantes. La maturité que présente notre plateforme est soutenue par les expérimentations effectuées à l'échelle industrielle au sein de la société BITASOFT. *Pour des raisons de confidentialité, les détails de ces expérimentations ne sont pas inclus dans ce manuscrit*.

7.3.1 L'architecture de la plateforme

La plateforme MaaS fournie des capacités de gestion de la mutualisation, qu'il s'agisse d'un nouveau développement ou d'une application existante à transformer. La figure 7.4 montre l'architecture de cette plateforme. Le seul pré-requis à son utilisation est que l'application cliente soit construite suivant le modèle d'architecture proposé par l'AOS, afin que le système VaaS puisse résoudre sa variabilité à distance. Mis à part ce pré-requis, les capacités de cette plateforme sont fournies à la demande sans imposer un modèle de développement ou une technologie d'application spécifique. Les fournisseurs de SaaS ont la liberté de choisir la plateforme et l'infrastructure souhaitées pour développer et déployer leurs applications. La plateforme MaaS est uniquement dédiée à la gestion de la mutualisation, dont l'utilisation suit des processus d'externalisation bien définis concernant la gestion de la variabilité et l'isolation des données. Son objectif est de permettre à l'application d'avoir une capacité de gestion native de la mutualisation. En plus de VaaS et DIaaS, cette plateforme fournie des services supplémentaires essentiels pour la sécurité et l'administration des applications SaaS mutualisées (voir section 7.3.2). Avant de détailler ces services, il faut noter qu'un élément clé de l'architecture de MaaS est la séparation claire qu'elle établie entre deux types de développeurs : le premier type concerne ceux qui sont orientés vers la logique métier de l'application et qui sont probablement non-conscients de la mutualisation et responsables uniquement du développement des interfaces graphiques, processus métier et services. Le deuxième type de développeurs concerne ceux qui sont orientés infrastructures et responsables des interactions avec cette plateforme. Ces derniers sont plus familiers avec les détails des ressources sensibles de l'application impactées par la mutualisation, et doivent veiller à ce que celles-ci fonctionnent efficacement, de façon fiable à faibles coûts.

Les avantages de l'architecture de MaaS sont doubles. D'un côté, elle soulage la plupart des développeurs de la complexité de la mutualisation de SaaS en simulant un environnement virtualisé de développement d'application à locataire unique. De toute évidence, cela contribue à la simplification et à l'accélération du développement des applications mutualisées, et ainsi à l'amélioration de manière significative de leur maintenabilité.

7.3.2 Des services supplémentaires

La plateforme MaaS fournie un ensemble de services pour couvrir tous les aspects de gestion de la mutualisation de SaaS. Dans les sections précédentes, nous avons expliqué en détails le fonctionnement des services fournissant des capacité de gestion de la variabilité (à travers le système VaaS) et de l'isolation des données de locataires (à travers le système DIaaS). Dans cette section nous allons nous concentrer sur le reste de services de gestion qui concernent principalement la *sécurité de l'application* et l'*administration de locataires*. Toutefois, ces différents services ont besoin de collaborer entre eux pour offrir la logique d'exécution globale de la plateforme. Par exemple, le système DIaaS communique avec le système VaaS pour modéliser sa variabilité et récupérer les résultats de bindings effectués. De même, le système VaaS a besoin d'accéder aux informations concernant les locataires afin de pouvoir évaluer

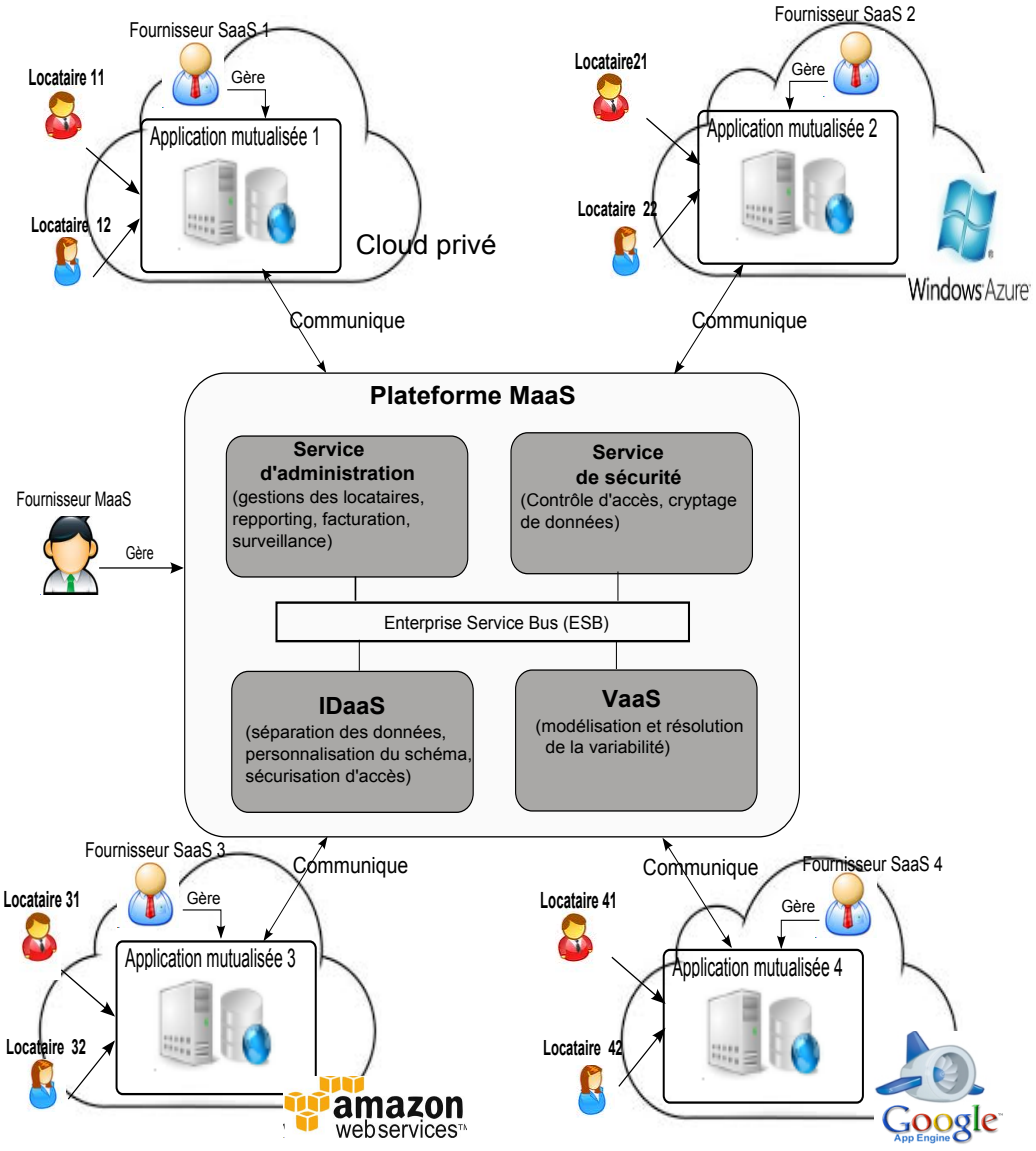


Figure 7.4 – Architecture de la plateforme MaaS

les conditions d'activation de différents modèles de variabilité existants (voir section 6.3). Ainsi, ces différents services sont inter-connectés à travers un ESB (Enterprise Service Bus) [Men07] pour permettre une communication facile entre eux. Dans ce qui suit, nous allons détailler ces services supplémentaires.

7.3.2.1 Service de sécurité

Dans un contexte SaaS mutualisé, la sécurité de l'accès à l'application et ses données représente un besoin essentiel [PLL10]. Étant donnée que les données de l'application se trouvent à l'extérieur de

l'organisation des locataires, ces derniers se méfient généralement de la sécurité. L'objectif de la prise en considération de ce besoin est de veiller à ce que les locataires soient confiants et convaincus par les normes de sécurité qui leur sont fournies. Par conséquent, il est préférable de se conformer aux normes et certifications de sécurité appropriées pour maintenir leur confiance. Nous allons nous concentrer sur deux aspects de la sécurité des applications SaaS mutualisées : le *contrôle d'accès* et le *cryptage de données*, qui sont actuellement supportés par notre plateforme.

Contrôle d'accès : un aspect principal de la sécurité dans une application SaaS mutualisée concerne le contrôle d'accès aux ressources de l'application. Il consiste principalement à donner l'exclusivité d'accès à une partie de ressources de l'application (ex : données, services, pages Web, etc.) à un ensemble de locataires bien précis, et les interdire aux autres locataires qui n'en ont pas actuellement besoin. Ceci est généralement lié au modèle économique de SaaS qui facture l'utilisation de l'application selon les ressources utilisées. Généralement, cet aspect est considéré comme compliqué à gérer parce que les besoins des locataires ne sont pas toujours les mêmes par rapport à la politique de sécurité requise. Cela nécessite de mettre en place un système de contrôle d'accès avec un certain niveau de flexibilité, qui signifie idéalement que l'application doit laisser les locataires définir eux-même leurs besoins en sécurité. Dans les applications à locataire unique, la règle générale de contrôle d'accès est basées sur des rôles. Ces derniers décrivent la hiérarchie interne de l'organisation cliente dont la politique de contrôle d'accès est définie en fonction. Toutefois, dans une application SaaS mutualisée, chaque locataire peut avoir une hiérarchie différente, et par conséquent, des rôles différents dans cette hiérarchie. Ainsi, les mécanismes de contrôle d'accès à base des rôles existants (RBAC [FK09], TMAC [GMPT01], etc.) n'assurent pas la flexibilité requise. Un mécanisme de contrôle d'accès nécessite d'être plus générique pour que n'importe quelle hiérarchie de locataire puisse l'utiliser suite à une configuration précise. De ce fait, l'approche adoptée par ce service pour contrôler l'accès aux ressources de l'application consiste à utiliser *les privilèges*. Ces derniers sont des droits d'accès définis au préalable d'une manière indépendante, et assignés aux rôles par les locataires eux-mêmes. Une fois l'utilisateur appartenant à un locataire mis en correspondance avec un rôle précis, il reçoit automatiquement les privilèges appropriés. Sur cette base, notre service de sécurité peut contrôler ce que l'utilisateur fait dans le système. Généralement, le contrôle d'accès peut être effectué dans des endroits différents de l'application. Ceci peut bien être effectué sur les entités de données tels que l'ajout et la mise à jour des enregistrements dans la base de données, ou bien sur un champ de données spécifique, par exemple, un utilisateur met à jour les détails d'un employé mais pas les informations du salaire. Ces types de contrôle d'accès qui concernent les données de l'application sont effectués au niveau du contrôleur du système DIaaS, et ce avant l'exécution des requêtes (voire section 7.2.3 pour plus de détails sur le fonctionnement du contrôleur). Les contrôles d'accès qui concernent la couche de présentation de l'application (GUI) sont à gérer par l'application elle-même. Nous rappelons que la plateforme MaaS est faiblement couplée avec l'application, et n'a pas un accès direct au code source des pages Web constituant sa couche de présentation. L'application devra ainsi interroger ce service pour récupérer les privilèges d'accès associés au rôle actuel et effectuée ensuite les actions appropriées.

Cryptage des données il constitue un autre aspect essentiel de la sécurité au sein d'une application mutualisée et doit être soigneusement étudié et planifié. Le cryptage est destiné à protéger l'intégrité et la confidentialité des données de chaque locataire. En d'autres termes, nous devons empêcher que les données d'un locataire soient accédées ou modifiées par d'autres locataires non-autorisés. Cela peut être effectué suite une tentative de piratage par l'exploitation d'une faille potentielle de

sécurité dans l'algorithme de transformation de requêtes que nous avons implémenté au niveau du système DIaaS. En plus, les données peuvent être accessibles par les locataires non-autorisés lorsque elles sont stockées dans la mémoire ou même échangées à travers les réseaux. Une manière traditionnelle pour protéger les données consiste à les crypter en utilisant des clés de cryptage et de décryptage. Cependant, dans une application mutualisée, le fait de partager les clés entre tous les locataires n'a pas de sens, car cela peut uniquement empêcher un locataire des attaquants externes, mais pas des attaques éventuelles d'autres locataires qui ont également l'accès aux mêmes clés. Par conséquent, chaque locataire doit posséder un ensemble unique de clés (sans les divulguer à d'autres locataires) pour crypter ses données critiques et privées. Théoriquement, nous pouvons crypter toutes les données avec l'algorithme le plus efficace dans n'importe quelle situation. Cependant, un compromis entre la sécurité et la performance devrait être adopté. Nous devons assurer un juste niveau de sécurité à travers le cryptage, sans plus [**?**]. D'un point de vue pratique, nous avons proposé les principes suivants pour faire un compromis acceptable en ce qui concerne le cryptage de données dans une application mutualisée :

- *Uniquement crypter les données les plus critiques :* en règle générale, la criticité de données peut être mesurée par domaine d'application spécifique (ex : les données financières peuvent avoir une plus grande priorité).
- *Sélectionner l'algorithme de cryptage approprié :* en général, les algorithmes de cryptage avec une sécurité renforcée peuvent affecter la performance de l'application. Dans certains cas, nous pouvons choisir des algorithmes de chiffrement mixtes pour les compromis. Par exemple, utiliser l'algorithme à base de clés publiques et privées pour protéger les clés symétriques qui sont finalement utilisées pour crypter les données [CCW06].

7.3.2.2 Service d'administration

La gestion d'une application SaaS mutualisée nécessite la mise en œuvre de nombreuses opérations d'administration. Celles-ci concernent principalement la gestion des informations sur les locataires et leur cycle de vie, ainsi que la surveillance de leur utilisation de l'application pour mesurer leurs activités effectuées et facturer l'utilisation de l'application en fonction. Voici les opérations d'administration principales supportées par ce service :

Gestion des locataires : cette fonctionnalité s'occupe principalement de l'inscription et la désinscription des locataires. Lors de l'inscription, il existe de nombreuses tâches à exécuter et à configurer. Il s'agit notamment de renseigner les détails du nouveau locataire et l'exécution des scripts spécifiques pour créer ses configuration par défaut. Il s'agit également de sa mise en correspondance avec des privilèges d'accès qui définissent les modules et les fonctionnalités de l'application à bloquer ou débloquer. La désinscription d'un locataire implique l'arrêt automatique du service pour tous ses utilisateurs. Les tâches à exécuter dans cette situation concernent l'exportation de toutes ses données sous forme des fichiers de format standard (ex : XML, CSV, ODF, etc.) pour qu'il puisse les exploiter ultérieurement.

Surveillance, mesure et facturation : cette fonctionnalité consiste à documenter des événements et des utilisations spécifiques qui peuvent ensuite être facturées. Par exemple, si un fournisseur de SaaS envisage que les locataires paient selon le nombre d'utilisateurs crées, il doit être capable d'être informé en temps réel des changements qui s'opèrent à ce niveau. Ce service propose des fonctionnalités spécifiques pour l'affichage et la récupération des événements mesurés et qui se produisent au niveau de la base de données de l'application, tels que l'ajout et la suppression des utilisateurs

ou d'autres types de données explicitement indiqués en tant qu'unités de mesures. En plus, ce service supporte chaque application avec un système de paiement en ligne où les locataires saisissent leurs identifications bancaires. Si l'application cliente n'appelle pas à une facturation automatique (par exemple, un paiement par chèque) ce service peut être configuré en mode « *manuel* », mais il continuera à générer des factures récupérables dans l'espace privé de chaque locataire.

Après avoir présenté l'architecture de la plateforme MaaS et ses principaux services, nous allons nous concentrer dans la section suivante sur la présentation des travaux similaires concernant également d'autres plateformes de gestion de la mutualisation de SaaS. Nous allons ainsi classifier et évaluer leurs capacités et montrer la différence avec notre approche.

7.4 Travaux similaires : classification et évaluation

Le développement, le déploiement et la gestion d'exécution des applications ont toujours été dépendants des plateformes spécifiques. Des exemples de plateformes qui dominent actuellement le marché sont *.NET* de *Microsoft* (à travers le cadre de développement Web *ASP.NET*), *JAVA/J2EE* d'*Oracle* (à travers le cadre *JSP*) et *Ruby* (à travers le cadre *Ruby on rails*). Ce sont des plateformes assez répandues offrant des technologies éprouvées et bien compatibles, avec différents outils et meilleures pratiques pour développer des applications Web. Dès lors du passage au modèle du cloud, la nécessité de pouvoir mutualiser les applications impose de nouvelles exigences en matière de technologies des plateformes dans ce domaine. Il est ainsi concevable que certaines de ces plateformes soient étendues pour supporter la mutualisation des applications. Mais il est probable que les grands acteurs du marché de plateformes de développement ait besoin d'une véritable conception adaptée à la spécificité des applications SaaS mutualisées, ce qui entraînera une potentielle discontinuité sur ce marché avec l'émergence des nouvelles architectures, normes et nouveaux modèles de programmation.

Selon les auteurs dans [**?**], il existe actuellement deux alternatives pour fournir des plateformes avec un support pour la construction des applications SaaS mutualisées : la première consiste à utiliser des services PaaS « classiques », alors que la deuxième consiste à fournir des plateformes sous forme de produits à installer dans l'environnement de développement de chaque client. Dans la suite, nous allons détailler et évaluer chacune de ces deux alternatives à travers des exemples concrets de plateformes actuellement présentes sur le marché. Nous précisons que le nombre de plateformes qui pourraient être utilisées pour supporter la mise en œuvre de la mutualisation dépassent celles qui sont citées ci-dessous. Le marché est nouveau et hautement volatile : de nouvelles plateformes sont apparues et d'autres ont arrêtées leurs services durant le développement de cette thèse. Néanmoins, les principaux alternatives de fourniture sont indiqués par la suite.

Mutualisation de SaaS à travers un service PaaS classique : dans ce mode, la plateforme est déployée et gérée par un fournisseur tiers et les clients acquièrent les informations d'identification pour accéder à la plateforme et utiliser les services proposés. Nous allons nous concentrer sur les plateformes Google App Engine*Google App Engine* [Ciu09] et *ForeSoft* [**?**] comme exemples.

- *Google App Engine (GAE) :* c'est l'un des exemples les plus cités d'un service PaaS. La plateforme GAE est introduite en avril 2008. Elle est constituée d'un ensemble de services que les développeurs peuvent utiliser pour créer leurs propres applications. Ces services comprennent l'accès aux données (à travers le système Google data store), la création d'interfaces graphiques, l'authentification (lié au système d'authentification propre à Google) et le déploiement. Pour créer une application sur GAE, les développeurs doivent tout d'abord télécharger un SDK (Kit de développement logiciel) qui inclut toutes les librairies de programmation nécessaires. Ces

dernières sont à installer sur l'environnement de développement *Eclipse*, et uniquement accessibles à partir de l'un des deux langages de programmation supportés : Java et Python. Les postes de développement utilisés doivent toujours s'assurer d'une connexion Internet pour pouvoir communiquer avec les services de la plateforme et tester l'application avant le déploiement. Les application déployées sur GAE s'exécutent dans un environnement virtualisé et peuvent automatiquement passer à une grande échelle (des milliers d'utilisateurs), sans aucune intervention ou gestion nécessaires. Dans les couches inférieures du service, GAE fonctionne sur la même infrastructure logicielle et matérielle dont Google utilise pour son moteur de recherche. Cette infrastructure est masquée par la couche de GAE et les développeurs n'ont pas de contrôle direct sur les ressources réservées à l'exécution de leurs applications. La plateforme persiste les données des applications à travers le système *Google data store* supporté par les technologies suivantes : *Google Big Table* [**?**] (un mécanisme de persistance de données non-relationnelles) et *Google File System* [**?**](un système de fichiers distribués). Sachant que ces technologies ne sont pas directement accessibles par les développeurs, qui s'appuient dans leur développement sur des interfaces abstraites d'accès aux données telles que *Java Data Objects* (JDO) et *Java Persistence API* (JPA).

GAE facilite le développement des applications mutualisées en proposant des services d'isolation des données de locataires via l'API *Namespace* [**?**]. À travers cette API, les données d'application sont partitionnées entre les locataires en spécifiant pour chacun entre eux un *Namespace* unique (l'identifiant du locataire). La gestion de ces *Namespaces* se fait à travers un gestionnaire dédié, appelé *NamespaceManager*. Chaque entité de données créée sur *Google data store* doit être marquée par un *Namespace* précis. Lors de l'envoie d'une requête d'accès aux données, le *Namespace* courant (équivalent au locataire connecté) doit être associé avec cette requête de sorte que le *Google data store* puisse filtrer les données. Ainsi, chaque locataire de l'application accède uniquement à ses propres données. Si l'application possède des données communes entre les locataires, GAE propose de définir un *Namespace* publique. L'API *Namespace* est aussi prise en charge par d'autres services de GAE, tels que le service de mise en cache *MemCach* et le service de gestion des tâches *TaskQueue* (pour différencier les données mises en cache et les tâches à exécuter pour chaque locataire). L'API *Namespace* est également soutenue par *AppScale* [**?**], la version *open-source* de GAE.

– *ForeSoft :* Il s'agit d'une entreprise qui a été créée en 2001 en tant qu'ASP (Application Service Provider), et a évolué au fil du temps vers un fournisseur d'application SaaS, y compris la fourniture des solutions CRM (Client Relation Management). Les applications de cette entreprise sont hébergées dans un centre de données tiers. La plupart de ses revenus sont tirés des frais d'abonnements à leurs applications par un grand nombre de clients distribués partout à travers le monde. En 2006, ForeSoft introduit dbFlex [**?**], une plateforme de développement issue de la technologie sous-jacente de ses applications. dbFLEX est une plateforme applicative offerte en tant que service et développée en C# et en s'appuyant sur l'environnement .NET et *SQL Server* de Microsoft comme des technologies clés. La plateforme est conçue pour passer automatiquement à l'échelle en ajoutant des capacités de traitement supplémentaires une fois ces capacités sont disponibles. Il n'y a pas un langage de programmation traditionnel offert aux développeurs à travers cette plateforme. Le développement des applications se fait via des outils graphiques. Celles-ci présentent une architecture trois-tiers (interfaces graphiques, logique métier et base de données), dont la logique métier et les interfaces graphiques sont codées sous forme de métadonnées interprétées à l'exécution pour permettre leur adaptation dynamique par les locataires. Les clients (fournisseurs SaaS) de cette plateforme créent les applications et les fournissent à

leurs locataires comme des modèles (templates) à configurer. Certaines configurations peuvent conduire à des applications entièrement différentes. La plateforme dbFlex est mutualisée, c.à.d. la même instance de la plateforme est utilisée par toutes les applications, mais chaque application dispose d'une base de données physiquement séparée. L'accès externe aux applications se fait via des Web services spécifiques et leurs données peuvent être importées et exportées à la demande. Tous les outils de développement sont hébergés par ForeSoft et sont accessibles par l'intermédiaire d'un simple navigateur Web. Actuellement, il y a près de 100 applications disponibles sur dbFLEX, développées par plus de 25 entreprises fournisseurs de SaaS, et supportent plus de 4000 locataires.

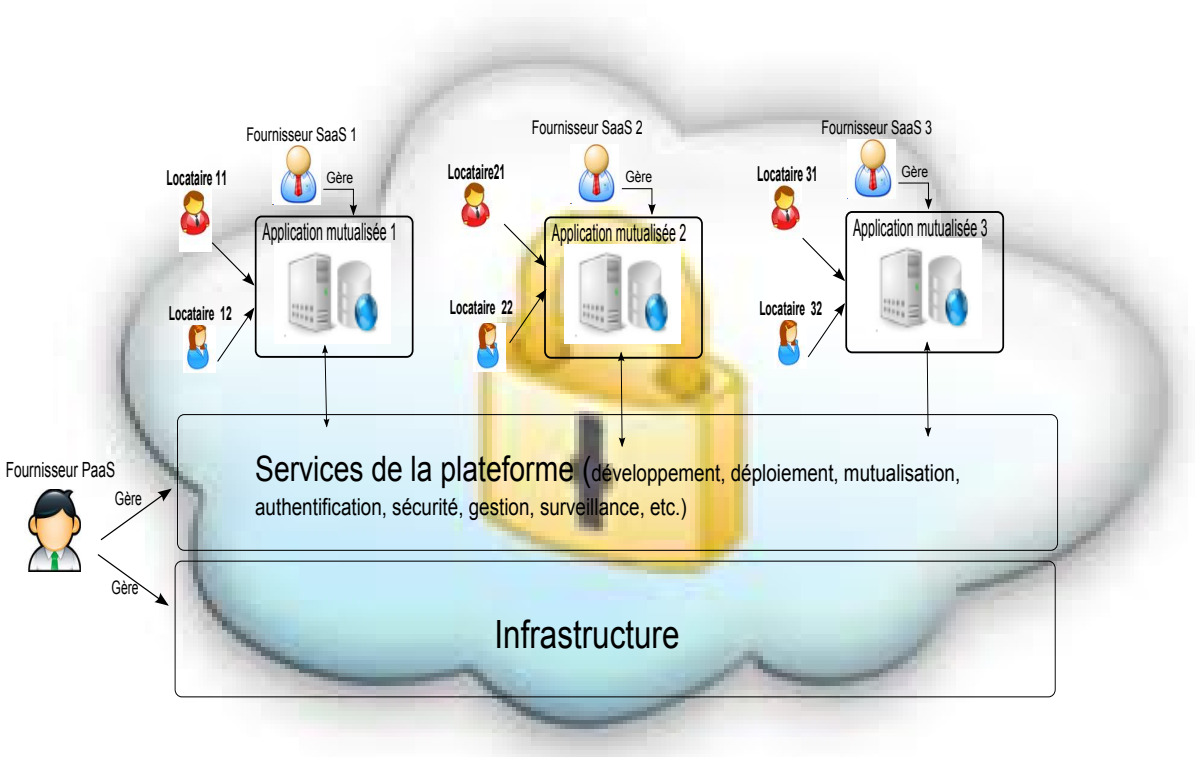


Figure 7.5 – Mutualisation de SaaS à travers un service PaaS classique

Évaluation : *Google App Engine* et *ForeSoft* (ainsi que *Windows Azure* et *Force.com*, voir section 5.3) sont les fournisseurs de service PaaS les plus matures au niveau mondial. Ils proposent des services de développement et de déploiement, ainsi que des services dédiés à la gestion de la mutualisation de SaaS. Cependant, les inconvénients de leur modèle de fourniture proviennent de la forte dépendance qui se crée entre la plateforme et les applications développées au-dessus. Ceci conduit à un problème de « *verrouillage* » avec un fournisseur spécifique (*vendor lock-in*) [ALPW10] (voir figure 7.5). En effet, les applications développées et déployées sur une de ces plateformes dépendent fortement de leur services du fait qu'il est difficile de les déplacer vers une autre plateforme du cloud offrant par exemple des services similaires à moindre coût ou de meilleur qualité. Le coût de ce déplacement peut se révéler prohibitif à cause des efforts d'adaptation et de réingenierie nécessaires [**?**]. Ceci engage un temps de travail considérable durant lequel les fournisseurs des applications clientes seront deux fois facturés (par l'ancienne plateforme aussi

que par la nouvelle). Ce double coût de déplacement doit être pris en considération par ces clients pour qu'ils ne soient pas enfermés dans une plateforme *moins qu'idéale*. Étant donné que le cloud, à la base, est un modèle économique de consommation des ressource informatiques, les clients sont généralement vulnérables à la hausse des prix, et ne seront pas en mesure de se déplacer librement vers de nouvelles et meilleures options quand elles seront disponibles. De ce fait, une deuxième alternative de fourniture de plateformes sous forme de produits est apparue. Cette alternative consiste à installer la plateforme entière sur les machines de développements des clients (fournisseurs de SaaS), qui sera ensuite déployée (avec les applications) sur l'infrastructure de leur choix, et parfois ouvertes à la modification et l'extension pour supporter les besoins spécifiques. Dans la suite, nous allons détailler deux exemples de plateformes adoptant cette alternative.

Mutualisation de SaaS à travers une plateforme fournie sous forme d'un produit : selon ce mode, la plateforme est installée dans l'environnement de développement de chaque client. Comme exemples, nous allons nous concentrer sur les plateformes *Apprenda* [**?**] et *Corenttech* [**?**].

- *Apprenda :* est une entreprise qui a été crée en 2005 avec l'intention initiale de développer des applications SaaS. À mi-chemin du développement, cette entreprise a tourné son attention vers la fourniture d'une plateforme de développement sous forme d'un produit, celle qui a été livrée en 2008 sous le nom *Apprenda SaaSGrid*. Cette plateforme est entièrement financée par l'entreprise et est dédiée exclusivement pour l'environnement de développement .NET. Son objectif principal est de permettre le développement des applications SaaS mutualisées avec peu d'efforts [**?**]. Son rôle principal consiste à créer une couche fonctionnelle de mutualisation entre l'application et son environnement. Elle intercepte et surveille toutes les interactions de l'application avec la base de données et les services qui la composent, et redirige ensuite toutes ces interactions vers une destination spécifique où elle ajoute l'isolement des fonctionnalités, des données et de performance entre les locataires. Ainsi, *Apprenda SaaSGrid* réagit comme une couche de mutualisation dans un environnement de développement à locataire unique. Cela est accompli par la superposition de cette plateforme entre l'application et son environnement d'exécution. L'adaptation aux besoins des locataires se fait par l'exploitation des capacités offertes par l'environnement de développement .NET et ses langages supportées (ex : VB, C#, J#, etc.). SaasGrid ajoute aussi une interface de programmation (API) dédiée pour le suivi de l'utilisation des ressources de l'application à travers la surveillance des interactions de locataires, et applique une politique de facturation personnalisable en fonction du modèle économique de l'application cliente.
- *Corenttech :* est une entreprise qui a été crée entre 2000 et 2002 par des exécutifs ayant de larges connaissances dans le Web, avec l'intention de construire une plateforme avancée qui aide à la migration vers le modèle SaaS. L'activité anticipée de l'entreprise a été consacrée à des projets de conseil en utilisant leur plateforme *SaaS-Suite*, qui est devenue disponible en 2008. Un peu plus tard, cette plateforme a évolué vers une nouvelle capacité de transformer des applications SaaS existantes en des applications mutualisées, et la fourniture des fonctionnalités de gestion du concept de locataire pour soutenir ces applications une fois transformées. Cette plateforme offre également un modèle d'application (*application template*) prédéfini y compris des règles et des interfaces graphiques préétablies dans le but de réduire l'effort de programmation d'application et de son adaptation selon les besoins des locataires. L'architecture interne de la plateforme est orientée service et conçue pour l'extensibilité par l'ajout des modules de développeurs tiers sous forme des *plug-ins*. La plateforme est codée en Java et est uniquement utilisée pour la transformation des applications construites dans les environnements suivants :

– *Système d'exploitation :* RedHat Enterprise Linux 5, Windows Server 2008 R2, SuSe Linux Enterprise Server10.
– *Serveur d'Application :* Tomcat 6.0.32, WebSphere Version7.0.0.7 Application Server
– *Base de données :* Oracle Database10g Release2, MySQL Community Server 5.1, DB2 version 9.7, MSSQL 2008, Postgre 9.0.

La clé du succès de cette plateforme est la manière efficace et transparente dont elle propose pour la ré-ingénierie d'une base de données existante en une nouvelle base de données prête à accueillir plusieurs locataires. Cela est réalisé par l'insertion, dans l'application existante, d'une couche additionnelle appelée « *Multi-tenant server (MTS)* » [tMT10] responsable de cette ré-ingénierie. Celle-ci intercepte toutes les requêtes d'accès aux données et supporte les trois modèles de séparation de données identifiés : bases séparées, schémas séparés et enregistrements séparés.

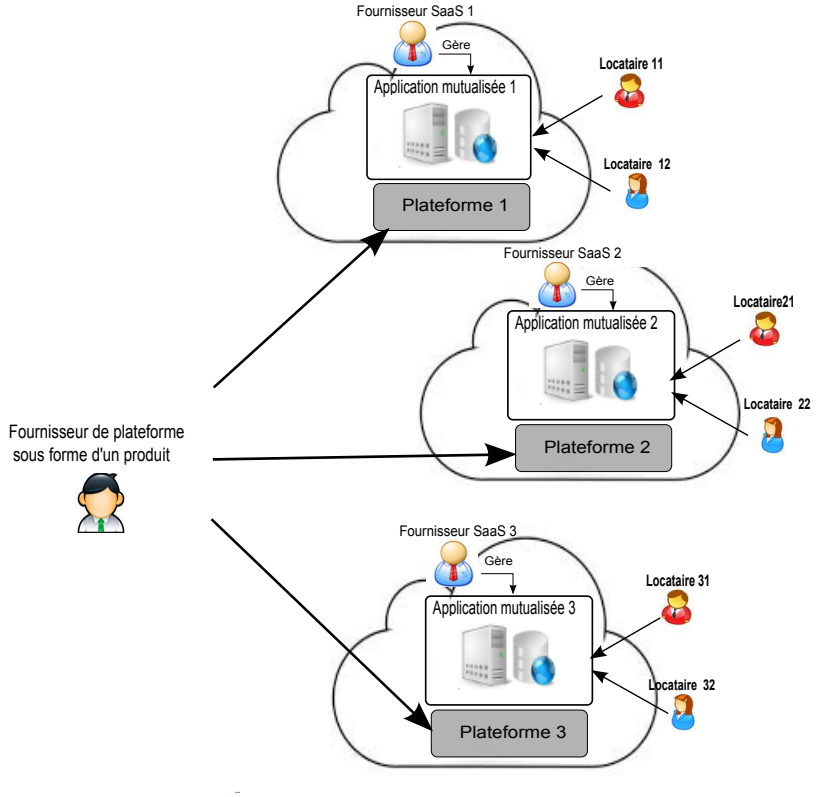


Figure 7.6 – Mutualisation de SaaS à travers une plateforme fournie sous forme d'un produit

Evaluation : *Apprenda* et *Corent* (ainsi que beaucoup d'autres comme *SaaSTenant* [**?**], *SoftServe* [**?**], *Techcello* [**?**], *LongJump* [**?**], etc.) sont des plateformes fournies sous forme des produits à installer localement dans l'environnement de développement des applications clientes et proposent des fonctionnalités pour gérer la mutualisation de ces applications. Ce type de plateformes a connu un réel succès puisque les clients (fournisseurs de SaaS) bénéficient dans ce cas d'une plateforme sous leur contrôle, déployable (avec l'application) sur l'infrastructure de leur choix,

et parfois modifiable et extensible pour supporter leurs besoins spécifiques. Cela libère les clients du fait d'être toujours limités à ce que le fournisseur de PaaS leur propose tel que c'est le cas dans l'alternative précédente. Cependant, et du point de vue des fournisseurs de ces plateformes, ce modèle de fourniture est difficile à maintenir. En effet, la plateforme en elle-même n'est pas mutualisée puisque chaque client a sa propre installation (voir figure 7.6). Cela, à notre sens, rend très difficile le passage à une grande échelle et complique la livraison de nouvelles fonctionnalités des plateformes sus-mentionnées. En plus, celles-ci ne proposent pas un modèle de paiement selon l'usage, ce qui contredit le principe du modèle économique du cloud. Ces plateformes sont commercialisées sous forme de licences qui sont relativement coûteuses, et mobilisent une trésorerie importante au départ.

À travers la plateforme MaaS, nous avons proposé une troisième alternative de gestion de la mutualisation de SaaS. Celle-ci diffère des alternatives adoptées par les plateformes existantes fonctionnellement et économiquement. Sur le plan fonctionnel, la plateforme proposée est faiblement couplée avec les applications clientes et propose des services dédiés uniquement à la gestion de la mutualisation sans imposer un modèle de développement ou une technologie d'application spécifique. Sur le plan économique, la plateforme MaaS propose un modèle économique innovant où les fournisseurs applications clientes payent son utilisation uniquement en cas de succès, c.à.d. lorsqu'ils atteignent un certain nombre de locataires à travers les services de gestion proposés (VaaS, DIaaS, sécurité, administration). L'utilisation de la plateforme MaaS est gratuite pendant toute la période du développement ou de ré-ingénierie des applications. Nous proposons pour cela le modèle de facturation *« Pay-on-Success »* qui diffère de celui dénommé *« Pay-per-Use »* actuellement proposé par les services PaaS, qui consiste à facturer leur utilisation d'une manière périodique et fixe, indépendamment du volume d'utilisation des applications clientes et ainsi de leur succès.

7.5 Conclusion

Pour aboutir à notre approche générale de gérer la mutualisation de SaaS d'une manière complètement externalisée, l'externalisation de la gestion de l'isolation de données de locataires est ainsi nécessaire. Pour cela, nous avons présenté dans ce chapitre une approche qui permet cette externalisation à travers la proposition d'un système d'isolation de données sous forme d'un service (DIaaS : Data Isolation as a Service). Ce dernier permet la mise en place de l'isolation de données requise sans introduire de changements majeurs sur les architectures des applications existantes. Ceci a pu être réalisé grâce à l'architecture du système DIaaS que nous avons conçue et développée et qui a pour rôle principale de simuler aux développeurs un environnement d'accès aux données à locataire unique. Cette architecture propose l'installation d'une base de données abstraite contenant un schéma de données équivalent à celui de la base de données réelle pour créer une couche d'abstraction qui la modélise. Ainsi, les requêtes d'accès aux données seront facilement redirigées vers la base de données abstraite où elles seront interceptées et dynamiquement transformées pour assurer l'isolation stricte de données de locataires d'une manière externalisée et quasi-transparente. Nous avons ensuite présenté la plateforme MaaS qui regroupe les systèmes VaaS et DIaaS ainsi que d'autres services liés à l'administration et la sécurité des applications mutualisées pour offrir une solution complète de gestion de ce type d'applications par externalisation. La plateforme en elle-même est mutualisée entre plusieurs applications et faiblement couplée avec elles. Pour bien situer cette plateforme par rapport aux plateformes existantes fournissant de services de gestion similaires, nous avons analysé les caractéristiques d'un échantillon de plateformes parmi celles

assez nombreuses présentées actuellement sur le marché. L'objectif de ce travail est de clarifier leurs différences avec notre approche, pour démontrer ses avantages aux niveaux fonctionnel et économique.

PARTIE IV

Conclusion et travaux futurs

Conclusion

Le Cloud Computing a récemment émergé dans la communauté d'informatique distribuée comme un nouveau modèle d'approvisionnement qui profite de l'avancée technologique du Web. Ce paradigme est actuellement soutenu par une communauté importante et très dynamique qui s'exprime à travers de nombreuses conférences, journaux ou ateliers dédiés. Il consiste à externaliser vers des centres d'hébergements maintenus par des fournisseurs tiers le développement, la maintenance et l'évolution des ressources informatiques matérielles et logicielles nécessaires pour la mise en œuvre d'une application. L'externalisation peut concerner l'infrastructure d'une telle application suivant le modèle IaaS, sa plateforme de développement suivant le modèle PaaS et l'application elle-même suivant le modèle SaaS. Ce dernier a particulièrement rencontré un vif succès auprès des clients (utilisateurs finaux) grâce à son principal avantage de réduire les investissements initiaux sur l'acquisition de nouveaux logiciels et leur facturation selon l'usage réel.

Le modèle économique de SaaS a ensuite évolué vers une nouvelle approche basée sur l'exploitation des économies d'échelle en offrant en même temps une unique instance d'application à plusieurs clients dénommés locataires, suivant le principe de mutualisation. L'objectif de ce principe à un niveau applicatif est de réduire les coûts opérationnels du service proposé et de capitaliser sur l'expérience cumulée à travers son partage. Cependant, sa mise en œuvre nécessite de relever un certain nombre de défis liés à sa structure organisationnelle, au sein de laquelle chaque locataire doit avoir l'impression d'utiliser une application qui lui est pleinement dédiée. Cela implique une gestion dynamique de la variabilité des besoins de ces locataires et une isolation stricte de leurs données.

Vu la complexité de conception et de développement entraînés par la mutualisation de SaaS, nos travaux de recherche et de développement ont été orienté vers une approche de gestion par externalisation. Cela signifie qu'une autre organisation de celle de fournisseur de SaaS est responsable de fournir des solutions de gestion de la mutualisation à travers une plateforme spécifique. L'analyse effectuée sur les plateformes existantes fournissant des solutions similaires de gestion a dégagé deux approches principales. La première approche consiste à fournir la gestion de la mutualisation de SaaS à travers les services PaaS classiques qui sont largement utilisés et acceptés (Google App Engine, Windows Azure, Force.com, ForeSoft, etc.), et ce, comme étant une partie intégrante de leurs services proposés. En revanche, la deuxième alternative consiste à fournir la plateforme de gestion de la mutualisation sous forme d'un produit à installer dans l'environnement de développement de chaque application cliente (Apprenda, Corent, Techcello, TenantSaaS, etc.). Malgré que chacune des approches présentent des avantages dans des situations particulières, elles présentent toutes les deux des inconvénients évidents qui leur rendent des solutions de gestion inefficaces (*vendor lock-in*, acheter en licences, technologie de développement imposée, etc.). Dans cette thèse, nous avons essentiellement contribué à la conception de plateformes de gestion de la mutualisation de SaaS, en proposant une troisième alternative. La plateforme de gestion proposée (MaaS) implémente un ensemble de services nécessaires pour relever les défis de la mutualisation, sans introduire des changements majeurs aux architectures des applications clientes. La plateforme proposée est mutualisée entre plusieurs applications et faiblement couplée avec elles. Pour arriver à ces résultats, nous avons contribué sur deux axes :

La **première contribution** consiste à concevoir un système de gestion de la variabilité sous forme d'un service (VaaS : Variability as a Service) permettant aux fournisseurs des applications SaaS mutualisées d'externaliser la gestion de la variabilité, dans ses aspects de modélisation et de résolution, vers

un fournisseur tiers. Dans cette optique, nous avons présenté un processus d'externalisation de gestion de la variabilité. Ce processus consiste à modéliser dans un premier temps la variabilité de l'application conformément à un méta-modèle que nous avons conçu. Ce méta-modèle est introduit et formalisé pour couvrir les limites des techniques de modélisation existantes (OVM, FODA, FORME, etc.) établies dans le domaine de lignes de produits logiciels dans le contexte de la spécificité du modèle SaaS mutualisé. Le processus d'externalisation propose ensuite d'exposer directement le modèle de variabilité de l'application (instance du méta-modèle) aux locataires pour qu'ils effectuent eux-mêmes le binding de variabilité. Ceci est réalisé par les locataires en choisissant les variantes souhaitées et en configurant unilatéralement leurs espaces d'utilisation. Un outil de binding spécifique a été conçu et implémenté pour assurer une configuration correcte et complète de l'application, tout en considérant la sémantique opérationnelle du méta-modèle. L'externalisation de la variabilité est finalement accomplie en proposant des capacités de résolution de la variabilité dans l'application cliente en temps réel. Ceci passe à travers l'envoie de demandes de résolution par cette application au système VaaS, dans le but que ces demandes soient traitées en fonction du modèle de variabilité de l'application et les configurations de ses locataires et en retournant les résultats de résolution appropriés.

La **deuxième contribution** consiste à fournir un système d'isolation de données de locataires sous forme d'un service (DIaaS : Data Isolation as a Service) permettant aux fournisseurs des applications SaaS mutualisées d'externaliser la gestion de l'isolation de données à un fournisseur tiers. Cette externalisation couvre les différents aspects du défi d'isolation notamment la séparation de données de locataires, la sécurisation d'accès à leurs données et la personnalisation du schéma de la base de données partagée. D'une façon similaire à l'externalisation de la gestion de la variabilité, cette contribution propose une externalisation de la gestion de l'isolation de données à travers un processus d'externalisation bien défini. Ce processus définit un ensemble des étapes à suivre pour configurer le système DIaaS selon la stratégie d'isolation de données souhaitée par les fournisseur de SaaS dans un premier temps, et d'appliquer cette isolation lors de l'utilisation de l'application par ses locataires dans un second temps. La clé de succès du système DIaaS est l'architecture qu'il propose pour isoler les données de locataires de manière quasi-transparente aux développeurs, à travers la mise en place d'une base de données abstraite contenant un schéma de données équivalent à celui de la base de données réelle de l'application. Ceci permet de rediriger les requêtes d'accès aux données de l'application vers cette base abstraite et d'injecter ensuite des filtres spécifiques pour assurer la sécurisation d'accès aux données et la récupération de données personnalisées.

En combinant les capacités de gestion proposées par les systèmes VaaS et DIaaS ainsi que d'autres services liés à la sécurité et à l'administration des applications SaaS mutualisées, nous avons pu fournir une plateforme dédiée à la gestion de ce type d'applications par externalisation. Cette plateforme que nous avons dénommé MaaS (Multitenancy as a Service) est actuellement utilisée pour supporter la mutualisation des applications SaaS développées au sein de la société BITASOFT, pour laquelle les travaux de recherches présentés dans ce manuscrit ont été destinés. La prochaine étape que nous nous préparons à s'affranchir concerne la fourniture des capacités de cette plateforme pour d'autres applications SaaS développées par d'autres sociétés souhaitant adopter le principe de mutualisation. Toutefois, et pour atteindre ce but, des améliorations s'imposent dans le cadre de nos futurs travaux. Nous revenons en détails sur ces améliorations dans le chapitre suivant.

Travaux Futurs

Les perspectives de travaux futurs sont en rapport direct avec les limitations actuelles de notre approche de gestion de la mutualisation de SaaS par externalisation, notamment face à la fourniture des capacités de la plateforme MaaS à d'autres applications situées en dehors du périmètre de la société. Ces travaux futurs se résument en trois points :

Améliorer la performance et le temps de réponse de la plateforme MaaS : nous avons évoqué dans la section 7.3 que la plateforme MaaS est mutualisée entre plusieurs applications. Actuellement, celle-ci est uniquement utilisée à l'échelle de la société BITASOFT pour supporter la mutualisation de ses applications SaaS. Cette plateforme est hébergée dans la même infrastructure informatique que les applications de la société et nous ne rencontrons pas, pour l'instant, de problèmes de performance sérieux. Toutefois, nous sommes conscient que le passage à une plus grande échelle et la fourniture des capacités de cette plateforme à d'autres applications SaaS appartenant à d'autres sociétés risque de dégrader la performance et faire augmenter le temps de réponse. Pour cette raison, nous sommes entrain d'analyser deux approches différentes pour permettre le passage à l'échelle souhaité. Ces approches consistent principalement à :

- *Installer chaque service de la plateforme sur un serveur séparé :* cette approche préserve le couplage faible existant entre la plateforme MaaS et les applications clientes d'un côté, et entre les services de cette plateforme (VaaS, DIaaS, sécurité, administration) d'un autre. Étant donné que le but consiste d'installer et d'exécuter chaque service sur un serveur séparé. Ces services passent ainsi à l'échelle séparément en fonction de leur volume d'utilisation.
- *Installer plusieurs instances de la plateforme :* cette approche consiste à passer à l'échelle d'une façon horizontale (scal out) à travers la duplication de la plateforme MaaS sur plusieurs serveurs. Un équilibreur de charge (*Load Balancer*) sera ainsi mis en place pour surveiller en permanence la charge de travail de chaque serveur et distribuer toute nouvelle demande d'une application au serveur le moins chargé. Cette approche est équivalente à celle proposée par le niveau 4 de maturité de SaaS (voir section 3.2.2) et exige une synchronisation en permanence entre les différentes instances de la plateforme en cours d'exécution.

Nous allons prochainement tester la performance et le temps de réponse de la plateforme dans chacune des approches de passage à l'échelle pour identifier celle qui assure le niveau de performance requis.

Supporter le modèle d'architecture SCA : dans la continuité de nos recherches pour fournir l'accès à la plateforme MaaS à un nombre important d'applications, le support d'autres modèles d'architecture que celui de l'AOS (uniquement supporté pour le moment) devient une obligation. Ainsi, nous essayerons de cibler à court terme le modèle d'architecture SCA (Service Component Architecture) [**?**], en raison de sa similarité avec l'AOS et son succès actuel dans la recherche et dans la pratique. En effet, ce modèle d'architecture fournit un ensemble de spécifications qui décrivent un modèle unifié de construction d'applications à base de services d'une manière similaire aux assemblages de composants dans les architectures à base de composants [HK11]. De nombreuses plateformes de développement actuellement disponibles supportent la construction des applications suivant ce modèle d'architecture. Nous citons la plateforme FraSCAti [**?**] soutenue

par l'INRIA qui apporte des améliorations significatives en terme de dynamicité, de configurabilité. L'extension envisagée de la plateforme MaaS pour soutenir la mutualisation des applications SaaS construites suivant ce modèle d'architecture consiste à étendre notre méta-modèle de variabilité pour intégrer certains concepts du méta-modèle de SCA [**?**]. Cette extension permettra à notre méta-modèle de variabilité la capacité de modéliser explicitement la variabilité dans les architectures SCA. Le système VaaS supportera nativement la résolution de cette variabilité à travers le mécanisme de résolution existant.

Évolution du système VaaS : dans le cadre de l'amélioration et de l'extension du système VaaS, nous tenterons de stabiliser ce système avant d'explorer ses limitations suivantes :

- *Autres types d'artéfacts :* dans les étapes de résolution de la variabilité supportées à travers le processus d'externalisation proposé par le système VaaS, nous avons uniquement considéré les artéfacts variables au niveau du code. L'application FIA (voir section 6.2) est un exemple d'illustration de ses artéfacts variables (processus métier, web-services, interface graphique, données, etc.). Cependant, au lieu de soutenir uniquement la variabilité du code, nous essayerons de généraliser l'approche VaaS à d'autres types d'artéfacts tel que les exigences, la conception, le test, etc.
- *VaaS et les applications existantes :* l'objectif est de permettre au système VaaS d'être en mesure de soutenir la variabilité aussi bien pour de nouvelles applications que pour des applications existantes, tels que c'est le cas dans le système DIaaS. Nous devons donc procéder à étudier la réorganisation d'une application existante afin d'injecter la variabilité en termes de modélisation et de résolution.
- *Évolution :* les applications sont en évolution continue. La gestion de cette évolution à travers le système VaaS est notre objectif ultime. Elle peut être vue à travers trois principales préoccupations : (i) l'évolution de la variabilité, (ii) la co-évolution des applications et de leurs modèles de variabilité et (iii) la co-évolution du méta-modèle de variabilité et ses modèles de variabilité instances.

Bibliographie

[ABM00] C. Atkinson, J. Bayer, and D. Muthig. Component-based product line development : the kobra approach. *KLUWER INTERNATIONAL SERIES IN ENGINEERING AND COMPUTER SCIENCE*, pages 289–310, 2000.

[AFG+10] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4) :50–58, 2010.

[AGJ+08] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. Multi-tenant databases for software as a service : schema-mapping techniques. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1195–1206. ACM, 2008.

[ALPW10] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. Racs : a case for cloud storage diversity. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 229–240. ACM, 2010.

[AP07] D. Ardagna and B. Pernici. Adaptive service composition in flexible processes. *Software Engineering, IEEE Transactions on*, 33(6) :369–384, 2007.

[Apa07] *Apache Synapse-The Apache Software Foundation*, 2007. `http://ws.apache.org/synapse/`.

[APG+10] Afkham Azeez, Srinath Perera, Dimuthu Gamage, Ruwan Linton, Prabath Siriwardana, Dimuthu Leelaratne, Sanjiva Weerawarana, and Paul Fremantle. Multi-tenant soa middleware for cloud computing. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 458–465. IEEE, 2010.

[App] *Apprenda Inc. : SaaSGrid Middleware*. `http://apprenda.com/platform/`.

[Ars04] Ali Arsanjani. Service-oriented modeling and architecture. *IBM developer works*, 2004.

[Aul11] S. Aulbach. Schema flexibility and data sharing in multi-tenant databases. 2011.

[AV07] T. Auechaikul and W. Vatanawood. A development of business rules with decision tables for business processes. In *TENCON 2007-2007 IEEE Region 10 Conference*, pages 1–4. IEEE, 2007.

[AZE+07] A. Arsanjani, L.J. Zhang, M. Ellis, A. Allam, and K. Channabasavaiah. S3 : A service-oriented reference architecture. *IT professional*, 9(3) :10–17, 2007.

[Azu10] *Windows Azure*, 2010. `http://www.microsoft.com/azure`.

[BBC+06] S. Bajaj, D. Box, D. Chappell, F. Curbera, G. Daniels, P. Hallam-Baker, M. Hondo, C. Kaler, D. Langworthy, A. Nadalin, et al. Web services policy 1.2-framework (ws-policy). *W3C Member Submission*, 25 :12, 2006.

[BCE+02] T. Bellwood, L. Clément, D. Ehnebuske, A. Hately, M. Hondo, Y.L. Husband, K. Januszewski, S. Lee, B. McKee, J. Munter, et al. Uddi version 3.0. *Published specification, Oasis*, 5 :16–18, 2002.

[BCF+03] S. Boag, D. Chamberlin, M.F. Fernández, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. Xquery 1.0 : An xml query language. *W3C working draft*, 12, 2003.

[BCFC06] Alessandro Bozzon, Sara Comai, Piero Fraternali, and Giovanni Toffetti Carughi. Conceptual modeling and code generation for rich internet applications. In *Proceedings of the 6th international conference on Web engineering*, pages 353–360. ACM, 2006.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5) :164–177, 2003.

[BFG+02] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, J. Obbink, and K. Pohl. Variability issues in software product lines. *Software Product-Family Engineering*, pages 303–338, 2002.

[BFH03] F. Berman, G. Fox, and A.J.G. Hey. *Grid computing : making the global infrastructure a reality*, volume 2. Wiley, 2003.

[BGH+06] J. Bayer, S. Gerard, O. Haugen, J. Mansell, B. Moller-Pedersen, J. Oldevik, P. Tessier, J.P. Thibault, and T. Widen. Consolidated product line variability modeling. 2006.

[BGL+04] F. Bachmann, M. Goedicke, J. Leite, R. Nord, K. Pohl, B. Ramesh, and A. Vilbig. A meta-model for representing variability in product family development. *Software Product-Family Engineering*, pages 66–80, 2004.

[BGP07] L. Baresi, S. Guinea, and L. Pasquale. Self-healing bpel processes with dynamo and the jboss rule engine. In *International workshop on Engineering of software services for pervasive environments : in conjunction with the 6th ESEC/FSE joint meeting*, pages 11–20. ACM, 2007.

[BHL08] P. Buxmann, T. Hess, and S. Lehmann. Software as a service. *Wirtschaftsinformatik*, 50(6) :500–503, 2008.

[BHP03] S. Bühne, G. Halmans, and K. Pohl. Modelling dependencies between variation points in use case diagrams. *REFSQŠ03*, page 43, 2003.

[BHST04] Y. Bontemps, P. Heymans, P.Y. Schobbens, and J.C. Trigaux. Semantics of foda feature diagrams. In *Proceedings SPLC 2004 Workshop on Software Variability Management for Product Derivation–Towards Tool Support*, pages 48–58, 2004.

[Biy11] Cihan Biyikoglu. *Federations : Building Scalable, Elastic, and Multi-tenant Database Solutions with Windows Azure SQL Database*, 2011. `http://social.technet.microsoft.com/wiki/contents/articles/2281.federations-building-scalable-elastic-and-multi-tenant-database-solution.aspx`.

[BLDGS04] Miguel L Bote-Lorenzo, Yannis A Dimitriadis, and Eduardo Gómez-Sánchez. Grid characteristics and uses : a grid definition. In *Grid Computing*, pages 291–298. Springer, 2004.

[BLP05] S. Buhne, K. Lauenroth, and K. Pohl. Modelling requirements variability across product lines. In *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*, pages 41–50. IEEE, 2005.

[Bos00] J. Bosch. *Design and use of software architectures : adopting and evolving a product-line approach*. Addison-Wesley Professional, 2000.

[Bos01] J. Bosch. Software product lines : organizational alternatives. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 91–100. IEEE Computer Society, 2001.

[BOU09] Emmanuel BOUCHER. Software as a service : Quelle est la maturité de ce marché et les possibilités d'utilisation par les entreprises ? Master's thesis, HEC-Paris : Management des Systèmes d'Information et de la Technologie, 2009.

[BP08] S. Beauche and P. Poizat. Automated service composition with adaptive planning. *Service-Oriented Computing–ICSOC 2008*, pages 530–537, 2008.

[BSBG08] N. Bencomo, P. Sawyer, G. Blair, and P. Grace. Dynamically adaptive systems are product lines too : Using model-driven techniques to capture dynamic variability of adaptive systems. In *2nd International Workshop on Dynamic Software Product Lines (DSPL 2008), Limerick, Ireland*, volume 38, page 40, 2008.

[BTW01] H. Boley, S. Tabet, and G. Wagner. Design rationale of ruleml : A markup language for semantic web rules. In *International Semantic Web Working Symposium (SWWS)*, pages 381–402, 2001.

[BYV+09] R. Buyya, C.S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging it platforms : Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems*, 25(6) :599–616, 2009.

[BZ10a] Cor-Paul Bezemer and Andy Zaidman. Challenges of reengineering into multi-tenant saas applications. *The Software Engineering Research Group Technical Reports*, 2010.

[BZ10b] C.P. Bezemer and A. Zaidman. Multi-tenant saas applications : maintenance dream or nightmare ? In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, pages 88–92. ACM, 2010.

[BZP+10] C.P. Bezemer, A. Zaidman, B. Platzbeecker, T. Hurkmans, and A. t Hart. Enabling multi-tenancy : An industrial experience report. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–8. IEEE, 2010.

[CBB07] K.S.M. Chan, J. Bishop, and L. Baresi. Survey and comparison of planning techniques for web services composition. *University of Pretoria Pretoria*, 2007.

[CBCP02] Geoff Coulson, Gordon S Blair, Michael Clarke, and Nikos Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15(2) :109–126, 2002.

[CBP+10] Navraj Chohan, Chris Bunch, Sydney Pang, Chandra Krintz, Nagy Mostafa, Sunil Soman, and Rich Wolski. Appscale : Scalable and open appengine application development and deployment. In *Cloud Computing*, pages 57–70. Springer, 2010.

[CC06] F. Chong and G. Carraro. Architecture strategies for catching the long tail. *MSDN Library, Microsoft Corporation*, pages 9–10, 2006.

[CCW06] Frederick Chong, Gianpaolo Carraro, and Roger Wolter. Multi-tenant data architecture. *MSDN Library, Microsoft Corporation*, 2006.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable : A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2) :4, 2008.

[CDK+02] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the web services web : an introduction to soap, wsdl, and uddi. *Internet Computing, IEEE*, 6(2) :86–93, 2002.

[CFP08] C. Cetina, J. Fons, and V. Pelechano. Applying software product lines to build autonomic pervasive systems. In *Software Product Line Conference, 2008. SPLC'08. 12th International*, pages 117–126. IEEE, 2008.

[CGJ+09] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina. Controlling data in the cloud : outsourcing computation without outsourcing control. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 85–90. ACM, 2009.

[Cis09] *Cisco Systems, Inc. WebEx : Web Conferencing.*, 2009. `http://www.sap.com/sme/solutions/businessmanagement/businessbydesign`.

[Ciu09] E. Ciurana. *Developing with Google App Engine*. Apress Berkely, 2009.

[CJB00] M. Coriat, J. Jourdan, and F. Boisbourdin. The split method. *KLUWER INTERNATIONAL SERIES IN ENGINEERING AND COMPUTER SCIENCE*, pages 147–166, 2000.

[CK85] G.P. Copeland and S.N. Khoshafian. A decomposition storage model. In *ACM SIGMOD Record*, volume 14, pages 268–279. ACM, 1985.

[CK07] S.H. Chang and S.D. Kim. A variability modeling method for adaptable services in service-oriented computing. In *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 261–268. Ieee, 2007.

[CLS+05] F. Curbera, F. Leymann, T. Storey, D. Ferguson, and S. Weerawarana. *Web services platform architecture : SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more*. Prentice Hall PTR, 2005.

[CM07] A. Charfi and M. Mezini. Ao4bpel : An aspect-oriented extension to bpel. *World Wide Web*, 10(3) :309–344, 2007.

[CMRW07] R. Chinnici, J.J. Moreau, A. Ryman, and S. Weerawarana. Web services description language (wsdl) version 2.0 part 1 : Core language. *W3C Recommendation*, 26, 2007.

[Cor] *Corent Technology*. `http://www.corenttech.com/`.

[CPTC08] C. Cetina, V. Pelechano, P. Trinidad, and A.R. Cortes. An architectural discussion on dspl. In *Proceedings of the 12th International Software Product Line Conference (SPLC 2008)*, pages 59–68, 2008.

[Cro06] D. Crockford. The application/json media type for javascript object notation (json). 2006.

[CS+01] W.L. Currie, P. Seltsikas, et al. Exploring the supply-side of it outsourcing : evaluating the emerging role of application service providers. *European Journal of Information Systems*, 10(3) :123–134, 2001.

[CT07] L. Console and W.S.D. Team. Ws-diamond : An approach to web services-diagnosability, monitoring and diagnosis. In *International e-Challenges Conference, The Hague (October 2007)*, 2007.

[CWZ10] Hong Cai, Ning Wang, and Ming Jun Zhou. A transparent approach of enabling saas multi-tenancy in the cloud. In *Services (SERVICES-1), 2010 6th World Congress on*, pages 40–47. IEEE, 2010.

[DBET11] Makogon David, Nene Bhushan, Keresteci Ercenk, and Swanson Trent. *The Cloud Ninja - SQL Azure Federations Project*, 2011. `http://shard.codeplex.com/`.

[DBF] *dbFlex Platform*. `http://www.dbflex.net/`.

[DMFM10] T. Dinkelaker, R. Mitschke, K. Fetzer, and M. Mezini. A dynamic software product line approach using aspect models at runtime. In *5th Domain-Specific Aspect Languages Workshop*, 2010.

[DS05] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1(1) :1–30, 2005.

[DT10] U.A.T.M. DER TECHNISCHEN. Techniques for application evolution in multi-tenant databases. 2010.

[DWC10] Tharam Dillon, Chen Wu, and Elizabeth Chang. Cloud computing : issues and challenges. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 27–33. Ieee, 2010.

[DWY10] Jia Du, Hao-yu Wen, and Zhao-jun Yang. Research on data layer structure of multi-tenant e-commerce system. In *Industrial Engineering and Engineering Management (IE&EM), 2010 IEEE 17Th International Conference on*, pages 362–365. IEEE, 2010.

[EC209] *Amazon Elastic Computing Cloud*, 2009. `http://aws.amazon.com/ec2`.

[Eff10] *EffiProz-A Pure C# Database*, 2010. `http://effiproz.codeplex.com/`.

[ELK+06] Christian Emig, Kim Langer, Karsten Krutz, Stefan Link, Christof Momm, and Sebastian Abeck. The soaŠs layers. *White paper, Cooperation & Managenient, Universitat Karlsruhe. Alemania*, 2006.

[EWA06] Christian Emig, Jochen Weisser, and Sebastian Abeck. Development of soa-based software systems-an evolutionary programming approach. In *Telecommunications, 2006. AICT-ICIW'06. International Conference on Internet and Web Applications and Services/Advanced International Conference on*, pages 182–182. IEEE, 2006.

[FDFI09] Franclin S Foping, Ioannis M Dokas, John Feehan, and Syed Imran. A new hybrid schema-sharing technique for multitenant applications. In *Digital Information Management, 2009. ICDIM 2009. Fourth International Conference on*, pages 1–6. IEEE, 2009.

[FDFI10] F.S. Foping, I.M. Dokas, J. Feehan, and S. Imran. An improved schema-sharing technique for a software as a service application to enhance drinking water safety. *Journal of Information Security Research*, 1(1) :1–10, 2010.

[Feu10] George Feuerlicht. Next generation soa : Can soa survive cloud computing ? pages 19–29, 2010.

[FG+09] A. Fox, R. Griffith, et al. Above the clouds : A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Tech. Rep. UCB/EECS*, 28, 2009.

[FK99] D. Florescu and D. Kossmann. Storing and querying xml data using an rdmbs. *IEEE data engineering bulletin*, 22(3) :27–34, 1999.

[FK09] David F Ferraiolo and D Richard Kuhn. Role-based access controls. *arXiv preprint arXiv :0903.2171*, 2009.

[FMF05] T. Friese, J. Müller, and B. Freisleben. Self-healing execution of business processes based on a peer-to-peer service architecture. *Systems Aspects in Organic and Pervasive Computing-ARCS 2005*, pages 108–123, 2005.

[For] *ForeSoft : Software On Demand*. `http://www.foresoft.net/`.

[Fos00] Ian Foster. Internet computing and the emerging grid. *nature web matters*, 7, 2000.

[Fos03] I. Foster. The grid : A new infrastructure for 21st century science. *Grid Computing : Making the Global Infrastructure a Reality*, pages 51–63, 2003.

[FP10] S. Frischbier and I. Petrov. Aspects of data-intensive cloud computing. *From active data management to event-based systems and more*, pages 57–77, 2010.

[Fra08] Gael Fraiteur. User-friendly aspects with compile-time imperative semantics in .net : an overview of postsharp. 2008.

[FRA12] Mass. FRAMINGHAM. Idc forecasts public cloud services spending will approach $100 billion in 2016, generating 41% of growth in five key information technology categories. 11 Sep 2012.

[Fri09] Grant Fritchey. Sql server execution plans. 2009.

[FS04] Keita Fujii and Tatsuya Suda. Dynamic service composition using semantic information. In *Proceedings of the 2nd international conference on Service oriented computing*, pages 39–48. ACM, 2004.

[FT02] R.T. Fielding and R.N. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2) :115–150, 2002.

[FZRL08] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE'08*, pages 1–10. Ieee, 2008.

[GAp09] *Google Apps for Business*, 2009. `http://www.google.com/Apps`.

[GAS+11] Bo Gao, Wen Hao An, Xi Sun, Zhi Hu Wang, Liya Fan, Chang Jie Guo, and Wei Sun. A non-intrusive multi-tenant database software for large scale saas application. In *e-Business Engineering (ICEBE), 2011 IEEE 8th International Conference on*, pages 324–328. IEEE, 2011.

[Gee09] J. Geelan. Twenty one experts define cloud computing. *Cloud Computing Journal*, 4 :1–5, 2009.

[GER08] E. Gjørven, F. Eliassen, and R. Rouvoy. Experiences from developing a component technology agnostic adaptation framework. *Component-Based Software Engineering*, pages 230–245, 2008.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[Gil09] P. Gillet. *Virtualisation des systèmes d'information avec VMware : architecture, projet, sécurité et retours d'expérience*. Editions ENI, 2009.

[GMPT01] Christos K Georgiadis, Ioannis Mavridis, George Pangalos, and Roshan K Thomas. Flexible team-based access control using contexts. In *Proceedings of the sixth ACM symposium on Access control models and technologies*, pages 21–27. ACM, 2001.

[GNT04] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning : Theory & Practice*. Morgan Kaufmann, 2004.

[GoG09] *Cloud Hosting, Cloud Computing and Hybrid Infrastructure from GoGrid*, 2009. `http://www.gogrid.com`.

[Gom00] H. Gomaa. Object oriented analysis and modeling for families of systems with uml. *Software Reuse : Advances in Software Reusability*, pages 43–84, 2000.

[Gom04] H. Gomaa. *Designing software product lines with UML.* Addison-Wesley Boston, USA ;, 2004.

[GP06] Sam Guckenheimer and Juan J Perez. *Software Engineering with Microsoft Visual Studio Team System (Microsoft. NET Development Series).* Addison-Wesley Professional, 2006.

[GR12] Roopali Goel and Vinay Rishiwal. Cloud computing and service oriented architecture. *International Journal of Recent Technology and Engineering (IJRTE)*, 1(1) :137–139, 2012.

[Gri97] M.L. Griss. Software reuse architecture, process, and organization for business success. In *Computer Systems and Software Engineering, 1997., Proceedings of the Eighth Israeli Conference on*, pages 86–89. IEEE, 1997.

[GSH+07] C.J. Guo, W. Sun, Y. Huang, Z.H. Wang, and B. Gao. A framework for native multi-tenancy application development and management. In *E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007. CEC/EEE 2007. The 9th IEEE International Conference on*, pages 551–558. IEEE, 2007.

[GSK+08] Martin Grund, Matthieu Schapranow, Jens Krueger, Jan Schaffner, and Anja Bog. Shared table access pattern analysis for multi-tenant applications. In *Advanced Management of Information for Globalized Enterprises, 2008. AMIGE 2008. IEEE Symposium on*, pages 1–5. IEEE, 2008.

[GTA11] A. Ghaddar, D. Tamzalit, and A. Assaf. Decoupling variability management in multi-tenant saas applications. In *Service Oriented System Engineering (SOSE), 2011 IEEE 6th International Symposium on*, pages 273–279, 2011.

[GTA12] Ali Ghaddar, Dalila TAMZALIT, and Ali Assaf. Gestion de la variabilité dans les applications saas multi-locataire. In *INFORSID. Congrès*, pages 239–256, Montpellier, France, 2012.

[Hab08] Irfan Habib. Virtualization with kvm. *Linux Journal*, 2008(166) :8, 2008.

[HBR08] Alena Hallerbach, Thomas Bauer, and Manfred Reichert. Managing process variants in the process life cycle. In José Cordeiro and Joaquim Filipe, editors, *ICEIS (3-2)*, pages 154–161, 2008.

[HH00] Martin Hancox and Ray Hackney. It outsourcing : frameworks for conceptualizing practice and perception. *Information Systems Journal*, 10(3) :217–237, 2000.

[HHPS08] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic software product lines. *Computer*, 41(4) :93–95, 2008.

[HIM02] Hakan Hacigumus, Bala Iyer, and Sharad Mehrotra. Providing database as a service. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 29–38. IEEE, 2002.

[HJLZ09] M. Hui, D. Jiang, G. Li, and Y. Zhou. Supporting database applications as a service. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 832–843. IEEE, 2009.

[HK11a] Jung-Soo Han and Gui-Jung Kim. Integration technology of literature contents based on saas. In *Information Science and Applications (ICISA), 2011 International Conference on*, pages 1–5. IEEE, 2011.

[HK11b] A. HOCK-KOON. *Contribution à la compréhension et à la modélisation de la composition et du couplage faible de services dans les architectures orientées services*. PhD thesis, UFR Sciences et Techniques, Université de Nantes, 2011.

[IB07] Srikanth Inaganti and Gopala Krishna Behara. Service identification : Bpm and soa handshake. *BPTrends*, 3 :1–12, 2007.

[JA+07] D. Jacobs, S. Aulbach, et al. Ruminations on multi-tenant databases. *BTW Proceedings*, 103 :514–521, 2007.

[JABK08] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. Atl : A model transformation tool. *Science of Computer Programming*, 72(1) :31–39, 2008.

[JB09] H. Jegadeesan and S. Balasubramaniam. A method to support variability of enterprise services on the cloud. In *Cloud Computing, 2009. CLOUD'09. IEEE International Conference on*, pages 117–124. Springer, 2009.

[JEA+07] D. Jordan, J. Evdemon, A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, et al. Web services business process execution language version 2.0. *OASIS Standard*, 11, 2007.

[JGJ97] I. Jacobson, M. Griss, and P. Jonsson. Software reuse : architecture, process and organization for business success. 1997.

[JHA+08] Rod Johnson, Juergen Hoeller, Alef Arendsen, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Darren Davison, Dmitriy Kopylenko, Mark Pollack, Thierry Templier, et al. The spring framework-reference documentation. *Interface21.(accessed 30.04. 07)*, 2008.

[JHB10] S. Jansen, G.J. Houben, and S. Brinkkemper. Customization realization in multi-tenant web applications : case studies from the library sector. *Web Engineering*, pages 445–459, 2010.

[Jin05] Hai Jin. Chinagrid : Making grid computing a reality. In *Digital Libraries : International Collaboration and Cross-Fertilization*, pages 13–24. Springer, 2005.

[JMF09] Shantenu Jha, Andre Merzky, and Geoffrey Fox. Using clouds to provide grids with higher levels of abstraction and explicit support for usage modes. *Concurrency and Computation : Practice and Experience*, 21(8) :1087–1108, 2009.

[JSGI09] Meiko Jensen, Jörg Schwenk, Nils Gruschka, and Luigi Lo Iacono. On technical security issues in cloud computing. In *Cloud Computing, 2009. CLOUD'09. IEEE International Conference on*, pages 109–116. IEEE, 2009.

[JTSJ07] Nico Janssens, Eddy Truyen, Frans Sanen, and Wouter Joosen. Adding dynamic reconfiguration support to jboss aop. In *Proceedings of the 1st workshop on Middleware-application interaction : in conjunction with Euro-Sys 2007*, pages 1–8. ACM, 2007.

[JWY+10] L. Jin, J. Wu, J. Yin, Y. Li, and S. Deng. Improve service interface adaptation using sub-ontology extraction. In *Services Computing (SCC), 2010 IEEE International Conference on*, pages 170–177. IEEE, 2010.

[Kay07] Michael Kay. Xsl transformations (xslt) version 2.0. *W3C Recommendation*, 23, 2007.

[Kho12] S. Khoshnevis. An approach to variability management in service-oriented product lines. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1483–1486. IEEE, 2012.

[KHSS10] Ali Khajeh-Hosseini, Ian Sommerville, and Ilango Sriram. Research challenges for enterprise cloud computing. *arXiv preprint arXiv :1001.3257*, 2010.

[KJ11] J. Kabbedijk and S. Jansen. Variability in multi-tenant environments : architectural design patterns from industry. *Advances in Conceptual Modeling. Recent Developments and New Directions*, pages 151–160, 2011.

[KKKS11] A. Khan, C. Kästner, V. Köppen, and G. Saake. Service variability patterns. *Advances in Conceptual Modeling. Recent Developments and New Directions*, pages 130–140, 2011.

[KKL+98] Kyo C Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. Form : A feature- ; oriented reuse method with domain- ; specific reference architectures. *Annals of Software Engineering*, 5(1) :143–168, 1998.

[KKS07] S. Kalasapur, M. Kumar, and B.A. Shirazi. Dynamic service composition in pervasive computing. *Parallel and Distributed Systems, IEEE Transactions on*, 18(7) :907–918, 2007.

[KMK12] Rouven Krebs, Christof Momm, and Samuel Konev. Architectural concerns in multi-tenant saas applications. In *Proceedings of the 2nd International Conference on Cloud Computing and Service Science (CLOSERŠ12), SciTePress*, 2012.

[KMY+10] Seungseok Kang, Jaeseok Myung, Jongheum Yeon, Seong-wook Ha, Taehyung Cho, Jiman Chung, and Sang-goo Lee. A general maturity model and reference architecture for saas service. In *Database Systems for Advanced Applications*, pages 337–346. Springer, 2010.

[KNL08] T. Kwok, T. Nguyen, and L. Lam. A software as a service with multi-tenancy support for an electronic contract management application. In *Services Computing, 2008. SCC'08. IEEE International Conference on*, volume 2, pages 179–186. IEEE, 2008.

[Koz10] Heiko Koziolek. Towards an architectural style for multi-tenant software applications. In Gregor Engels, Markus Luckey, and Wilhelm Schäfer, editors, *Software Engineering*, volume 159 of *LNI*, pages 81–92. GI, 2010.

[Koz11] H. Koziolek. The sposad architectural style for multi-tenant software applications. In *Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on*, pages 320–327. IEEE, 2011.

[KPR09] Balachandra Reddy Kandukuri, V Ramakrishna Paturi, and Atanu Rakshit. Cloud security issues. In *Services Computing, 2009. SCC'09. IEEE International Conference on*, pages 517–520. IEEE, 2009.

[KQCM09] D. Kitchin, A. Quark, W. Cook, and J. Misra. The orc programming language. *Formal Techniques for Distributed Systems*, pages 1–25, 2009.

[KRL+00] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhaes, and Roy H Campbell. Monitoring, security, and dynamic configuration with the dynamictao reflective orb. In *Middleware 2000*, pages 121–143. Springer, 2000.

[Kru02] C. Krueger. Variation management for software production lines. *Software Product Lines*, pages 107–108, 2002.

[KS12] Muhammad Naeem Khan and Arsalan Shahid. Object-relational mapping framework to enable multi-tenancy attributes in saas applications. *International Journal of Cloud Computing and Services Science (IJ-CLOSER)*, 2(1) :65–75, 2012.

[KSPBC06] W. Kongdenfha, R. Saint-Paul, B. Benatallah, and F. Casati. An aspect-oriented framework for service adaptation. *Service-Oriented Computing–ICSOC 2006*, pages 15–26, 2006.

[KSSA09] M. Koning, C. Sun, M. Sinnema, and P. Avgeriou. Vxbpel : Supporting variability for web services in bpel. *Information and Software Technology*, 51(2) :258–269, 2009.

[KW93] Karen Ketler and John Walstrom. The outsourcing decision. *International journal of information management*, 13(6) :449–459, 1993.

[Law08a] George Lawton. Developing software online with platform-as-a-service technology. *Computer*, 41(6) :13–15, 2008.

[Law08b] George Lawton. New ways to build rich internet applications. *Computer*, 41(8) :10–12, 2008.

[LGK+11] Soumaya Louhichi, Mohamed Graiet, Mourad Kmimech, Mohamed Tahar Bhiri, Walid Gaaloul, and Eric Cariou. Atl transformation of uml 2.0 for the generation of sca model. In *ICSEA 2011, The Sixth International Conference on Software Engineering Advances*, pages 418–425, 2011.

[LGX09] Meiqun Liu, Kun Gao, and Lifent Xi. Knowledge extracting platform based on web service. In *WSEAS International Conference. Proceedings. Mathematics and Computers in Science and Engineering*, number 3. World Scientific and Engineering Academy and Society, 2009.

[LHX12] Yang Dan Li Heng and Zhang Xiaohong. A new meta-data driven data-sharing storage model for saas. *IJCSI International Journal of Computer Science Issues, Vol. 9, Issue 6, No 1, November 2012 ISSN (Online) : 1694-0814*, 2012.

[LJ06] Bert Lagaisse and Wouter Joosen. True and transparent distributed composition of aspect-components. In *Middleware 2006*, pages 42–61. Springer, 2006.

[LK10] J. Lee and G. Kotonya. Combining service-orientation with product line engineering. *Software, IEEE*, 27(3) :35–41, 2010.

[LKN+09] Alexander Lenk, Markus Klems, Jens Nimis, Stefan Tai, and Thomas Sandholm. What's inside the cloud ? an architectural map of the cloud landscape. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 23–31. IEEE Computer Society, 2009.

[Lon] *LongJump Platform*. `http://www.longjump.com/`.

[LSGF08] N. Loughran, P. Sánchez, A. Garcia, and L. Fuentes. Language support for managing variability in architectural models. In *Software Composition*, pages 36–51. Springer, 2008.

[LSW09] Shuai Luan, Yuliang Shi, and Haiyang Wang. A mechanism of modeling and verification for saas customization based on tla. *Web Information Systems and Mining*, pages 337–344, 2009.

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

[LZV08] P.A. Laplante, J. Zhang, and J. Voas. What's in a name ? distinguishing between saas and soa. *IT Professional*, 10(3) :46–50, 2008.

[MA02] D. Muthig and C. Atkinson. Model-driven product line architectures. *Software product lines*, pages 79–90, 2002.

[Ma07] D. Ma. The business model of software as a service. In *Services Computing, 2007. SCC 2007. IEEE International Conference on*, pages 701–702. IEEE, 2007.

[Man09] Open Cloud Manifesto. Open cloud manifesto. *Availabe online : www. opencloudmanifesto. org/Open*, 20, 2009.

[Mar09] Oliver Marks. *Cloud Bursts as Coghead Calls It Quits*, 2009. `http://www.zdnet.com/blog/collaboration/cloud-bursts-as-coghead-calls-it-quits/349`.

[MBNJ09] Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jezequel. Taming dynamically adaptive systems using models and aspects. In *Proceedings of the 31st International Conference on Software Engineering*, pages 122–132. IEEE Computer Society, 2009.

[MCA+11] Kun Ma, Zhenxiang Chen, Ajith Abraham, Bo Yang, and Runyuan Sun. A transparent data middleware in support of multi-tenancy. In *Next Generation Web Services Practices (NWeSP), 2011 7th International Conference on*, pages 1–5. IEEE, 2011.

[Men07] Falko Menge. Enterprise service bus. In *Free and open source software conference*, pages 1–6, 2007.

[MG11] P. Mell and T. Grance. The nist definition of cloud computing. *National Institute of Standards and Technology special publication*, 800 :145, 2011.

[MGZ07] F. Montesi, C. Guidi, and G. Zavattaro. Composing services with jolie. In *Web Services, 2007. ECOWS'07. Fifth European Conference on*, pages 13–22. IEEE, 2007.

[MHP+07] A. Metzger, P. Heymans, K. Pohl, P.Y. Schobbens, and G. Saval. Disambiguating the documentation of variability in software product lines : A separation of concerns, formalization and automated analysis. In *Requirements Engineering Conference, 2007. RE'07. 15th IEEE International*, pages 243–253. IEEE, 2007.

[Mie10] R. Mietzner. *A method and implementation to define and provision variable composite applications, and its usage in cloud computing*. PhD thesis, Stuttgart University, 2010.

[MK11] Christof Momm and Rouven Krebs. A qualitative discussion of different approaches for implementing multi-tenant saas offerings. In *Software Engineering (Workshops)*, volume 11, 2011.

[MKBJ08] Brice Morin, Jacques Klein, Olivier Barais, and Jean-Marc Jézéquel. A generic weaver for supporting product lines. In *Proceedings of the 13th international workshop on Early Aspects*, pages 11–18. ACM, 2008.

[ML97] M.H. Meyer and AP Lehnerd. The power of product platforms : Building value and cost leadership. *New York, NY*, 10020 :39, 1997.

[ML08] R. Mietzner and F. Leymann. Generation of bpel customization processes for saas applications from variability descriptors. In *Services Computing, 2008. SCC'08. IEEE International Conference on*, volume 2, pages 359–366. IEEE, 2008.

[MLP08] R. Mietzner, F. Leymann, and M.P. Papazoglou. Defining composite configurable saas application packages using sca, variability descriptors and multi-tenancy patterns. In *Internet and Web Applications and Services, 2008. ICIW'08. Third International Conference on*, pages 156–161. IEEE, 2008.

[MMLP09] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl. Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In

Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems, pages 18–25. IEEE Computer Society, 2009.

[MRD08] O. Moser, F. Rosenberg, and S. Dustdar. Non-intrusive monitoring and service adaptation for ws-bpel. In *Proceedings of the 17th international conference on World Wide Web*, pages 815–824. ACM, 2008.

[MT00] A. Maccari and A.P. Tuovinen. System family architectures : current challenges at nokia. *Software Architectures for Product Families*, pages 107–115, 2000.

[MUTL09] R. Mietzner, T. Unger, R. Titze, and F. Leymann. Combining different multi-tenancy patterns in service-oriented applications. In *Enterprise Distributed Object Computing Conference, 2009. EDOC'09. IEEE International*, pages 131–140. IEEE, 2009.

[NCH11] T. Nguyen, A. Colman, and J. Han. Modeling and managing variability in process-based service compositions. *Service-Oriented Computing*, pages 404–420, 2011.

[NG09] J. Namjoshi and A. Gupte. Service oriented architecture for cloud based travel reservation software as a service. In *Cloud Computing, 2009. CLOUD'09. IEEE International Conference on*, pages 147–150. IEEE, 2009.

[NKV+09] Y Natis, Eric Knipp, Ray Valdes, D Cearley, and Daniel Sholler. Whoś who in application platforms for cloud computing : the cloud specialists. *Gartner Research*, 2009.

[Nor02] L.M. Northrop. Sei's software product line tenets. *Software, IEEE*, 19(4) :32–40, 2002.

[NWG+09] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on*, pages 124–131. IEEE, 2009.

[OAS09] OASIS. *Service component architecture assembly model specification version 1.1*, 2009. `http://www.oasis-opencsa.org/`.

[OGTP09] C Osipov, German Goldszmidt, Mary Taylor, and Indrajit Poddar. Develop and deploy multi-tenant web-delivered solutions using ibm middleware : Part 2 : Approaches for enabling multi-tenancy. *IBM Corp. Website*, 2009.

[Pap03] M.P. Papazoglou. Service-oriented computing : Concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pages 3–12. IEEE, 2003.

[Par09] P. Parys. Xpath evaluation in linear time with polynomial combined complexity. In *Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 55–64. ACM, 2009.

[PBP06] K. Petersen, N. Bramsiepe, and K. Pohl. Applying variability modeling concepts to support decision making for service composition. In *Service-Oriented Computing : Consequences for Engineering Requirements, 2006. SOCCER'06*, pages 1–1. IEEE, 2006.

[PBVDL05] K. Pohl, G. Bockle, and F. Van Der Linden. *Software product line engineering : foundations, principles, and techniques*. Springer-Verlag New York Inc, 2005.

[PDFS01] Renaud Pawlak, Laurence Duchien, Gérard Florin, and Lionel Seinturier. Jac : A flexible solution for aspect-oriented programming in java. In *Metalevel architectures and separation of crosscutting concerns*, pages 1–24. Springer, 2001.

[Pel03] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10) :46–52, 2003.

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

[PR07] B. Pernici and A.M. Rosati. Automatic learning of repair strategies for web services. In *Web Services, 2007. ECOWS'07. Fifth European Conference on*, pages 119–128. IEEE, 2007.

[PVDH07] M.P. Papazoglou and W.J. Van Den Heuvel. Service oriented architectures : approaches, technologies and research issues. *The VLDB journal*, 16(3) :389–415, 2007.

[PY10] P. Poizat and Y. Yan. Adaptive composition of conversational services through graph planning encoding. *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 35–50, 2010.

[PZ03] Eduardo Kessler Piveta and Luiz Carlos Zancanella. Aspect weaving strategies. *J. UCS*, 9(8) :970, 2003.

[PZ06] C. Pahl and Y. Zhu. A semantical framework for the orchestration and choreography of web services. *Electronic Notes in Theoretical Computer Science*, 151(2) :3–18, 2006.

[Rai09] Geoffrey Raines. Cloud computing and soa. *MITRE technical papers, MITRE Corp., Massachusetts, USA*, 2009.

[Rap04] M.A. Rappa. The utility business model and the future of computing services. *IBM Systems Journal*, 43(1) :32–42, 2004.

[RCB08] S. Resnick, R. Crane, and C. Bowen. *Essential windows communication foundation : for. net framework 3.5*. Addison-Wesley Professional, 2008.

[RCL09] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. A taxonomy and survey of cloud computing systems. In *INC, IMS and IDC, 2009. NCM'09. Fifth International Joint Conference on*, pages 44–51. Ieee, 2009.

[RES10] S Ramgovind, Mariki M Eloff, and E Smith. The management of security in cloud computing. In *Information Security for South Africa (ISSA), 2010*, pages 1–7. IEEE, 2010.

[RG11] Tejaswi Redkar and Tony Guidici. *Windows Azure Platform*. Apress, 2011.

[RJ11] Sameer Rajan and Apurva Jairath. Cloud computing : The fifth generation of computing. In *Communication Systems and Network Technologies (CSNT), 2011 International Conference on*, pages 665–667. IEEE, 2011.

[RJKG11] Bhaskar Prasad Rimal, Admela Jukan, Dimitrios Katsaros, and Yves Goeleven. Architectural requirements for cloud computing systems : an enterprise cloud approach. *Journal of Grid Computing*, 9(1) :3–26, 2011.

[RMVG+10] Luis Rodero-Merino, Luis M Vaquero, Victor Gil, Fermín Galán, Javier Fontán, Rubén S Montero, and Ignacio M Llorente. From infrastructure delivery to service management in clouds. *Future Generation Computer Systems*, 26(8) :1226–1240, 2010.

[RW04] J.W. Ross and G. Westerman. Preparing for utility computing : The role of it architecture and relationship management. *IBM systems journal*, 43(1) :5–19, 2004.

[Rys05] M. Rys. Xml and relational database management systems : inside microsoft® sql serverŹ 2005. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 958–962. ACM, 2005.

[Saa] *SaaS-Tenant : Develop Multi Tenant SaaS system*. `http://www.saas-tenant.com`.

[Sal09] *Salesforce Site Officiel*, 2009. https://www.salesforce.com/fr/.

[San03] Ravi Sandhu. Good-enough security. *Internet Computing, IEEE*, 7(1) :66–68, 2003.

[SAP09] *SAP AG. SAP Business ByDesign : The Best of SAP, On Demand.*, 2009. http://www.webex.com.

[SBNH05] J. Siljee, I. Bosloper, J. Nijhuis, and D. Hammer. Dysoa : Making service systems self-adaptive. *Service-Oriented Computing-ICSOC 2005*, pages 255–268, 2005.

[SCA06] CM Saracca, D. Chamberlin, and R. Ahuja. Db2 9 : pure xml–overview and fast start. *IBM Redbooks*, 2006.

[SCG+12] J. Schroeter, S. Cech, S. Götz, C. Wilke, and U. Aßmann. Towards modeling a variable architecture for multi-tenant saas-applications. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, pages 111–120. ACM, 2012.

[SD07] M. Sinnema and S. Deelstra. Classifying variability modeling techniques. *Information and Software Technology*, 49(7) :717–739, 2007.

[SDNB04] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. Covamof : A framework for modeling variability in software product families. *Software Product Lines*, pages 25–27, 2004.

[Sha11] Q. Shao. *Towards Effective and Intelligent Multi-tenancy SaaS*. PhD thesis, ARIZONA STATE UNIVERSITY, 2011.

[SK11] S Subashini and V Kavitha. A survey on security issues in service delivery models of cloud computing. *Journal of Network and Computer Applications*, 34(1) :1–11, 2011.

[Sky08] *Windows Live SkyDrive*, 2008. windows.microsoft.com/fr-FR/skydrive.

[SLLW09] Y. Shi, S. Luan, Q. Li, and H. Wang. A multi-tenant oriented business process customization system. In *New Trends in Information and Service Science, 2009. NISS'09. International Conference on*, pages 319–324. IEEE, 2009.

[SM10] M. Stollberg and M. Muth. Service customization by variability modeling. In *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops*, pages 425–434. Springer, 2010.

[SMF+09] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and J-B Stefani. Reconfigurable sca applications with the frascati platform. In *Services Computing, 2009. SCC'09. IEEE International Conference on*, pages 268–275. IEEE, 2009.

[SMLF09] Borja Sotomayor, Rubén S Montero, Ignacio M Llorente, and Ian Foster. Virtual infrastructure management in private and hybrid clouds. *Internet Computing, IEEE*, 13(5) :14–22, 2009.

[Sof] *SoftServe Platform*. http://www.softserveinc.com/.

[SP06] A. Schnieders and F. Puhlmann. Variability mechanisms in e-business process families. In *9th International Conference on Business Information Systems (BIS 2006)*, volume 85, pages 583–601, 2006.

[SR11] B. Sengupta and A. Roychoudhury. Engineering multi-tenant software-as-a-service systems. In *Proceeding of the 3rd international workshop on Principles of engineering service-oriented systems*, pages 15–21. ACM, 2011.

[SRG+00] Ben Segal, Les Robertson, Fabrizio Gagliardi, Federico Carminati, and G Cern. Grid computing : The european data grid project. In *IEEE Nuclear Science Symposium and Medical Imaging Conference*, volume 1, page 2, 2000.

[SRS+10] C. Sun, R. Rossing, M. Sinnema, P. Bulanov, and M. Aiello. Modeling and managing the variability of web service-based systems. *Journal of Systems and Software*, 83(3) :502–516, 2010.

[SSS00] K. Scribner, K. Scribner, and M.C. Stiver. *Understanding Soap : Simple Object Access Protocol*. Sams, 2000.

[Sul11] Nabil Ahmed Sultan. Reaching for the Ş cloud Ť : How smes can manage. *International Journal of Information Management*, 31(3) :272–278, 2011.

[SVGB05] M. Svahnberg, J. Van Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Software : Practice and Experience*, 35(8) :705–754, 2005.

[SVL01] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing i/o devices on vmware workstation Š s hosted virtual machine monitor. In *Proceedings of the General Track : 2002 USENIX Annual Technical Conference*, pages 1–14, 2001.

[SZG+08] W. Sun, X. Zhang, C.J. Guo, P. Sun, and H. Su. Software as a service : Configuration and customization perspectives. In *Congress on Services Part II, 2008. SERVICES-2. IEEE*, pages 18–25. IEEE, 2008.

[Tao01] L. Tao. Shifting paradigms with the application service provider model. *Computer*, 34(10) :32–39, 2001.

[TBB03] M. Turner, D. Budgen, and P. Brereton. Turning software into a service. *Computer.*, 36(10) :38–44, 2003.

[tBGFZ10] MH ter Beek, S. Gnesi, A. Fantechi, and G. Zavattaro. Modelling variability, evolvability, and adaptability in service computing. *MH*, 2010.

[Tec] *Techcello : Cloud ready Multi-tenant Application Platform*. `www.techcello.com`.

[TJA10] Hassan Takabi, James BD Joshi, and Gail-Joon Ahn. Security and privacy challenges in cloud computing environments. *Security & Privacy, IEEE*, 8(6) :24–31, 2010.

[tMT10] Corent Technology : SOA Approach to Multi-Tenancy. 2010. `http://www-304.ibm.com/partnerworld/gsd/showimage.do?id=30745`.

[TOM08] B. T OGRAPH and Y.R. MORGENS. Cloud computing. *Communications of the ACM*, 51(7), 2008.

[TRCPB07] P. Trinidad, A. Ruiz-Cortés, J. Pena, and D. Benavides. Mapping feature models onto component models to build dynamic software product lines. In *International Workshop on Dynamic Software Product Line, DSPL*, 2007.

[TSPB02] Rajesh K Thiagarajan, Amit K Srivastava, Ashis K Pujari, and Visweswar K Bulusu. Bpml : A process modeling language for dynamic business models. In *Proceedings of the Fourth IEEE International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS'02)*, page 239. IEEE Computer Society, 2002.

[TVJ+01] Eddy Truyen, Bart Vanhaute, Bo Nørregaard Jørgensen, Wouter Joosen, and Pierre Verbaeton. Dynamic and selective combination of extensions in component-based applications. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 233–242. IEEE Computer Society, 2001.

[TZ+10] Asoke K Talukder, Lawrence Zimmerman, et al. Cloud economics : Principles, costs, and benefits. In *Cloud computing*, pages 343–360. Springer, 2010.

[VdL02] Frank Van der Linden. Software product families in europe : the esaps & cafe projects. *Software, IEEE*, 19(4) :41–49, 2002.

[vdML02] T. von der Maßen and H. Lichter. Modeling variability by uml use case diagrams. In *Proceedings of the International Workshop on Requirements Engineering for product lines*, pages 19–25. Citeseer, 2002.

[VGBS01] J. Van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 45–54. IEEE, 2001.

[Vog08] WA Vogels. Head in the cloudsŮthe power of infrastructure as a service. In *First workshop on Cloud Computing and in Applications (CCAŠ08)(October 2008)*, 2008.

[Vou08] Mladen A Vouk. Cloud computingŮissues, research and implementations. In *Information Technology Interfaces, 2008. ITI 2008. 30th International Conference on*, pages 31–40. Ieee, 2008.

[VRMCL08] L.M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds : towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1) :50–55, 2008.

[WB09] C.D. Weissman and S. Bobrowski. The design of the force.com multitenant internet application development platform. In *Proceedings of the 35th SIGMOD international conference on Management of data*, pages 889–896. ACM, 2009.

[WEBYE09] Y Wolfsthal, E Elmroth, M Ben-Yehuda, and W Emmerich. The reservoir model and architecture for open federated cloud computing. 2009.

[WG04] D.L. Webber and H. Gomaa. Modeling variability in software product lines with the variation point model. *Science of Computer Programming*, 53(3) :305–331, 2004.

[WGG+08] Z.H. Wang, C.J. Guo, B. Gao, W. Sun, Z. Zhang, and W.H. An. A study and performance evaluation of the multi-tenant data tier design patterns for service oriented computing. In *e-Business Engineering, 2008. ICEBE'08. IEEE International Conference on*, pages 94–101. IEEE, 2008.

[WTJ11] Stefan Walraven, Eddy Truyen, and Wouter Joosen. A middleware for flexible and cost-efficient multi-tenant applications. In *Proceedings of the 12th International Middleware Conference*, pages 360–379. International Federation for Information Processing, 2011.

[WVLY+10] Lizhe Wang, Gregor Von Laszewski, Andrew Younge, Xi He, Marcel Kunze, Jie Tao, and Cheng Fu. Cloud computing : a perspective study. *New Generation Computing*, 28(2) :137–146, 2010.

[XQL10] Zheng Xuxu, Li Qingzhong, and Kong Lanju. A data storage architecture supporting multi-level customization for saas. In *Web Information Systems and Applications Conference (WISA), 2010 7th*, pages 106–109. IEEE, 2010.

[ZCB10] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing : state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1) :7–18, 2010.

[Zeh10] E. Zehoo. Oracle xml support. *Pro ODP. NET for Oracle Database 11g*, pages 253–285, 2010.

[ZJ05] T. Ziadi and J.M. Jézéquel. Manipulation de lignes de produits logiciels : une approche dirigée par les modèles. *Ingénierie Dirigée par les Modèles (IDMŠ05)*, 2005.

[ZSTC10] Xuesong Zhang, Beijun Shen, Xucheng Tang, and Wei Chen. From isolated tenancy hosted application to multi-tenancy : Toward a systematic migration method for web application. In *Software Engineering and Service Sciences (ICSESS), 2010 IEEE International Conference on*, pages 209–212. IEEE, 2010.

Une contribution à la gestion des applications SaaS mutualisées dans le cloud: approche par externalisation

Ali GHADDAR

Résumé

Le modèle économique du Cloud Computing, plus précisément dans sa couche applicative de services SaaS, a évolué vers une nouvelle approche basée sur l'exploitation des économies d'échelle. Ceci a pu être réalisé en offrant en même temps une unique instance d'application à plusieurs clients dénommés *locataires*, suivant le principe de *mutualisation*. L'objectif principal de ce principe à un niveau applicatif est de réduire les coûts opérationnels du service proposé et de capitaliser sur l'expérience cumulée à travers son partage. Cependant, sa mise en œuvre nécessite de relever un certain nombre de défis liés à sa structure organisationnelle, au sein de laquelle chaque locataire doit avoir l'impression d'utiliser une application qui lui est pleinement dédiée. Cela implique *une gestion dynamique de la variabilité* des besoins de locataires et une *isolation stricte de leurs données*. Dans cette thèse, nous nous intéressons à ce principe de mutualisation et aux principaux défis qui en découlent avant de proposer nos contributions. Celles-ci se résument en trois axes : (i) le premier concerne la spécification d'un méta-modèle de variabilité introduisant de nouveaux concepts de modélisation pour mieux traiter la variabilité et externaliser sa gestion sous forme d'un service. Nous avons pour cela introduit la notion de *VaaS (Variability as a Service)* comme un nouveau membre de la famille des services du Cloud. (ii) Le second axe consiste à étendre la politique de gestion par externalisation, initialement adoptée pour gérer la variabilité, afin de l'appliquer au niveau des données en proposant un *système d'isolation de données sous forme d'un service*. Le principal avantage de ce système est d'isoler les données de locataires d'une manière quasi-transparente aux développeurs, sans introduire de changements majeurs sur les architectures des applications existantes. (iii) Le dernier axe concerne le regroupement des deux premières contributions ainsi que d'autres services liées à l'administration et à la sécurité des applications mutualisées dans une plateforme dédiée, vers une approche globale de gestion de ce type d'applications par externalisation.

Mots-clés : Cloud Computing, SaaS, mutualisation, multi-locataires, externalisaion, architecture orientée service, variabilité, méta-modèle, isolation de données.

Abstract

The business model of cloud computing, especially in its application layer of services SaaS, has evolved into a new approach based on the exploitation of scale economies. This could be done by offering at the same time a unique application instance to several customers called *tenants*, following the *multi-tenancy* principle. The objective of this principle at the application level is to reduce the operating costs of the proposed service and to capitalize on the cumulative experience throught its sharing. However, the implementation of such principle requires addressing a number of challenges realted to its organizational structure, in which each tenant must have the feeling of using a fully dedicated application for his usage. This implies a *dynamic management of variability* in tenants needs and a *strict isolation of their data*. In this dissertation, we focus on the principle of multi-tenancy and the main challenges arising before proposing our contributions. These can be summarized in three axes: (i) the first is the specification of a variability meta-model introducing new modeling concepts to better address the variability and outsource its management as a service. For this, we have introduced the concept of VaaS *(Variability as a Service)* as a new member of the Cloud services family. (ii) The second axis is to extend the management outsourcing policy, initially adopted to manage the variability, in order to apply it at the data level by providing a *data isolation system as a service*. The main advantage of such system is to isolate tenants data in an almost transparent way to developers, without introducing major changes to the architectures of existing applications. (iii) The last axis concerns the combination of the first two contributions and other related services to the administration and security of multi-tenant applications in a dedicated platform, towards a holistic approach of multi-tenancy management by outsourcing.

Keywords: Cloud Computing, SaaS, multi-tenancy, multi-tenant, outsourcing, service oriented architecture; variability; software product line engineering; meta model, data isolation