KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering

Middleware and Methods for Customizable SaaS

Stefan Walraven

Supervisor:
Prof. dr. ir. W. Joosen
Dr. E. Truyen, co-supervisor

Dissertation presented in partial fulfilment of the requirements for the degree of Doctor in Engineering

June 2014

Middleware and Methods for Customizable SaaS

Stefan WALRAVEN

Dissertation presented in partial fulfilment of the requirements for the degree of Doctor in Engineering

June 2014

ISBN 978-94-6018-849-7

D/2014/7515/73

Acknowledgements

The ultimate inspiration is the deadline.
— Nolan Bushnell

This dissertation is the result of long nights of hard work, intense but inspiring discussions and encounters, and the unstoppable combination of deadlines and my will to succeed. However, this would never have been possible without the support of many people to whom I want to express my sincere gratitude.

First of all, I would like to thank my supervisor Wouter Joosen for convincing me and providing me the opportunity to do a PhD, for giving me the freedom to explore my interests, and sometimes also for taming my temper. His advice did not only have an impact on my research, but will remain invaluable throughout the rest of my career.

Secondly, I would like to thank Eddy Truyen, my co-supervisor and direct coach, for our quite successful collaboration. His research vision has lead us to a (back then) new upcoming domain, and actually made us the cloud computing pioneers within DistriNet. He also kept impressing me with his knowledge of related work, which often forced us to rethink and improve our ideas, even hours before a paper deadline. Together, we formed a great team that could move mountains of work in a very short period of time. No better proof for that than the Middleware '11 paper.

Furthermore, I am grateful to the other members of my jury for the interesting discussion and the valuable feedback: Yolande Berbers, Erik Duval, Eric Steegmans, Dimitri Van Landuyt, and Gordon Blair. A special thanks goes to Dimitri for the pleasant and efficient collaboration, and to Gordon for inviting me to Lancaster for a short but inspiring research visit. I would also like to thank Carlo Vandecasteele for chairing the jury and the practical advice.

Needless to say, many thanks go to my long-term office mates Bart, Bert, Maarten, and Wouter. Our office is probably known as the loudest of the building (or even of the campus), because of our daily intense debates on diverse topics. But it is also a very productive and supporting environment with complementary profiles, always willing to give their no-nonsense opinion about your work or to help when you're struggling. A special thanks goes to Bert who introduced me in the domain of adaptive middleware during my master thesis, and who continued to play a major role in the development of my research and writing skills. Likewise, I thank the LantaM taskforce for the critical feedback on my work in progress as well as for the regular fun and entertainment: Adriaan, Ansar, Bart, Bert, Dimitri, Eddy, Fatih, Frans, Jasper, Kristof, Maarten B, Maarten D, Marko, Steven, and Wouter. The influence of this mixed group of middleware developers, software engineers, language designers and system admins had an important impact on the course of my PhD.

I would also like to thank the many (ex-)colleagues at the DistriNet research group for the very enjoyable and creative work environment. In particular, I want to thank my current and former Alma mates Alex, Bert L, Bert VH, Koen, Lieven, Maarten, Marko, Nelis, Lex, Peter, Philippe, Steven, Thomas, Willem, Wouter, and Yves: the daily walk to Alma was often a source of great ideas, or at least I got up-to-date with the latest facts and trivia. I may also not forget the long discussions (mostly till the very end) with the other regular attendees ("plakkers") of the different receptions: Davy, Dries, Jef, Kim and Koen. In addition, I am grateful to the administrative and technical staff of the department for their often overlooked but valuable support. Special thanks to Annick, Esther, Ghita, Katrien, and Marleen, for always helping me with a smile and getting everything arranged. Finally, many thanks to my colleagues and friends in Lancaster for getting me quickly settled during my stay there.

Many thanks to all the "Nerds on a high" climbers as well as the brothers and sisters at Kung Fu Leuven for the necessary distraction during several hours per week and for the opportunity to vent my frustrations. Special thanks to Bart, Boris, Jan, Jem, Jeroen, José, master Lo, Mika, Nele, Nelis, Peter, Radu, Sam, Steven, Vassili, Ward, Wouter DB, Wouter V, and Zhaoyin, for improving my climbing and fighting skills and for pushing me to (and over) my physical limits, sometimes with insane exercises and competitions. Furthermore, I greatly enjoyed the entertaining and intellectual debates till the early hours with Ben, Geert, Jelle, Samuel, Steven and Wouter (often evoking fond memories of our student years).

Ten slotte wil ik mijn dankbaarheid betuigen aan diegenen die mij het meest dierbaar zijn: mijn familie. In het bijzonder wil ik mijn ouders en grootouders bedanken voor hun continue en onvoorwaardelijke steun, hun luisterend oor en de wijze raad. Vooral in de moeilijkere periodes gaf dit mij de kracht om door te zetten. Ook mag ik zeker mijn broer Daniël en zus Andrea niet vergeten, om mij eens af te reageren en te ontspannen, om te luisteren (of te doen alsof) als ik weer aan het zagen was, en omdat ik altijd op hen kan rekenen. Verder wil ik nonkel Guy ("Oh-Noh") bedanken bij wie ik altijd terecht kon met mijn blessures (helaas al te vaak), maar gelukkig ook voor het nodige plezier en amusement. Ik wil ook de rest van de familie bedanken voor hun continue interesse in mijn werk, en de familiefeesten waar ik mij eens goed kan laten gaan in hevige discussies. Heel erg bedankt allemaal!

Up to the next adventure!

— Stefan Walraven

Abstract

Software as a Service (SaaS) has been increasingly adopted by software vendors as their main software delivery model, as it provides the opportunity to offer their software applications to a larger market and to benefit from the economies of scale. One of the key enablers to leverage economies of scale is multi-tenancy: resources are shared among multiple customer organizations (the so-called tenants), which leads to higher resource utilization and scalability. The highest degree of resource sharing is achieved with application-level multi-tenancy. However, this focus on increased resource sharing typically results in a one-size-fits-all approach. As a consequence, multi-tenant SaaS applications are inherently limited in terms of flexibility and variability, and cannot be customized to the different and varying requirements of the different tenants.

This dissertation presents both a middleware framework and a software engineering method to facilitate the development, operation and management of customizable, multi-tenant SaaS applications. More specifically, the middleware framework improves the flexibility of multi-tenant SaaS applications by enabling tenant-specific customizations, while preserving the economies of scale and limiting the application engineering complexity. The focus is on dynamically composing software variants on a per tenant basis as well as on enforcing tenant-specific performance service level agreements (SLAs) throughout the SaaS application. The service line engineering (SLE) method aims to reduce the management complexity of many co-existing tenant-specific configurations as well as the effort to provision tenants and to update and maintain the customizable SaaS application.

This work has been validated and evaluated in the context of two types of industry-relevant SaaS applications, i.e. a request-driven online hotel booking application and a batch-driven document processing application. We have implemented different prototypes on top of existing cloud platforms and the evaluation shows the effectiveness of our solution while introducing only a very limited performance overhead.

Beknopte samenvatting

Software as a Service (SaaS) maakt in toenemende mate opgang bij softwareleveranciers als het belangrijkste distributiekanaal. Het biedt hen de mogelijkheid om hun softwaretoepassingen aan een grotere markt beschikbaar te stellen en om te profiteren van schaalvoordelen. Een van de voornaamste manieren om deze schaalvoordelen te bekomen is multi-tenancy, nl. het verdelen van IT-middelen over meerdere bedrijfsklanten (de zogenaamde tenants), wat leidt tot een hogere benutting en schaalbaarheid van deze middelen. Maximale benutting van IT-middelen wordt bekomen via multi-tenancy op het applicatieniveau. Dit leidt echter tot een uniforme aanpak. Bijgevolg zijn multi-tenant SaaS toepassingen intrinsiek beperkt op het gebied van flexibiliteit en variabiliteit, en kunnen ze niet aangepast worden aan de verschillende en variërende vereisten van de verschillende tenants.

Dit proefschrift presenteert zowel een middlewareraamwerk als een software engineering methode om de ontwikkeling, uitvoering en beheer van aanpasbare, multi-tenant SaaS toepassingen te ondersteunen. Meer bepaald, de middleware verhoogt de flexibiliteit van multi-tenant SaaS toepassingen doordat het aanpassingen op maat van de tenant mogelijk maakt, terwijl de schaalvoordelen behouden blijven en de complexiteit voor het ontwikkelen van de applicatie beperkt blijft. De focus ligt hierbij zowel op het dynamisch en per tenant samenstellen van softwarevarianten, als op het afdwingen van de dienstverleningsovereenkomsten met de tenants (d.w.z. SLA's) over de geleverde prestaties en dit doorheen de gedistribueerde toepassing. De service line engineering (SLE) methode heeft als doel om de beheerscomplexiteit te verminderen met betrekking tot de vele, naast elkaar bestaande configuraties van de verschillende tenants, maar ook om de vereiste inspanning en de kosten voor de dienstverlening te beperken en om de SaaS toepassing te updaten en te onderhouden.

Dit werk werd gevalideerd en geëvalueerd in de context van twee soorten bedrijfsrelevante SaaS toepassingen, nl. een hotelboekingssysteem en een documentverwerkingssysteem. We hebben verschillende prototypes ontwikkeld boven op bestaande cloud platformen, en uit de evaluatie van deze prototypes blijkt de doeltreffendheid van onze oplossingen, met een minimale impact op de geleverde prestaties.

Contents

Chapter 1

Introduction

Computing may someday be organized as a public utility just as the telephone system is a public utility.
— Prof. John McCarthy

During the last decades, computing systems have transformed from the mainframe paradigm (since 1950s) over enterprise computing with client-server (since 1960s), the web (since 1990s) and service-oriented architecture (SOA) (since 2000s), to the paradigm of cloud computing (nowadays). Cloud computing represents the next step in the continuous evolution towards the (ultimate) goal of utility computing [12, 162], as envisioned by Professor John McCarthy in 1961:

> *Each subscriber needs to pay only for the capacity he actually uses, but he has access to all programming languages characteristic of a very large system ... Certain subscribers might offer service to other subscribers ... The computer utility could become the basis of a new and important industry.*

This trend involves the servicing and deployment of computing capabilities in a more efficient and flexible way. On the one hand, each new step towards utility computing enables service providers to offer their services to a larger customer base and to reduce operational costs by increasing the economies of scale [12, 44]. On the other hand, customers (i.e. end users as well as organizations and companies) are able to outsource more non-core activities to a service provider, and have on-demand access to a whole range of services, while only being charged on actual usage basis.

However, with each new paradigm comes the need for new software systems and middleware platforms as well as updated software engineering approaches to fully exploit the benefits associated with the new paradigm. Moreover, the focus on increased economies of scale typically results in a one-size-fits-all solution that does not support the continuously changing demands and requirements of the growing customer base. Therefore, these new approaches should tackle this trade-off and ensure that other important software qualities, such as customizability, maintainability, performance and security, are preserved.

This dissertation focuses on providing such methodical support and middleware for the development and operation of customizable Software-as-a-Service (SaaS) applications in an efficient and scalable manner.

This chapter first presents the cloud computing paradigm, introducing concepts such as SaaS, and its main characteristics and benefits (compared to the previous paradigms). Second, we highlight the main challenges with respect to developing and managing cost-efficient SaaS applications tailored to the customers' needs. Next, the goals and approach of this dissertation are discussed. Subsequently, the contributions of this work are presented. Finally, this chapter concludes with an overview of the structure of this dissertation.

1.1 Cloud computing

We define cloud computing as a paradigm that enables the on-demand delivery of ICT solutions as online services, covering software applications, system software, and hardware infrastructure. Moreover, these services can be rapidly provisioned on request of the customers with minimal manual effort required from the provider. This definition is derived from the National Institute of Standards and Technology (NIST) [130], Armbrust et al. [12] and Zhang et al. [224].

This section elaborates on the main characteristics and benefits of cloud computing and compares them with previous paradigms. Furthermore, it presents the cloud computing architecture, including the different delivery and deployment models.

1.1.1 Characteristics and benefits

Although some claim that cloud computing is just another term to describe already existing approaches [62], cloud computing has some specific characteristics and associated benefits [130, 224] that differ from what is offered by previous paradigms.

Higher degree of distribution. A cloud environment is distributed at a large scale to achieve *high availability* and high performance (i.e. high throughput and low latency) via replication. A typical cloud consists of many data centers (using commodity hardware) at different geographical locations [12, 224], creating a heterogeneous geo-distributed system. Furthermore, the cloud services are accessible from any device and location through standard mechanisms and APIs. Such a high degree of distribution is lacking in the previous paradigms. For example, a mainframe is a single, highly reliable machine, and a SOA consists in practice of multiple services distributed over a limited set of nodes (often within a single organization) that are interconnected via an enterprise service bus (ESB).

Multi-tenancy. Multi-tenancy is one of the key enablers to realize *economies of scale* [44, 84]. While in the previous paradigms a service provider manages one dedicated service setup per customer (cf. traditional application service provisioning (ASP)), multi-tenancy enables cloud providers to employ resources more efficiently by dynamically assigning the available resources to multiple customer organizations, the so-called *tenants*. Thus, operational costs are significantly reduced by multiplexing (shared) resources among multiple tenants. In this perspective, multi-tenancy can be seen as an evolution of time-sharing on mainframes in the 1960s and 1970s. In principle, a multi-tenant architecture also simplifies administration and provisioning of tenants as well as maintenance. For example, shared resources can be upgraded for all tenants at once.

Elasticity. Computing resources are dynamically provisioned and released on demand (i.e. in an elastic way), giving customers the appearance of infinite capacity. Compared to the traditional model(s), cloud computing offers customers the *flexibility* to scale out rapidly (and preferably automatically) at a fine-grained level based on the current workload, without having to constantly provision sufficient resources for peak demand and thus avoiding over- or underprovisioning. Traditionally, systems scale up by upgrading nodes or replacing them with more powerful ones. This so-called vertical scaling results in expensive nodes and eventually reaches hardware limitations. Elasticity, however, is achieved by *scaling horizontally*: adding more nodes to the system and distributing the load. This is usually cheaper (commodity hardware) and can in theory scale infinitely.

Delivery as a service. As indicated by its definition, cloud computing adopts a service-driven delivery model[1]: customers do not (have to) manage the resources and applications themselves, but they are outsourced to the cloud provider. This results in economic benefits for the customers by supporting a *utility-based pricing strategy* and thus by converting capital expenses (capex) to operational expenses

[1]Notice the difference with SOA: in the context of SOA, the fundamental building block is a service, but the application itself (i.e. the orchstration of these services) is not necessarily delivered as a service.

(opex) [12, 224]. This way, customer organizations do not have to make upfront commitments and investments in infrastructure, but they only have to pay for the resources and applications they actually use (i.e. pay per use). However, this also has consequences for the cloud provider: he has to measure the usage per customer at an appropriate level of abstraction (e.g. amount of low-level resources used, number of end users per customer organization), (i) to control the operational costs, (ii) for billing purposes, and (iii) to ensure that the delivered service is compliant to the agreed service level agreements (SLAs). This requires support for monitoring, management of the operational costs and SLAs, and reporting.

Self-service. To preserve *scalability* with an increasing number of customers and to rapidly respond to changing service demand and customer requirements, capabilities should be provisioned automatically without requiring human interaction with the cloud provider. This requires a higher degree of automation compared to previous paradigms, which mainly involved human interaction with slow response times as a consequence. Moreover, the customers should be empowered to manage the cloud services they are subscribed to via APIs, web tools and configuration interfaces, e.g. enrolling for a service, managing end users and access control, allocating resources on demand, configuring applications etc.

1.1.2 Cloud computing architecture

Cloud applications are typically structured according to a multi-layered software architecture: they comprise of at least an infrastructure layer, a platform or middleware layer, and an application layer (see Fig. 1.1). Furthermore, monitoring and cloud management (e.g. resource allocation and configuration management) are responsibilities that cross these different layers.

Depending on which layers are maintained by respectively the cloud provider and the customer (cloud consumer), three cloud service delivery models can be distinguished [130, 224]:

- *Infrastructure as a Service (IaaS)* delivers fundamental computing resources, such as processing, storage and network capacity, as a service (cf. bottom layer in Fig. 1.1). Using *virtualization* technology these resources are partitioned and assigned to different customers. Customers can deploy, execute and fully control arbitrary software in their virtual machines that run on top of the shared infrastructure. This includes operating systems, system software and applications. However, this also implies that the customers bear full responsibility. Amazon EC2 [2] is a well-known example of an IaaS offering.

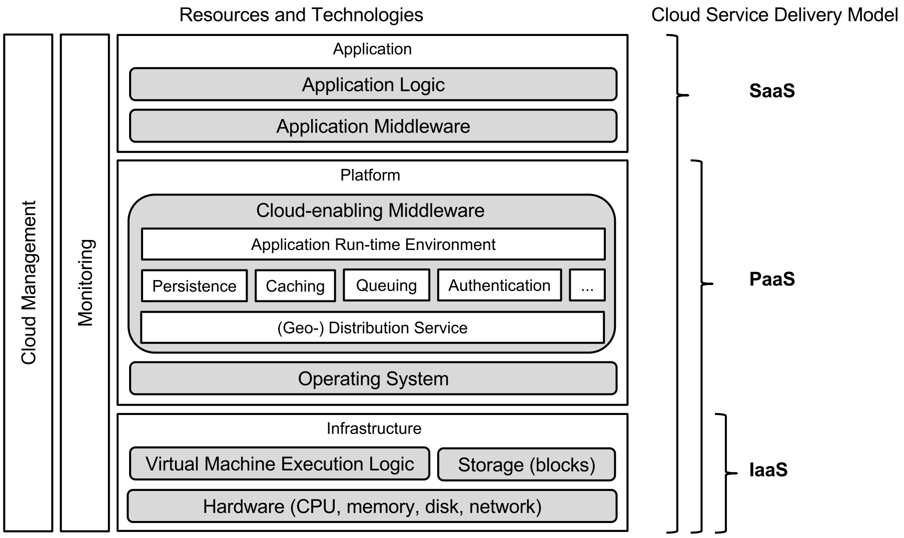

Resources and Technologies
Cloud Service Delivery Model
Cloud Management
Monitoring
Application
Application Logic
Application Middleware
Platform
Cloud-enabling Middleware
Application Run-time Environment
Persistence
Caching
Queuing
Authentication
...
(Geo-) Distribution Service
Operating System
Infrastructure
Virtual Machine Execution Logic
Storage (blocks)
Hardware (CPU, memory, disk, network)
SaaS
PaaS
IaaS

Figure 1.1: Overview of the multi-layered cloud architecture and the corresponding service delivery models

- *Platform as a Service (PaaS)* provides a higher-level computing platform and solution stack upon which applications and services can be developed and hosted by using programming concepts and tools supported by the provider. The core of the platform layer consists of a *cloud-enabling middleware* that provides distribution and clustering support as well as the run-time environment and common cloud services such as distributed storage and caching (cf. middleware layer in Fig. 1.1). Many current middleware vendors (e.g. JBoss, Oracle and SpringSource) have adapted their offerings for this purpose. The customer only has control over the deployed applications, while the platform manages infrastructure-related concerns like load balancing, scalability and availability. Examples are Google App Engine [77], Red Hat OpenShift [175] and Windows Azure [134].
- *Software as a Service (SaaS)* delivers software applications as online, on-demand services, for example Salesforce CRM [183]. The application logic consists of a web application or service, a workflow or even an entire business process, and it is often combined with application- and domain-specific middleware (cf. top layer in Fig. 1.1). The control of the customer is limited to adapting the customer-specific application configuration. The infrastructure, platform and application are managed by the provider.

Orthogonally to these cloud service delivery models, the cloud computing paradigm has several deployment models [12, 130, 224]:

- *Public clouds* provision resources over the Internet for use by the general public. The cloud providers ensure scalability by providing large amounts of computing resources, while customers do not need to invest in hardware and infrastructure. However, fine-grained control over data and applications is lacking.
- A *private cloud* is operated solely for a single organization (i.e. internal customers) and is managed by this organization or a third party (on-premise or hosted). Typically, private clouds are considered more secure, do not depend on Internet availability, and offer more control and customizability. However, they have a limited resource capacity.
- A *multi-cloud* is a composition of multiple cloud environments with as purpose to improve availability and to avoid vendor lock-in. A *hybrid cloud* is a special type of multi-cloud that consists of at least a public and a private cloud environment, thus combining the unlimited capacity of public clouds with the increased control of private clouds into an integrated system [113]. Typically, customers use public clouds as a spill-over cloud to handle peak workloads (i.e. cloud bursting), while only having to pay for these extra resources when they are actually needed. A key concern with respect to a multi-cloud is portability of applications and data across the different cloud environments.

This dissertation addresses enhanced support for the development and management of *public SaaS* applications, i.e. SaaS applications that are offered to external customer organizations. Public SaaS offerings are aimed at a larger market and thus have more explicit requirements with respect to cost efficiency, performance, scalability and availability compared to an internal SaaS application. Notice that a public SaaS application can be deployed on top of a public as well as a private cloud infrastructure (IaaS or PaaS). In the latter case, the underpinning private cloud platform and/or infrastructure is only accessible internally, but the application running on top of it is publicly available. *The enhanced support in this dissertation is manifested in an application middleware, located between the actual application logic and the underpinning cloud platform (see Fig. 1.1).*

1.2 Challenges for the customization of SaaS

New and existing software vendors increasingly adopt SaaS as their main software delivery model. The SaaS model provides these software vendors the opportunity to offer their software applications to a larger market and to benefit from the economies of scale associated with cloud computing. However, this migration to the SaaS model is not straightforward for them: the cloud computing paradigm introduces, in contrast to the traditional on-premise delivery model, additional responsibility and complexity to manage and monitor the underpinning infrastructure and system software, as well as to deploy and host the applications. Several (often open source) solutions exist to support SaaS providers in setting up a private cloud environment, for example OpenStack [156] and Red Hat OpenShift Origin [175], but this still requires large investments in infrastructure and system management. Alternatively, SaaS providers can (partially) outsource this complexity to an external IaaS or PaaS provider.

In this dissertation we focus on the challenges at the application level, more specifically the development, operation and management of a SaaS application. In order to make a SaaS offering profitable, the SaaS provider is required to harvest full benefit from economies of scale. If well-achieved by the SaaS provider, these benefits add value to the offering and reduce the operational costs of hosting and maintaining the SaaS application. As explained in Section 1.1, one of the key enablers to leverage economies of scale is *multi-tenancy* by sharing resources among tenants, which leads to higher resource utilization and scalability. Different architectural strategies can be applied to achieve multi-tenancy: at the infrastructure level via virtualization, at the OS and middleware level, and at the application level (see Fig. 1.2). Application-level multi-tenancy results into the highest degree of resource sharing [44, 84]: end users from different tenants are simultaneously served by a single application instance on top of shared hardware and software infrastructure.

In addition, customization capabilities are critical to allow a SaaS provider to differentiate from competitors [201]. Every tenant is unique with its own requirements, and consequently standardized functionalities and services lose their value as these requirements become more complex and diverse [81]. Especially in the case of cloud computing with its potentially large customer base, *mass customization* becomes increasingly relevant, i.e. offering software services tailored to the tenant's needs, where and when they need it, and at a price similar to the default (non-customizable) alternative [56, 81].

Comparing the different levels of multi-tenancy, application-level multi-tenancy achieves the highest degree of resource sharing between tenants, but typically results in a one-size-fits-all approach: the multi-tenant application only satisfies the requirements that are common to all tenants. However, the customization of multi-tenant SaaS applications hinders economy of scale. Thus, a trade-off is required between operational cost efficiency and flexibility, as illustrated in Fig. 1.2. It is our ambition to address this trade-off by enabling the customization of multi-tenant SaaS applications while preserving the economies of scale.

We have identified the following concrete challenges with respect to the customization of multi-tenant SaaS applications:

1. *Increased engineering complexity.* Application-level multi-tenancy already introduces additional application engineering complexity for the SaaS provider to ensure the necessary isolation between the different tenants as well as to provide a tenant-

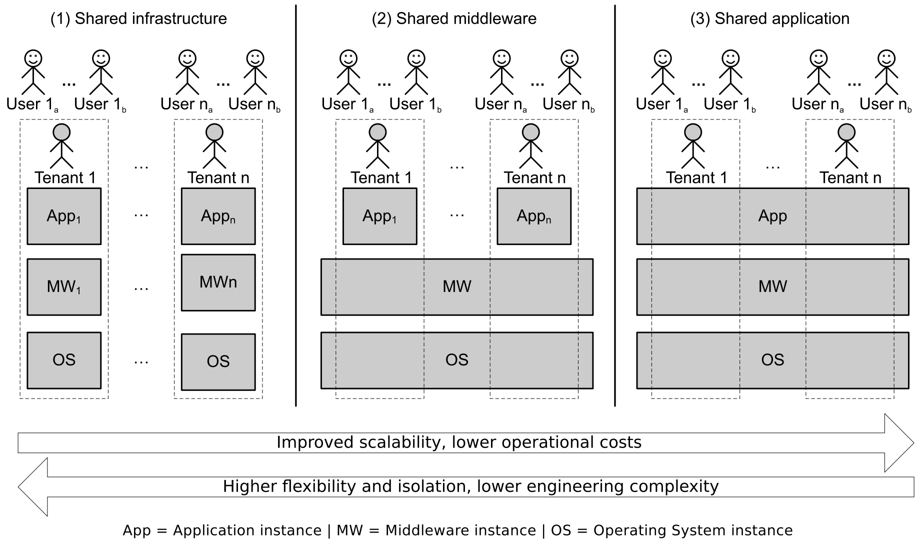

(1) Shared infrastructure
(2) Shared middleware
(3) Shared application
User 1a ... User 1b
User na ... User nb
Tenant 1
Tenant n
App1
Appn
App
MW1
MWn
MW
OS
Improved scalability, lower operational costs
Higher flexibility and isolation, lower engineering complexity
App = Application instance | MW = Middleware instance | OS = Operating System instance

Figure 1.2: Illustration of the trade-off between high cost efficiency and scalability versus high isolation and flexibility as well as low engineering cost, in the context of the different architectural strategies for multi-tenancy.

specific management facility [84] (see Fig. 1.2). In addition, customizations have to be isolated so that they only apply to a particular tenant and do not affect the service behaviour that is delivered to other tenants (e.g. functionality, performance, security). Support for such concurrent tenant-specific customizations is lacking, which results in complex software constructions at the application level and high performance overhead.

2. *Explosion of variations.* Mass customization of the SaaS application tailored to the different and varying requirements of all tenants leads to an explosion of variations. Furthermore, continuous maintenance and evolution of the application (e.g. patches or upgrades) will only enhance this effect with different versions of the application as well as the different variations. This problem worsens with the success of a SaaS offering: with an increasing amount of tenants the degree of variability becomes unmanageable by the SaaS provider, and thus threatens the crucial scalability and cost efficiency [201].

1.3 Goals and approach

Customization of SaaS applications involves non-trivial challenges with respect to the software engineering approach as well as the supporting middleware. Consequently, the overall goal of this dissertation is to study, develop and evaluate both methods and middleware platforms to develop and operate customizable SaaS applications, while preserving the key benefits of cloud computing such as cost efficiency and scalability. More specifically, the focus is on tackling the trade-off presented above (cf. Fig. 1.2) by introducing flexibility into multi-tenant SaaS applications. To achieve this, we build upon two research lines: (i) software product line engineering (SPLE) [48, 168] to improve reuse by exploiting commonalities and to manage variations, and (ii) adaptive and reconfigurable middleware [31, 52, 103, 104, 206] to realize run-time customizations.

We break up the overall goal into the following subgoals:

1. *Dynamic and context-sensitive adaptation and composition.* A multi-tenant SaaS application should adapt at run time based on the current context. This context is defined by the tenant associated to the current interaction and its specific requirements, as well as by the state and behaviour of the cloud environment (e.g. the current load has an impact on performance). Because a multi-tenant application is shared between end users of multiple tenants, variants for different tenants have to be composed and activated concurrently within the same SaaS application instance. Therefore, it is critical that these *co-existing variants* are isolated from each other, i.e. the adaptations should occur within the scope of a single tenant and may not have an impact on other tenants.
2. *Scalable variability management.* Variability emerges from different and varying tenant requirements, maintenance and evolution, and has a high impact on the manageability and thus scalability of the SaaS application. To avoid an explosion of variations, the versatility of software variations should be improved. More specifically, customizations, driven by high-level configurations, have to be *automatically* translated into the composition of a set of reusable software modules. Furthermore, a part of the configuration complexity should be shifted towards the tenant (i.e. *self-service*) to maintain the scalability with an increasing amount of tenants.

Approach. Our approach is *application-driven* in the sense that two common types of enterprise SaaS applications have been used throughout this work (i) to analyse the current state of practice, (ii) to illustrate the open challenges, and (iii) to validate and evaluate the proposed solutions. The *hotel booking application* is a component-based multi-tier (web) application, which is request-driven and deployed as a single (monolithic) software service. Customization is thus limited to this single software service, although the service can be replicated for scalability purposes. The second application is based on a real-world SaaS offering in the domain of online *B2B document processing*. Concretely, it is a batch-driven service composition: for each incoming batch of input data a particular workflow is executed. Both the workflow and the different services in that workflow are subject to the customization requirements of the tenants. In addition, non-functional concerns such as performance apply to the end-to-end execution of the workflow.

The research approach starts with the study of and hands-on experience with the current state of practice of middleware support for SaaS, i.e. the existing cloud-enabling middleware platforms and PaaS offerings. By building upon commonly used platforms, we ensure the applicability and (re)usability of the middleware solution. Representative platforms have been investigated with respect to their support for the development and customization of multi-tenant SaaS applications, and the open challenges have been identified.

The second step in our approach *combines the concepts and method of SPLE with the flexibility of adaptive middleware* in order to achieve the two goals we defined above. Moreover, we incrementally improve and evolve both research lines to make them suitable for usage in a multi-tenant context. In particular, the software engineering method should address the management complexity of many co-existing tenant-specific configurations, and the adaptive middleware should offer a composition mechanism that enables run-time customization scoped to a specific tenant. Furthermore, the adaptive middleware has to be integrated into the software engineering method. As common abstraction for variations in this integrated approach we use *features*. A feature is a distinctive functionality, service, quality or characteristic of a software system or systems in a domain [100]. We rely on feature-oriented modularization [9, 51, 118, 171] to implement these features as composable and reusable software modules [163] that can be composed by the underpinning middleware.

1.4 Contributions

The key contributions of this dissertation are summarized below:

- *Comparison of representative PaaS platforms and gap analysis.* We have compared three different and representative PaaS offerings with respect to their support for SaaS application development, based on hands-on experience with the (re-)engineering and implementation of the hotel booking application. Concretely, we have investigated the portability of the application code base, the available support for creating and managing multi-tenant applications, and the quality of the tool support. Furthermore, we have identified the following research challenges for the improvement of current PaaS offerings in terms of supporting SaaS application development:

  - Support for *application-level multi-tenancy* in all its aspects is lacking. PaaS offerings should offer support for data isolation and tenant-specific customization, tenant-specific application management as well as performance isolation.
  - *Portability* of SaaS applications across different cloud-enabling middleware platforms and PaaS offerings is required to improve migration and to tackle vendor lock-in, especially in the context of common cloud services (e.g. scalable storage), multi-tenancy support, and application management and monitoring.

  This dissertation focuses on the multi-tenancy-related challenge.

- *Middleware support for co-existing variants in multi-tenant SaaS applications.* We provide a middleware framework that (i) complements existing cloud-enabling middleware and PaaS offerings, and (ii) improves flexibility of multi-tenant SaaS applications by enabling tenant-specific customizations, while (iii) preserving the operational cost benefits and limiting the engineering complexity. Concretely, tenants can customize the multi-tenant SaaS application to their specific functional and non-functional requirements via a feature-based configuration interface. Based on these tenant-specific configurations and the current context, the middleware layer dynamically performs the end-to-end customization throughout the distributed SaaS application on a per tenant basis. Two specific instances of this middleware framework have been developed with focus on:

  - The *run-time composition of software variants* in multi-tier component-based SaaS applications driven by tenant-specific configurations. We have implemented a prototype on top of Google App Engine, and its evaluation in the context of the hotel booking application shows improved flexibility with little impact on the operational costs.
  - The *run-time enforcement of performance isolation* in compliance with tenant-specific SLAs over distributed environments for multi-tenant SaaS. The prototype implementation on a JBoss AS-based private cloud platform has been evaluated using the document processing application, introducing only limited performance overhead.

- *Service line engineering method.* We present a method similar to SPLE, called service line engineering (SLE), that facilitates SaaS providers to introduce and manage variability in multi-tenant SaaS applications without compromising the scalability. The concept of a *service line* is based on the notion of software product lines, but offers a shared application instance that still is dynamically customizable to different tenant-specific requirements. The SLE method is highly integrated and generic, in the sense that the feature-level variability is consistently and explicitly supported in each of the subsequent stages, while these stages are open for existing work to be leveraged upon. The application of this method results (i) in efficiency benefits with respect to addressing the management complexity of many co-existing tenant-specific configurations, and (ii) in a significant reduction of effort to provision tenants (i.e. configuring and composing an application variant) as well as to update and maintain the service line as a whole, though at the expense of a higher initial development effort. The SLE method has been validated in the development of the document processing SaaS application.

1.5 Structure of the dissertation

This dissertation consists of the unedited versions of three key publications and one paper draft made in the course of my doctoral program. It is structured as follows.

Chapter 2 presents the first contribution of this dissertation: the analysis of the state of practice in the development support for SaaS applications on top of representative PaaS offerings, and the identification of open research challenges in this context. This work has been published in the Springer Computing journal [215].

The middle part consists of three chapters that discuss the second contribution in detail. Chapter 3 presents an overview of the middleware framework that enables SaaS providers to build and deploy customizable, multi-tenant SaaS applications. The focus of Chapter 4 is on the middleware support for the dynamic and cost-efficient composition of software variants in multi-tier component-based SaaS applications. The content of Chapter 4 has been published in the proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware (Middleware 2011) [214]. Chapter 5 elaborates on the middleware architecture to enforce performance isolation in distributed SaaS applications in compliance to the tenant-specific SLAs.

In Chapter 6, the third contribution of this dissertation is described, i.e. the service line engineering (SLE) method that facilitates the development, deployment, run-time configuration and composition, operation and maintenance of multi-tenant SaaS applications, without compromising scalability. This work has been published in the 91th volume of the Journal of Systems and Software [217].

Finally, Chapter 7 concludes this dissertation and discusses its applicability. Furthermore, ongoing work and remaining challenges for future research in the context of multi-tenant SaaS and utility computing are presented.

Chapter 2

Comparing PaaS offerings in light of SaaS development *

Preamble

The adoption of Platform as a Service (PaaS) in industry is growing and PaaS shows the potential of becoming the main development and deployment platform for enterprise SaaS applications [146]. This chapter compares different existing PaaS offerings with respect to their support for SaaS application development. More specifically, the concerns with respect to application engineering and operational costs (cf. Section 1.2) have been translated into the development requirements of portability of application code base, multi-tenancy support, and qualitative tool support. The main contribution of this work is introduced and summarized in Sections 2.1 and 2.7, i.e. the distinction between three categories of PaaS offerings, the discussion of the results of the comparative study, and the identification of open challenges for the development of SaaS applications. None of the platforms offers sufficient support for tenant-specific customization. Actually, most platforms even lack any support for multi-tenancy. The details of the comparison are described in Sections 2.2—2.6, including a description of the requirements and criteria, the three representative PaaS offerings that we investigated, and the technical comparison using the online hotel booking application (as introduced in Section 1.3).

*The content of this chapter has been published in the Springer Computing journal [215].

Comparing PaaS offerings in light of SaaS development: A comparison of PaaS platforms based on a practical case study

Abstract

Software vendors increasingly aim to apply the Software-as-a-Service (SaaS) delivery model instead of the traditional on-premise model. Platforms-as-a-Service (PaaS), such as Google App Engine and Windows Azure, deliver a computing platform and solution stack as a service, but they also aim to facilitate the development of cloud applications (SaaS). Such PaaS offerings should enable third parties to build and deliver multi-tenant SaaS applications while shielding the complexity of the underpinning middleware and infrastructure. This paper compares, on the basis of a practical case study, three different and representative PaaS platforms with respect to their support for SaaS application development. We have reengineered an on-premise enterprise application into a SaaS application and we have subsequently deployed it in three PaaS-based cloud environments. We have investigated the following qualities of the PaaS platforms from the perspective of SaaS development: portability of the application code base, available support for creating and managing multi-tenant-aware applications, and quality of the tool support.

2.1 Introduction

Cloud computing is a trend that refers to the delivery of ICT solutions as online services, covering software applications, system software, hardware infrastructure, etc. The concept of cloud computing includes three cloud service delivery models [130, 209]:

- *Infrastructure-as-a-Service (IaaS)* is the delivery of fundamental computing resources (e.g. processing, storage and networks) as a service, for example Amazon EC2 [2].
- *Platform-as-a-Service (PaaS)* provides a computing platform and solution stack upon which applications and services can be developed and hosted by using programming concepts and tools supported by the provider. Examples include Force.com [182, 219] and Google App Engine [77].
- *Software-as-a-Service (SaaS)* is a software deployment model that delivers software applications as online, on-demand services, for example the Salesforce CRM application [183].

Software vendors increasingly deliver their software applications as online services, in accordance with the above-mentioned SaaS model, and thus become SaaS providers. This enables their customers to use these software services on demand. Customers are thus freed from the management of the underpinning middleware and infrastructure. But the SaaS model requires software vendors to harvest full benefit from economies of scale. Resources could thus be employed more efficiently and maintenance efforts can be centralized. These benefits, if well-achieved by the SaaS provider, add value to the offering, and reduce the operational costs of providing the software services.

One of the key enablers to leverage economies of scale is *multi-tenancy* [44, 84]. Typically, a service provider manages one dedicated application instance per customer (i.e. single-tenancy). We focus on the particular case of application-level multi-tenancy. In this context, we define multi-tenancy as an architectural style that enables service providers to serve end users from different *tenants* (i.e. customer organizations and companies) simultaneously by a single application instance on top of shared hardware and software infrastructure [84]. Besides the benefits of economies of scale, a multi-tenant architecture simplifies administration and provisioning of tenants, for example upgrades of the application only have to be applied to the shared instance and all tenants will automatically use the most recent version.

The decision to apply multi-tenancy for a SaaS application depends on the application- and tenant-specific requirements. Often this is a trade-off between cost efficiency and flexibility [214]. With multi-tenancy a higher level of sharing can be achieved, which leads to higher scalability and maximal resource utilization. However, it also introduces additional complexity for the SaaS developers (i) to ensure the necessary isolation between the different tenants [84], (ii) to offer a customizable application in order to meet the unique requirements of the different tenants [201], and (iii) to manage and monitor the different tenants in a fine-grained way.

PaaS tries to solve the challenges sketched above by delivering a higher-level computing platform and software stack, which aims to facilitate the *development and deployment* of cloud applications while also managing availability, elastic scalability, load balancing and other infrastructure-related concerns. PaaS can therefore play a major role for software vendors in the adoption of the cloud computing paradigm. If such a combination of a scalable platform and a full software stack can be provided, it would reduce the entrance barrier for software vendors to offer their respective capabilities to a wide market with a minimum of entry costs and infrastructure requirements [188]. Most recent research has focused on IaaS (e.g. Eucalyptus [153] and OpenNebula [198]), which allows software vendors to deploy and run arbitrary software on top of a virtualized infrastructure [130]. However, the maintenance of operating systems, application servers and middleware platforms is expensive and requires sufficient system administrators with the appropriate skills. This would make PaaS a more interesting choice for software vendors willing to enter the SaaS market.

As with any paradigm shift, many new and somehow competing technologies enter the arena; more technologies emerge and compete when the new paradigm (SaaS in cloud computing in the context of this research) is promising major changes in the landscape of solution and platform providers. These circumstances generate an imminent need to investigate and conduct independent research on the relative value and performance of new types of technologies.

A comparable situation has emerged, for example in the eighties, when performance-critical software applications were expected to be executed by various kinds of parallel computing technologies, ranging from MIMD [26], SIMD [194], shared-memory environments [195] etc. Very often, parallel computing architectures came along with specific programming paradigms and tools for parallel computing, such as parallelizing compilers [91], message-passing environments [196] etc. Back then, the research community had to deliver a lot of practical research to share experience and know-how, and to drive the qualitative improvement of the corresponding platforms. Today the domain of PaaS platforms that support the paradigm shift towards SaaS in cloud computing needs similar research.

The general research question that is addressed in this paper therefore is to which extent different types of Platform-as-a-Service solutions support the development and deployment of SaaS applications that start from an existing code base. The approach that has been applied is case study driven. The specific scope of the paper is in principle limited to the broad category of enterprise applications (such as the ones based on Java EE and .NET).

Clearly, PaaS offerings need a critical assessment: can PaaS platforms enable software developers to build and deliver SaaS applications without the complexity of dealing with the underpinning middleware and infrastructure? This experience report compares PaaS platforms with respect to their support of SaaS application development. We distinguish between the following three categories of PaaS platforms:

Category 1: PaaS platforms that mimic and match the APIs of popular enterprise application servers and middleware platforms. Examples are Windows Azure [134] using the .NET framework and the IIS web server, Red Hat OpenShift [175] based on the JBoss platform, Oracle Cloud [159] running on top of the WebLogic Server, and Cloud Foundry [211] using VMware and Spring technology.

Category 2: Focused PaaS platforms that aim to optimally support specific types of cloud applications. These platforms typically deploy other middleware and storage facilities that supposedly scale better. Google App Engine [77] and GigaSpaces' XAP Elastic Application Platform [76] belong to this category.

Category 3: Metadata-driven PaaS platforms. Similar to focused PaaS platforms, these platforms are designed with SaaS applications in mind. In addition, metadata-driven PaaS platforms introduce a higher-level composition and configuration interface that makes abstraction of the typical middleware level. However, this limits the complexity of the applications that can be implemented. Examples include Force.com [182, 219], WOLF [221], and TCS InstantApps [192].

In order to compare these PaaS categories, we have reengineered an on-premise application towards the SaaS model. We have performed this task three times, each time yielding a working application on top of one representative platform for each of the categories listed above. In particular: we have used Windows Azure (category 1), Google App Engine (category 2), and Force.com (category 3).

We have addressed three requirements of SaaS providers with respect to support for SaaS application development. (1) The cloud platform should support the *portability* of SaaS applications [32, 166], between the on-premise implementation and the SaaS-enabled implementations. Portability aims for minimizing the development and code migration efforts when re-engineering an application towards a PaaS deployment environment. (2) Furthermore, the platform should support *the creation and management of multi-tenant applications* [44, 84], which is critical to achieve economies of scale. (3) Finally, *tool support* is essential for SaaS developers to facilitate development and testing on top of PaaS platforms [75]. Other requirements, for instance regarding deployment, scalability and security, are out of scope of this paper.

We have conducted this case study in a disciplined and systematic way. The outcome is explained in detail in this paper, and the practical and technical results are publicly available[1] to enable reproduction of the case study, or more likely, extension of the acquired experience and know-how by adding either application case studies or implementations on new and/or evolved PaaS platforms. We believe that there is a lack of in-depth case studies that reach beyond the exploratory use of PaaS platforms, thus yielding high-level impressions rather than in-depth experience.

The contributions of this work consist of (i) the elaboration of the requirements for the development of multi-tenant SaaS applications, (ii) the comparison of three different and representative PaaS platforms with respect to these requirements based on hands-on experience in a practical case study, (iii) the identification of research challenges for the improvement of current PaaS platforms in terms of supporting SaaS application development, and (iv) an understanding of the impact of the selection of (types of) PaaS platforms when developing SaaS applications.

The remainder of this paper is structured as follows. Section 2.2 elaborates on the PaaS requirements imposed by SaaS developers and presents the case study that is used throughout this paper. The representative PaaS platforms are introduced in Section 2.3. Section 2.4 compares the different PaaS platforms with respect to portability. Section 2.5 covers the support for multi-tenancy, and Section 2.6 addresses the tool support for SaaS developers. Section 2.7 discusses the results and identifies the challenges ahead. Section 2.8 covers related work. Section 2.9 concludes the paper.

2.2 Requirements & illustration

First we define the requirements that software vendors must consider when developing enterprise SaaS applications on top of PaaS platforms. Furthermore, we illustrate and further motivate these requirements by describing the application case study.

The focus of this paper is on two types of stakeholders: *PaaS providers* and *SaaS providers*. SaaS providers offer on-demand software services to their *tenants*, potentially using a PaaS platform to develop and deploy these SaaS applications. The SaaS provider often becomes a customer of the PaaS provider. Another relevant stakeholder for this paper is the *tenant administrator*. Each tenant should assign this special role to someone who is responsible for managing the provided SaaS application on behalf of the tenant, for example to configure the tenant-specific preferences.

[1] `https://distrinet.cs.kuleuven.be/projects/CUSTOMSS/comp/`

2.2.1 Requirements

The requirements that are central to this study are somehow trivial if one considers the overall context of the paradigm shift to SaaS in cloud computing: if software vendors consider evolving to become a SaaS provider, then they typically have economical concerns such as the *cost of re-engineering* the software to become SaaS-enabled, as well as the *cost of delivering service* based on the resulting (SaaS) solution[2]. As for the cost of re-engineering, this concern translates directly into the requirement of application portability and into a second requirement of tool support that can further reduce the efforts that are required to develop and deploy on a specific PaaS platform. As for the cost of delivering service, it is important that investments can be shared for a maximal set of clients; this makes multi-tenancy extremely important in this study.

This section therefore elaborates on three key requirements: (i) support for portability of SaaS applications, (ii) support for creating and managing multi-tenant-aware applications, and (iii) availability of tool support for SaaS developers (e.g. development and testing tools).

Portability. Software portability is the ability to easily deploy and use applications on different environments (e.g. operating systems and middleware platforms). In literature, a software application is defined as portable when the cost and effort to port and adapt it to a new environment is less than the cost of redevelopment of a new version and training of the developers [70, 144]. Portability is a desirable property for SaaS applications [32, 166]. From the perspective of the SaaS providers, portability enables them to port and adapt a SaaS application to different platforms (on premise as well as in the cloud) with minimal changes. This will subsequently enable the deployment and execution on multiple PaaS platforms of the same type, and even enable a hybrid approach (both external PaaS platform and in-house). In addition, the application easily works with other applications on local and remote systems, and developers require little or no retraining. The requirements on portability are decomposed into the following subrequirements:

1. *Compatibility with mainstream programming models and middleware frameworks* is necessary to ensure portability of the application code to several platforms, without requiring the rewriting of the SaaS application or extensive training of the SaaS developers. This includes all software concepts for software engineering activities throughout the software life cycle, such as programming models, component models, testing and deployment.

   The SaaS applications are developed and deployed on top of a PaaS platform instead of the traditional middleware container used in the on-premise model (e.g. Java

[2]This is a generic observation, early practitioners of parallel computing shared similar and essential concerns.

EE or .NET). Such a container provides a specific programming model and a set of common services (often based on standards), covering the non-functional requirements of the application, such as persistence and security. However, PaaS platforms also have to support these common programming concepts and middleware frameworks, without tight integration between application and platform, to facilitate the development and to enable portability.

2. A PaaS platform should offer *support for easy integration with cloud services*, both internal and external. Typically, a PaaS platform offers additional, cloud-specific services. For example, a cloud environment needs a highly performant and scalable datastore and distributed caching. Furthermore a service-oriented environment also requires composition with external (cloud) services and applications, such as scalable cloud storage. Therefore, the portability requirement applies to the platform-provided as well as external (cloud) services that are used by the SaaS application.

Multi-tenancy. As explained above, multi-tenancy is one of the key enablers for the SaaS model to achieve economies of scale and operational cost efficiency [44, 214], but it also introduces some complexity [84]. Therefore a PaaS platform should offer SaaS providers a way to endow applications with multi-tenancy, in a transparent way or via an API, in order to allow them to benefit from the improved cost efficiency. To enable multi-tenancy at the application level the following subrequirements should be fulfilled [25, 84]:

1. The core idea behind multi-tenancy is the sharing of both infrastructure and software among all tenants. Therefore *data isolation* between the different tenants is a key requirement that needs to be ensured, for instance by means of a multi-tenant database, to prevent that tenants can access data belonging to other tenants. In addition, the tenant of an incoming user request should be identified (via a tenant ID) and stored as long as the request is processed (i.e. the tenant context).
2. Each tenant has its specific needs. Therefore *tenant-specific configuration and customization* are critical to satisfy the different requirements [201]. However, these customizations should only be applied to a specific tenant and may not affect the other tenants. In addition, a configuration interface is required to enable the selection of tenant-specific customizations in a safe and consistent way. Each tenant should assign the tenant administrator role to someone who is responsible for configuring the SaaS application to the tenant-specific preferences, setting up the application data (e.g. user management), and monitoring the overall service.
3. In order to manage and control multi-tenant applications, SaaS providers need a *tenant-specific application management facility*. Key requirements with respect to application management are billing and metering. The SaaS provider should be able to monitor the SaaS application on a per tenant basis to determine the

resource consumption by a particular tenant (for pay-per-use billing) and to enforce the different service level agreements (SLA). On the other hand, tenants want to monitor the total usage by their users and check whether the agreed SLAs are provided.

Tool support. Tool support is crucial for the (fast) adoption and ease of use of new technologies. We identified the following subrequirements [75]:

1. A *development environment* is required to facilitate the development of SaaS applications using the programming model of a specific PaaS platform. Eclipse and Visual Studio are examples of popular IDEs among developers.
2. The developed SaaS application will be deployed on the PaaS platform. A *local development server and/or storage* offers SaaS developers the possibility to easily test the application on a local environment simulating the PaaS environment, and to optimize the deployment configuration. This includes support for different types of databases.
3. A *testing framework* is required to efficiently test applications, for instance by means of executing unit tests. These testing tools should be able to interact with the cloud platform (or the simulated environment) and other provided services.

2.2.2 Case study

This section presents the case study in the travel business domain that facilitates the analysis of the development support provided by the different PaaS platforms.

Online hotel booking

A software vendor sells hotel booking systems to travel agencies. This allows the travel agencies to offer their customers an online service to search and book rooms in hotels. Typically, this hotel booking application is deployed on a local server or hosted by the software vendor. The latter is based on a traditional application service provisioning (ASP) model: each travel agency has its own, separate application instance (single-tenancy).

The software vendor decides to offer the online booking application to its customers as a multi-tenant SaaS application, customizable to the different requirements of the travel agencies. The base application is offered at a low cost, but additional services and features can be selected at an additional charge to fit the tenant-specific requirements. In this case the travel agencies play the role of tenant whereas the employees and customers of a travel agency are considered the end users. In addition, the travel agencies have to assign the tenant administrator role to someone of their ICT staff to specify the tenant-specific customization preferences, and to set up the application data (e.g. user management) via the configuration interface. In this example the configuration is limited to the selection of a price calculation implementation.

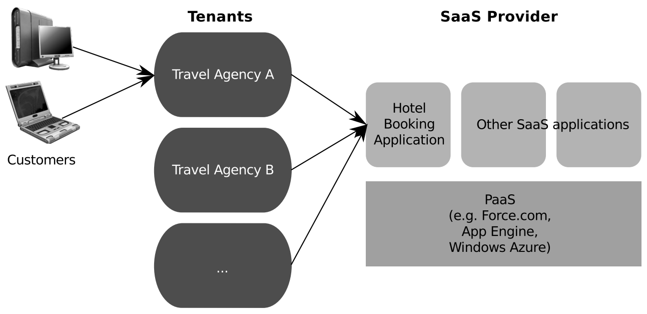


Figure 2.1: The hotel booking application with SaaS deployment model

With the multi-tenant SaaS model all tenants share one or more instances of the online hotel booking application, instead of serving a separate application instance per tenant. This approach reduces the operational costs significantly for the software vendor: (i) hardware and software infrastructure can be more cost-efficiently divided among the tenants, and (ii) upgrades of the software only have to be applied to the shared instance(s) and all tenants will automatically use the most recent version. To provide a scalable platform that allows the travel agencies to cope with the seasonal peeks in booking requests, the software provider wants to deploy the SaaS application on a PaaS platform, as depicted in Fig. 2.1. In principle, this deployment model has the additional advantage that the PaaS platform takes care of the infrastructure-related concerns, and the software vendor can focus on the development and maintenance of the hotel booking application.

The core design of the hotel booking application (Fig. 2.2) consists of two major parts: the data model and the business logic. The data model describes the main entities in the travel business domain. A `Hotel` consists of a name, an `Address` and a list of rooms. The information about a `Room` is captured in a `RoomDetails` object (e.g. the room number and the price per night). Each `Room` also contains a list of bookings. `Booking` contains the details of a booking: the name of the customer that makes the booking, a start and end date (booking period), the total price of the booking and the name of the hotel as well as the room number.

The business tier contains the different components to perform bookings of rooms (`BookingComponent`) and to manage the application data (`ManagementComponent`). These components also provide interfaces that can be used by a front end. The

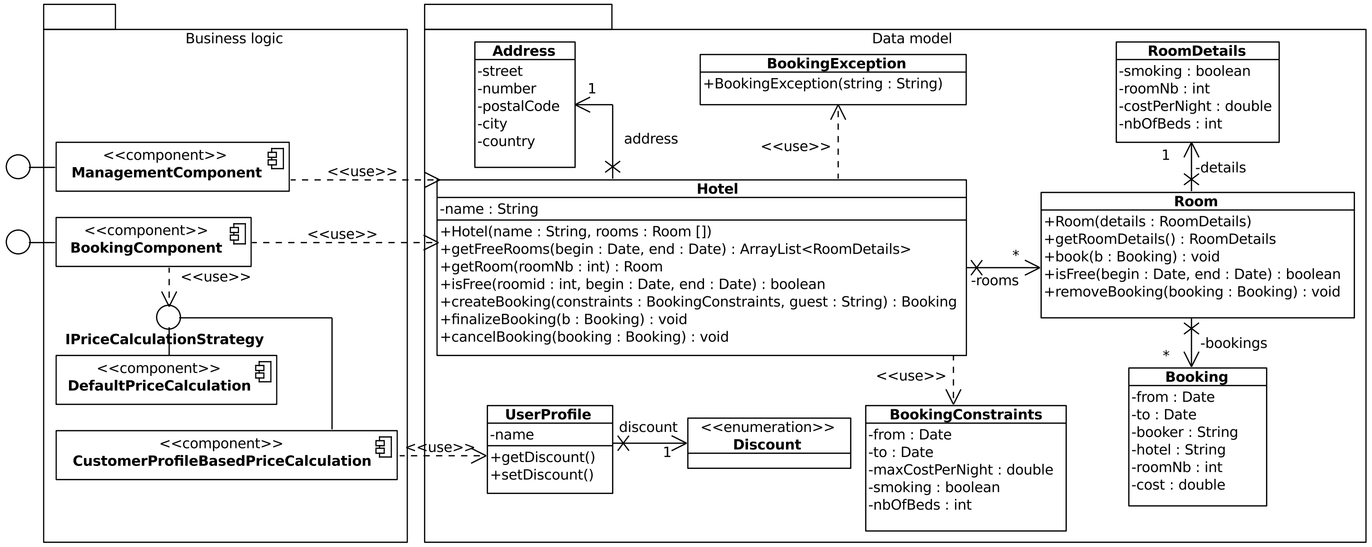

Business logic
<<component>>
ManagementComponent
<<component>>
BookingComponent
<<use>>
IPriceCalculationStrategy
<<component>>
DefaultPriceCalculation
<<component>>
CustomerProfileBasedPriceCalculation
<<use>>
<<use>>
<<use>>
Data model
Address
-street
-number
-postalCode
-city
-country
1
address
BookingException
+BookingException(string : String)
<<use>>
RoomDetails
-smoking : boolean
-roomNb : int
-costPerNight : double
-nbOfBeds : int
1
-details
Hotel
-name : String
+Hotel(name : String, rooms : Room [])
+getFreeRooms(begin : Date, end : Date) : ArrayList<RoomDetails>
+getRoom(roomNb : int) : Room
+isFree(roomid : int, begin : Date, end : Date) : boolean
+createBooking(constraints : BookingConstraints, guest : String) : Booking
+finalizeBooking(b : Booking) : void
+cancelBooking(booking : Booking) : void
*
-rooms
Room
+Room(details : RoomDetails)
+getRoomDetails() : RoomDetails
+book(b : Booking) : void
+isFree(begin : Date, end : Date) : boolean
+removeBooking(booking : Booking) : void
*
-bookings
Booking
-from : Date
-to : Date
-booker : String
-hotel : String
-roomNb : int
-cost : double
<<use>>
UserProfile
-name
+getDiscount()
+setDiscount()
discount
1
<<enumeration>>
Discount
BookingConstraints
-from : Date
-to : Date
-maxCostPerNight : double
-smoking : boolean
-nbOfBeds : int

Figure 2.2: Design of the hotel booking application

management interface enables the travel agencies to (un)register hotels and to get an overview of the bookings.

Customers can make inquiries about the availability of rooms during a certain period (using `getFreeRooms(Date, Date)`) in the currently selected hotel. It also provides methods to book rooms: `createBooking(BookingConstraints, Guest)` creates a tentative booking which can be finalized by using `finalizeBooking(Booking)`. It is possible to cancel a booking using `cancelBooking(Booking)`. Customers can pass their requirements by means of `BookingConstraints`.

A customer has the opportunity to make several bookings in multiple hotels concurrently. Therefore, conversational state, such as the currently selected hotel and the list of tentative bookings, should be kept on the server for the duration of the customer's session. The `finalizeBookings(List<Booking>)` operation of the `BookingComponent` finalizes all tentative reservations in the customer's session.

When a new booking is created, the total price is calculated by `IPriceCalculatorStrategy`. In this simplified case study, price calculation is offered as a feature with two available implementations: (i) the default approach, i.e. multiplication of the number of nights and the price per night, and (ii) a discount is assigned to customers based on their profile (e.g. regular customers get a 10% discount on their bookings). In the multi-tenant SaaS application, the tenants have to select one of these feature implementations. Thus different tenants can have a different configuration.

Implementation highlights

We have implemented an on-premise version of the hotel booking application as a component-based web application, using Java EE as well as the .NET Framework (see Table 2.1).

In the Java EE version we used Enterprise JavaBeans (EJB) [210], a server-side component architecture, for the business logic and the Java Persistence API (JPA) [57], a standard framework for managing relational data, for the data model. JPA also defines an object-oriented query language for entities stored in a relational database, called the Java Persistence Query Language (JPQL). Further, we developed a web tier as front end using Java Servlet [145] technology and Java Server Pages (JSP).

The .NET version is implemented using the ASP.NET MVC 3 framework [133], a web application framework that implements the Model-View-Controller (MVC) pattern. For persistence we used the ADO.NET Entity Framework [132], i.e. a set of data-access APIs similar to JPA. It is commonly used by programmers to access and modify data stored in relational database systems, though it can also access data in non-relational sources. Data querying capabilities are provided via the .NET Language Integrated Query (LINQ) framework. The business logic is implemented in basic C#.

Table 2.1: Overview of the quantitative metrics of the size of the Java EE and .NET versions of the on-premise hotel booking application, expressed in lines of code (comments and empty lines not included). These metrics are used to evaluate the portability to the different PaaS platforms. Build and query scripts are not taken into account.

	Type	**# files**	**# lines of code**
Java EE	Java	28	1210
	JSP	13	490
	XML (config)	4	106
	HTML	3	19
	CSS	1	61
.NET	C#	37	1280
	ASP.NET web page	20	788
	XML (ASP.NET config)	6	118
	CSS	1	274

For the on-premise Java EE version we used the Eclipse development environment, the Glassfish application server (with TopLink SQL database), and JUnit as testing framework. The tool support for the .NET implementation consisted of Visual Studio 2010 (with Unit Testing Framework), the Internet Information Services (IIS) web server, and SQL Server Express.

2.2.3 Approach

The case study has been conducted in a systematic way by starting from a stable and operational implementation of the proposed enterprise application, offering a well-specified functionality as summarized above. For all the platforms that have been addressed, we have conducted the minimal but sufficient modifications to deliver exactly the same functionality of the corresponding enterprise application. We have reported systematically on our activities and we then had two independent scores by experts to express the quality of the underpinning PaaS environment.

First, we have migrated the on-premise application to a SaaS application on top of the different PaaS platforms, without taking into account multi-tenancy. After this first step, quantitative results with respect to portability have been gathered in the form of the amount of lines of code that have been changed compared to the on-premise version. This results into a score for the compatibility requirement. Furthermore, scores are assigned to each platform to express the quality of the provided integration support.

Next, the SaaS implementations on the different platforms have been adapted and extended (if necessary) to support application-level multi-tenancy. Subsequently, a qualitative score is assigned for each subrequirement of the multi-tenancy support (as described in Section 2.2.1). This score is mainly based on the availability of built-in support to address the specific requirement and to which degree. When a platform does not provide built-in support, we have validated whether it is possible to fulfill the requirement by means of a custom implementation, and we have assessed how much effort such custom implementation requires (in terms of lines of code).

Finally, the quality of the tool support associated with each platform has been assessed based on the hands-on experience with these tools during the entire development process. Again, scores have been generated for each individual subrequirement.

The details of the different case study implementations and results are discussed for each requirement separately in Sections 2.4, 2.5, and 2.6. In addition, with each new release of a platform that has been investigated, the implementations and scores have been updated (when necessary). These updates were logged to keep track of the evolution of the platforms. Relevant findings with respect to these new releases are discussed in Section 2.7.

2.3 Presentation of the PaaS platforms

We distinguished three categories to classify PaaS platforms with respect to their support of SaaS application development (Table 2.2). For each category we selected a mature and representative platform to compare these different categories, in particular Force.com [182, 219] (Category 3), Google App Engine [77] (Category 2), and Windows Azure [134] (Category 1). This section summarizes each of the PaaS platforms that have been selected. Furthermore, the specific setup and version that we used during the comparison are specified.

2.3.1 Force.com (Version 19.0 - 25.0) - *Category 3*

Force.com [182, 219] is a cloud computing platform that makes the core technologies behind Salesforce CRM [183] available for developing and deploying custom enterprise applications, and also extensions for the Salesforce CRM application. It provides a development environment to deliver business applications that run against the Salesforce.com database (data-centric) [11]. The targeted consumers are private enterprises and commercial software providers.

The key enabling technologies of the Force.com platform are multi-tenancy and metadata. Everything exposed to developers and application users is stored internally

Table 2.2: Classification of the major PaaS platforms into the three categories we distinguished. The majority of the current PaaS platforms belong to category 1, because they build on common application servers and middleware platforms and focus on migration of existing applications to the cloud.

	PaaS platforms	
Category 1	Amazon Elastic Beanstalk [5]	Oracle PaaS [159]
	Cloud Foundry [211]	Red Hat OpenShift [175]
	CloudBees [49]	SAP NetWeaver [184]
	Heroku [89]	*Windows Azure* [134]
	IBM SmartCloud [94]	WSO2 Stratos [17, 222]
Category 2	AppScale [43]	*Google App Engine* [77]
	GigaSpaces' XAP Elastic Application Platform [76]	
Category 3	Cordys PaaS [157]	TCS InstantApps [192]
	Force.com [182, 219]	WOLF [221]

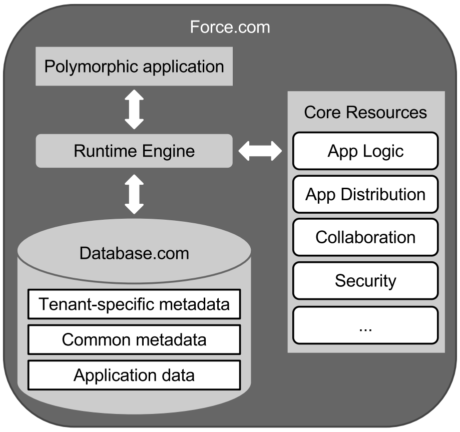


Figure 2.3: Force.com's metadata-driven architecture, based on [219]

as *metadata*. At runtime the required application components are generated by the runtime engine (kernel), based on these metadata [219]. These application components are combined into a polymorphic application, fulfilling the individual expectations of various tenants and their users [219]. As depicted in Fig. 2.3, the underpinning database clearly separates the metadata describing the base functionality of an application, the metadata corresponding to tenant-specific customizations, and the actual application data. Custom business logic and functionality can be written in Apex, a strongly typed, object-oriented programming language (part of App Logic in Fig. 2.3). Other cloud services provided by the Force.com platform, called *core resources*, are for instance AppExchange (App Distribution), a marketplace for cloud applications, and Chatter (Collaboration), a suite of collaboration and social features.

As is typical for PaaS platforms of category 3, Force.com applies a different approach and programming model compared to the traditonal middleware frameworks because of the metadata-driven architecture. It takes a more scripting-based approach using a dynamically interpreted language (i.e. Apex). We used the pure Force.com platform without third-party extensions for implementing the application case. The implementation is developed on versions 19.0 (Summer '10) till 25.0 (Summer '12) of the Force.com platform.

2.3.2 Google App Engine (Java SDK 1.3.6 - 1.6.0) - *Category 2*

Google App Engine (GAE) [77] is a PaaS plaform that enables the development and hosting of web applications on the same systems that power Google applications. It is targeted at traditional web applications, but it is less suitable for general-purpose computing [11]. This allows Google to enforce a clean separation between a stateless computation tier and a stateful storage tier. Furthermore, GAE supports automatic scaling and load balancing as the needs for traffic and data storage grow. To handle large and complex requests or to do continuous processing, GAE provides more powerful backend instances. Backends do not automatically scale based on the load and can retain the state of the application.

The following programming languages are supported:

- Standard Java technologies, including the JVM, Java servlets and the Java programming language, or any other language using a JVM-based interpreter or compiler (e.g. JavaScript, Ruby and Scala).
- The Python progamming language, including the Python standard library.

The Java and Python runtime environments are implemented as a secure sandbox where an application can run without interference from other applications, independent from the underlying hardware and operating system. In the rest of this paper the discussion is limited to the GAE's Java runtime environment.

A simplified overview of the GAE architecture is presented in Fig. 2.4. Incoming requests are routed to an application server by a load balancer. Clones of the application are automatically created when needed. The platform provides a variety of cloud services, such as the users and mail services, to support application development. At the lowest level a distributed datastore service provides schemaless object storage, a blobstore serves binary large objects (called *blobs*), and a memcache service offers a high performance in-memory key-value cache. The datastore consists of the Megastore

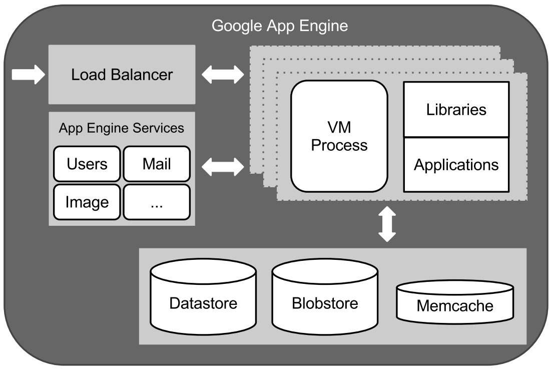


Figure 2.4: Overview of the Google App Engine architecture

storage system [19], an abstraction layer built on top of Bigtable [40]. It combines the scalability of a NoSQL datastore with the convenience of a traditional relational database. The datastore is divided into separately replicated partitions, with full ACID semantics within these partitions, but limited consistency guarantees across them [19]. In addition, only the scalable traditional relational database features are supported.

For this comparison we used GAE with Guice [78], Google's dependency injection framework, to add better modularity and customization capabilities to the platform. This allows us to develop the price calculation variation as a module that can dynamically be selected when needed. We used GAE SDKs 1.3.6 till 1.6.0 in this comparison.

2.3.3 Windows Azure (SDK 1.2 - 1.6, C#) - *Category 1*

The Windows Azure platform [134] is a cloud computing platform supporting the development and deployment of Windows applications that provide services to both businesses and consumers. These applications can be created using the .NET framework in languages such as C# and Visual Basic, or they can be built without the .NET framework in C++, Java, PHP, Ruby and Python. Windows Azure is originally a PaaS platform but since November 2010 it is also possible to deploy virtual machine (VM) images of Windows Server (2008), which is a form of IaaS. Notice that Windows Azure provides not necessarily an execution environment for all supported languages. For example, Java APIs are available for the different Azure services, but the developer has to install a JVM and an application server in an Azure VM himself.

The Windows Azure platform is a group of cloud technologies, each providing a specific set of services to application developers [41] (see Fig. 2.5):

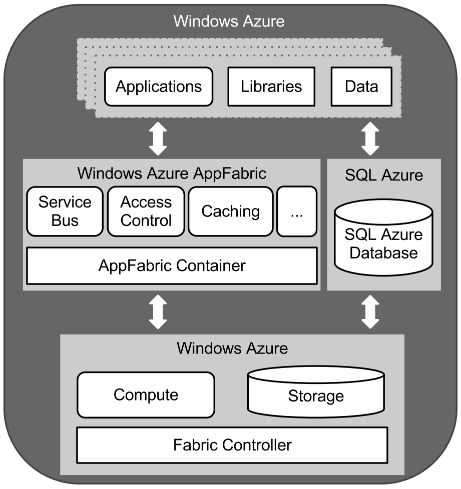


Figure 2.5: Overview of the Windows Azure architecture [41]

- *Windows Azure*: A Windows environment for running applications (*Compute*) and storing data (*Storage*) in the cloud. A Windows Azure application can be created using three kinds of compute service types, called *roles*: (i) a Web role for running web-based applications, (ii) a Worker role for (background) processing, and (iii) a VM role to run VM images. An application is structured as any combination of role instances. Web and worker role instances consist of a Windows Server 2008 R2 VM with the .NET 4.0 framework pre-installed. In a web role, there is also an IIS web server available.

  The storage service consists of blob storage (course-grained, unstructured data), table storage (set of entities with properties, NoSQL), and queues (asynchronous communication between different roles). All data is accessible via a REST API and the OData query language for tables. The Fabric Controller makes abstraction of the underlying infrastructure and offers a pool of processing power to the compute and storage services.

- *SQL Azure*: A cloud-based service for relational data, built on Microsoft SQL Server.

- *Windows Azure AppFabric*: A middleware platform on top of Windows Azure that provides a higher level of abstraction and reduces the complexity of cloud development [135]. The AppFabric Container provides a new programming model and runtime for cloud application development using .NET core languages [148]. The AppFabric Container itself is not publicly available and therefore not further discussed in this paper, but Microsoft offers several AppFabric Services on top of this container, for example caching. Recently, the AppFabric Services have been renamed to Windows Azure Features.

The Windows Azure platform provides an environment that strongly corresponds to a traditional on-premise setup (cf. Category 1), this especially applies to the Windows Azure Compute service and SQL Azure. In this comparison we focus on Windows Azure using traditional .NET frameworks in the C# programming language and the Ninject dependency injection framework [102]. We also include SQL Azure into the comparison. The case study application is implemented using Azure SDKs 1.2 - 1.6.

2.4 Portability

The main driver behind portability over PaaS platforms is the need (by SaaS providers) for support to facilitate the transition from on-premise towards the SaaS-based deployment models. More specifically, we have identified the following subrequirements: (i) compatibility with the mainstream programming model and middleware frameworks, and (ii) easy integration with cloud services, platform-specific as well as external.

The comparison of the portability support is structured as follows. We describe for each platform first the compatibility of the programming model and then the integration support. The former subrequirement discusses a) the platform-provided programming model and middleware, and b) the extensibility of the platform. The latter consists of integration with platform-specific services, and external (cloud) services and applications. Finally we give a brief summary of our findings regarding portability. Table 2.3 presents an overview of the portability support on the three PaaS platforms.

2.4.1 Force.com

The programming model of the Force.com platform is incompatible with the on-premise model of the hotel booking application. However, it provides extensive integration facilities with external applications and services on different (cloud) platforms and using other programming languages.

Compatibility with mainstream technology

The metadata-driven architecture and the Salesforce.com-specific Apex language are incompatible with common programming models and middleware frameworks and therefore hinder portability across different platforms. Concretely, this means none of the implementation artifacts of the on-premise versions can be reused. A full rewrite of the application is required. Due to this lack of compatibility with

Table 2.3: Overview of portability support on the three investigated PaaS platforms

	Compatibility	**Integration support**
Force.com	Not compatible: • Metadata-driven (e.g. data model) • Apex language (Force.com-specific)	• Apex integration with core resources (e.g. database) • Web Services & REST API • Including utilities (e.g. APIs) for different programming languages and platforms
GAE	Limited compatible: • White list of Java classes (e.g. no EJB or JAX-WS servers) • JPA: limited + GAE-specific code • Extensible: REST, Guice... (support limited by white list)	• Datastore (Bigtable) via Megastore • Google Apps services • URL Fetch (low-level) • Remote API (GAE-specific)
Azure	Fully compatible: • Common .NET frameworks (MVC, WCF, Unity...) • Extensible by importing libraries • ADO.NET EF: only SQL Azure	• Platform-provided services via common .NET frameworks • Azure Table Storage: no support • Web Services & REST API • Service Bus (AppFabric)

traditional programming models, it is also not possible to (re)use existing middleware frameworks on top of the Force.com platform. Such middleware support should be provided by the platform itself, or be specifically written for Force.com and made (publicly) available, for instance via the AppExchange marketplace. First, we describe the programming model used by Force.com. Next, the Apex language is introduced and illustrated.

Business objects. Force.com focuses on building data-centric business applications and this influences the way the application is designed. The developer has to reason in terms of *business objects (BOs)*. Business objects represent the data that will be stored in the database. In contrast to for example entities in Java EE, BOs cannot contain any behavior. Moreover, they are not specified in a programming language (code), but have to be created via a *point-and-click web interface* and are stored as metadata in XML (Listing 2.1).

Relationships [42] between business objects are created by specifying relationship fields that map records in one object to records in another. Force.com has two types of relationship fields:

- The *Lookup Relationship* is used to create one-to-one and one-to-many relationships between objects, without a direct dependency.
- The *Master-Detail Relationship* creates a tight parent-child relationship between two objects. The relationship field is required on all child records and deletes are cascaded from the parent to the child. This relationship type also enables the creation of many-to-many relationships through the use of a junction object.

In the case study implementation on top of Force.com we used both relationship types. When a travel agency removes a hotel from the online hotel booking application, all rooms and bookings related to that hotel should also be deleted. Therefore the application needs master-detail relationships: rooms are details of a hotel, while bookings are details of room. However, the master-detail relationship requires that all relationship fields are set at the moment of object creation. This prevents creating tentative bookings that are not yet added to a room. To solve this issue, we had to extend our data model with an `Invoice` that manages all tentative bookings of a specific customer during one session (via a master-detail relationship), and stores the total price of these bookings in the invoice object. An invoice also has a status field, and only when the status of the invoice changes to 'closed' a booking is considered finalised. Between a `Hotel` and an `Address` we used a lookup relationship, as shown on lines 31-39 in Listing 2.1.

Listing 2.1: A fragment of the raw metadata of a simple business object (BO) representing the address of a hotel. In total it consists of 100 lines of XML. Clearly, implementing even a simple BO is not recommended using this metadata representation.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<CustomObject xmlns="http://soap.sforce.com/2006/04/metadata">
    ...
    <label>Address</label>
    <pluralLabel>Addresses</pluralLabel>
    <description>Address of a hotel.</description>
    <fields>
        <fullName>City__c</fullName>
        <externalId>false</externalId>
        <label>City</label>
        <length>25</length>
        <required>true</required>
        <type>Text</type>
        <unique>false</unique>
    </fields>
    <fields>
        <fullName>Nb__c</fullName>
        <description>House number.</description>
        ...
    </fields>
    <fields>
        <fullName>Postal_Code__c</fullName>
        <description>Postal code</description>
        ...
    </fields>
    <fields>
        <fullName>Street__c</fullName>
        <description>Name of the street.</description>
        ...
    </fields>
    <fields>
        <fullName>Hotel__c</fullName>
        <externalId>false</externalId>
        <label>Hotel</label>
        <referenceTo>Hotel__c</referenceTo>
        <relationshipLabel>Addresses</relationshipLabel>
        <relationshipName>Addresses</relationshipName>
        <type>Lookup</type>
    </fields>
    ...
</CustomObject>
```

Apex. The *Apex language* is used to create, persist and update instantiations of business objects, and to query the database via a built-in query language. When a business object is materialized in the Apex language, it is called an *sObject*. `sObject` is also the generic abstract type, comparable with `Object` in Java. Furthermore the Apex language is used to implement controller classes for the user interface, to invoke external web services, and to expose operations in a web service.

Queries on the Force.com database are expressed in the *Salesforce Object Query*

Language (SOQL). SOQL is an object query language (cf. JPQL for Java Persistence) that is integrated in the Apex language and uses the abstraction provided by the business objects and their relationships to navigate through the application data. The syntax is similar to the one of SQL, but it does not support all advanced features such as joins, wildcards or calculation expressions. Besides SOQL, Force.com also supports the *Salesforce Object Search Language (SOSL)* which enables text searches across all persisted objects.

Since a business object itself cannot validate input data, validation rules need to be specified separately. The validation rules consist of mathematical formulas that allow to compare input data with other data (predefined thresholds or data from other BOs). For instance, a validation rule checks whether the end date of a booking period is after the start date. However, these mathematical formulas are not powerful enough to perform all necessary validation checks. For example, it is not possible to define a loop that iterates over multiple BOs (in a list). This is required in the case study to check the availability of a room (by iterating over the different bookings of the room): a room may only be booked when there is no other booking in the same time period.

To implement more complex business logic and validation rules like checking the availability of a room, the Apex language needs to be used. By means of database triggers, pieces of Apex code are executed before or after events occur on an `sObject`, for instance the insertion of a BO. Apex triggers enable developers to change BOs when such events occur, or to perform complex validation checks.

Listing 2.2: Apex trigger to check the booking period of a new Booking. It illustrates the use of the Apex language and other aspects of the Force.com programming model. Custom `sObjects` are represented by their `sObject` name with '`__c`' appended to the end as a suffix, for instance `Booking__c`. Relationship names in SOQL queries consist of the related object name followed by '`__r`', e.g. `Room__r`.

```
trigger CheckBookingPeriod on Booking__c (before insert,
                                          before update) {
 for(Booking__c triggeredBooking : Trigger.new) {
  // Retrieve all finalized bookings
  List<Booking__c> finalizedBookings =
   [ SELECT b.Id, b.Start__c, b.End__c FROM Booking__c b
     WHERE b.Invoice__r.Status__c = 'Closed'
     AND b.Room__r.Id = :triggeredBooking.Room__c ];

  // Check for overlap
  for(Booking__c b : finalizedBookings) {
   if(b.End__c < triggeredBooking.Start__c
       || b.Start__c > triggeredBooking.End__c) {
        continue;
   } else {
        triggeredBooking.Start__c.addError(
          'Room may not be booked in given period.');
}}}}
```

For example, `CheckBookingPeriod` (Listing 2.2) is an Apex trigger in our case study implementation to check the availability of a room. For each newly inserted or updated booking, this trigger first retrieves via a SOQL query all finalized bookings for the same room as the triggered booking (lines 4-7). Next, the trigger checks whether there is an overlap between the booking period of the triggered booking and the finalized bookings (lines 10-18). An overlap will result in an error (lines 15-16). When no overlap is detected, the execution of the normal insert (or update) operation will continue (line 13).

Integration

The Force.com platform provides extensive integration capabilities. First of all, it offers a tight integration between the Apex language and the core resources (platform-provided services), for example the database. Apex is used for creating, modifying and querying database objects (via SOQL) and to write database triggers. Furthermore the language can be directly applied to implement user interface controllers, unit tests, and web services. Thanks to this integration with Apex, developers can use these facilities out of the box without having to write additional (integration) code.

Next, the platform provides support for integration with external applications and services that use other programming languages and platforms. Developers can create and expose SOAP-based web services using Apex, or invoke external SOAP- and REST-based web services from the platform. In addition, the Force.com Web Services and REST APIs provide direct access to the Force.com data from an external system. These APIs are fundamental integration points, allowing access via clients on any platform. To ease the integration with these external (client) applications, software utilities (called toolkits) are built around these core APIs. There exist toolkits for (i) programming languages such as Java, .NET and Ruby, (ii) cloud platforms such as Amazon Web Services [4] (S3 and EC2), Google App Engine [77] and Windows Azure [134], and (iii) mobile platforms like Android and iOS. Concretely, they support native development approaches with using the Force.com Web Services and REST APIs. Furthermore the Force.com database is also available as a standalone service (Database.com) using the same toolkits. This enhances the reusability of applications, data and services outside the Force.com environment. For example, a new (version of a) SaaS application can be developed on a different platform without having to port all data to the other platform.

Finally, additional integration facilities are available in the AppExchange marketplace, for instance ERP connectors and data integration solutions.

2.4.2 Google App Engine

Google App Engine (GAE) supports Java and other JVM-related languages and is extensible with common middleware frameworks. But this compatibility has its restrictions, limiting the portability across different Java-based platforms. Furthermore, the platform-provided cloud services are well supported. Integration with external services, however, is limited.

Compatibility with mainstream technology

In contrast to Force.com, GAE is compatible with the common programming model used in the on-premise case study implementation by supporting Java and other JVM-related languages, which facilitates the portability enormously. However, this compatibility is limited: to maintain scalability a number of restrictions are imposed, and some technologies and APIs are completely or partially unsupported by the underpinning platform. This results in adaptations to the application specifically for GAE.

In addition, GAE can be extended with other middleware frameworks, for instance Restlet [152] for RESTful web services. To use these frameworks additional (external) libraries have to be imported into the application. However, the imposed restrictions and incompatibilities also limit this extensibility. The result is that many middleware extensions do not work on top of GAE or require changes to the source code, which hinders an easy migration. For example, Restlet offers a separate edition, specifically adapted to fully work with GAE.

Major parts of the data and web tier of the Java EE version could be reused (Table 2.4). However, the business tier consisting of EJBs is not supported by GAE and therefore this logic is integrated into the web tier. The other code changes in the data and web tier are related to persistence. The different restrictions and incompatibilities imposed by GAE as well as the differences with respect to persistence are discussed in further detail in the following paragraphs.

Restrictions. A white list defines which Java classes of the JRE are allowed to be used on the platform [79]. For example, the APIs to write to the filesystem or to access the network are not available. Applications should use the provided datastore and URL Fetch services. Also the use of threads or other kinds of system calls are not allowed. Moreover web requests should be handled (in a single process) within 60 seconds. Processes that take longer to respond are terminated to avoid overloading of the web server [77]. Actually, GAE is optimized for applications with short-lived requests that take a few hundred milliseconds. These restrictions limit the options

Table 2.4: Reuse of code (in terms of lines of code) between Java EE version and GAE implementation. The last column indicates the ratio of the reusable code compared to the full GAE implementation.

	# common files	**# common LoC**	**% reuse of Java EE**	**% LoC of GAE**
Data tier	11	555	91.74%	91.43%
Business tier	0	0	0.00%	/
Web tier	28	916	90.42%	76.21%
Total	**39**	**1471**	**78.00%**	**81.32%**

of the developer, but are not insurmountable: the restrictions and their alternatives need to be taken into account during application design. For example, the execution of long-running processes can be moved to backend GAE instances.

Unsupported. GAE offers built-in support for only a limited set of technologies. Among other things, this is due to the imposed restrictions. The web page "Will it play in Java?" [80] lists the level of compatibility with various Java technologies and frameworks. GAE's main focus is on web applications using Java Servlets and Java Server Pages (JSP), which have not a developer-friendly API to develop against. Enterprise Java Beans (EJB) and JAX-WS (SOAP Web Services) for servers are examples of two common technologies for enterprise applications that are not supported. Other technologies are only partially supported, for example JPA, the persistence API we used in the Java EE version of the online booking application.

Persistence. As explained in Section 2.3, GAE's datastore is based on Bigtable [40], a NoSQL database, while JPA is a standard interface for interacting with relational databases. To support JPA the Megastore abstraction layer [19] provides an object-relational mapping from RDBMS to NoSQL, though some features are not supported by this mapping, such as 'Join'-queries and aggregation queries. Therefore it is not possible to perform JPQL queries that use those more advanced features. An alternative for JPA is the Java Data Objects (JDO) persistence interface. JDO is independent of the underlying database and therefore a better choice when building a new GAE application.

However, independently from the used persistence interface, problems remain with the use of primary keys for entities. In common practice a primary key is a generated `Long` integer. Google App Engine, however, requires that in an entity relationship the children objects have a primary key that represents the parent object. Types that support such primary keys are a `Key` class, or a `Key` value encoded as a string. This

Key class is a Google-specific class, which hinders portability. Listing 2.3 and 2.4 show a small entity representing the address of a hotel (child-parent relationship), respectively using default JPA and the version in GAE. Because the primary key refers to the parent object (i.e. the hotel), each child object can have only one parent. This limits the use of relationships and requires the data model to be structured in an hierarchical way (e.g. bidirectional bindings are not allowed).

Listing 2.3: Address entity using default JPA.

```


@Entity
public class Address {

  @Id
  @GeneratedValue(strategy=
           GenerationType.AUTO)
  private Long id;
  private String street;
  private int number;
  private int postalCode;
  private String city;
  private String country;

  public Address() {
    ...
  }
  ...
}
```

Listing 2.4: Address entity in GAE (JPA).

```
import com.google.appengine.
                  api.datastore.Key;
@Entity
public class Address {

  @Id
  @GeneratedValue(strategy=
           GenerationType.IDENTITY)
  private Key id;
  private String street;
  private int number;
  private int postalCode;
  private String city;
  private String country;

  public Address() {
    ...
  }
  ...
}
```

Finally, it is not possible to use the @PersistenceContext annotation to declare a dependency to an entity manager, which is used to create, remove and query persistent entity instances. The developer has to manually create an instance of the EntityManager when needed and close it afterwards, instead of it being managed by the underpinning middleware.

Integration

While Force.com offers extensive integration support for platform-provided as well as external (cloud) services, Google App Engine clearly focuses on the integration of its own cloud services. The main platform-provided cloud services in GAE are the storage and caching services. Each of these services have an API that supports the integration with the application, often based on a standard, for example JPA, JDO, and JCache, a proposed interface standard for memory caches. The Megastore abstraction layer [19] provides the integration between the Bigtable datastore [40] and the persistence interfaces by means of an object-relational mapping, though with some incompatibilities. Furthermore, several Google Apps services are well integrated into Google App Engine, for example Google Accounts for authentication via the User service API and Gmail to send emails via the Mail service API.

Built-in support for integration with external applications or resources, however, is limited. GAE applications can use the URL Fetch service, allowing HTTP(S) calls to other services (via low-level connections and streams). In addition, SOAP-based web services can be accessed by implementing a client using JAX-WS. Except for the Java Servlet API, GAE does not offer built-in support to make GAE applications accessible from external services and applications. To provide or call a RESTful web service, the SaaS provider has to import additional libraries, for example Jersey [158] and Restlet [152]. The latter even requires a GAE-specific library with limited features. Further, to develop a SOAP-based web service on top of GAE, SaaS developers have to directly use javax.xml and JAX-B, and implement their own handlers for SOAP messages. Finally, there exists a remote API to access the datastore of a GAE application remotely, but this requires the installation of the GAE SDK and it offers no standardized interface.

2.4.3 Windows Azure

Windows Azure is fully compatible with the existing .NET programming model and frameworks. In addition, many options are available for integration with external services and local as well as cloud applications. However, extensive support for integration with the Azure Table Storage is lacking.

Compatibility with mainstream technology

In contrast to the white list of available classes in GAE, Windows Azure provides a full .NET stack, directly supporting commonly used .NET frameworks, such as the WCF framework [137] and ASP.NET MVC [133]. Thus, the programming model used for the .NET version of the case study implementation can be reused on Windows Azure without any compatibility issues, except for persistence in combination with Azure Table Storage. Also no restrictions like writing to the file system are imposed, although this should be done with caution in a cloud environment.

Furthermore, the platform can be extended with any third-party .NET framework, as long as this can be achieved by importing libraries. If an installation is required to extend the platform, then a VM role should be used, because installing is not allowed in the web and worker roles. However, in this case the SaaS provider will have to manage and maintain the whole stack himself.

When using SQL Azure, 100% of the on-premise .NET implementation could be reused (Table 2.5). Only some additional configuration files were required to use the application on top of Azure. In the case that Azure Table Storage is used, less code of the on-premise application can be reused. The majority of changes are related to the data tier, where a custom (and probably not reusable) persistence layer had to be developed to integrate with the Azure Table Storage. This issue with the Azure Table Storage is explained in depth below.

Table 2.5: Reuse of code (in lines of code) between .NET version and Azure implementations. The last column indicates the ratio of the reusable code compared to the full Azure implementations.

	# common files	**# common LoC**	**% reuse of .NET**	**% LoC of Azure**
SQL Azure				
Data tier	23	584	100.00%	100.00%
Business tier	8	292	100.00%	100.00%
Web tier	33	1584	100.00%	94.45%
Total	**64**	**2460**	**100.00%**	**96.36%**
Azure Table				
Data tier	21	512	87.67%	47.50%
Business tier	8	247	84.59%	76.00%
Web tier	33	1572	99.24%	93.96%
Total	**62**	**2331**	**97.76%**	**75.78%**

Persistence. For persistence the SaaS provider has the choice between the Azure cloud storage (providing blobs, tables and queues), or SQL Azure (a relational database deployed in the cloud). The latter option supports a transparent migration: applications can access data stored in SQL Azure using the ADO.NET Entity Framework, in the same way as accessing SQL Server locally. The major difference with the local version is that the SQL Azure database is located outside the SaaS provider's boundaries. Therefore we have to change the connection string. In Listing 2.5 the implementation of an `Address` entity is presented. While in Force.com validation rules for BOs are separately defined, the ADO.NET Entity Framework allows developers to add validation rules as annotations.

On the other hand, the Azure cloud storage provides a schemaless persistence service with a much higher scalability than SQL Azure (comparable with Google's Bigtable). As Google App Engine does inherently, the Azure Table Storage can be used as

Listing 2.5: Address entity using the ADO.NET Entity Framework.

```
1  public class Address
2  {
3      [Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]
4      public int Id { get; set; }
5      [StringLength(30, MinimumLength=3)]
6      public string Street { get; set; }
7      [Range(1, 9999)]
8      public int Number { get; set; }
9      [Range(1, 999999)]
10     public int PostalCode { get; set; }
11     [StringLength(25, MinimumLength=3)]
12     public string City { get; set; }
13     [StringLength(15, MinimumLength=3)]
14     public string Country { get; set; }
15 }
```

an alternative for SQL Azure. However, the ADO.NET Entity Framework (EF) is not compatible with the Azure Table Storage, because it is an object-relational mapping (ORM) framework. Therefore, there exists no programming support for domain entities and entity relationships when using the Azure Table Storage. In our implementation we decided to represent an `Address` entity the same way as in Listing 2.5, but without a (generated) identifier (without lines 3-4). The validation rules can be used by the MVC framework to do input validation.

Integration

Windows Azure offers extensive integration facilities for platform-provided services as well as external services and applications. The main platform-provided services are the storage services and the AppFabric services (i.e. service bus, access control and caching) [135]. The programming model for these services is often based on common .NET frameworks, such as the ADO.NET Entity Framework (for SQL Azure) and WCF [137], which enables easy integration. The caching service provides an API that is comparable to the Memcache service of GAE, and can be used after limited configuration changes. However, the Azure Table Storage is not well integrated in the programming model. Integration with external services and applications is provisioned by the Service Bus by AppFabric and by the use of SOAP- and REST-based web services. In the following paragraphs we discuss in depth the issues with the Azure Table Storage, and integration with external services and applications.

Azure Table Storage. GAE hides the complexity of the Bigtable datastore by providing a mapping from RDBMS (JPA) to NoSQL (Bigtable). Windows Azure, however, does not offer this integration support for its table storage and this results in additional complexity for the developer (cf. Table 2.5). Developers have to manually map the (self-defined) implementation of domain entities to a model that can be stored in the Azure Table Storage. This is especially complex when the domain model contains many entity relationships.

Tables in Azure can contain different types of entities and each entity should have a unique primary key. This key consists of a partition key and a row key. The former defines in which partition an entity is stored. Different partitions can be stored on different servers. The row key should be unique within one partition. However, to improve searching for specific entities those keys should be built using well-known information (e.g. non-mutable properties of the entity) instead of generated unique IDs. The only advantage of this complex interface is that it provides developers more control to optimize data storage for specific queries.

Another approach to improve searching is the denormalization of data: data can be stored in multiple tables to facilitate specific search queries. However this complicates the updating of the data enormously: when updating data in multiple tables, consistency should be preserved.

For the implementation of our application case on top of Windows Azure using the table storage, we distinguished two options: (i) store all entities of the same type in one partition, or (ii) use one partition per hotel (together with its address, rooms and bookings). The first option is the easiest to implement: the entity type is used as partition key, and in the row keys of the rooms and bookings we integrate the hotel name and room number to represent the entity relationships. The second option offers a better performance to query and update the rooms and bookings of a particular hotel. These entities are all stored in one partition, and thus require only one transaction. The choice between these or other options depends on which queries are most frequently used. In case of a small set of data, all entities can be stored in a single partition.

We decided to apply the first option for our implementation. Because each hotel has an address, the `Hotel` and `Address` entities are stored in one row of the table for hotels. This `HotelRow` (Listing 2.6) inherits from `TableServiceEntity` to indicate it is an entity in the Azure Table Storage. For the other domain entities we made similar `Row` classes. Based on [136], we developed a custom persistence layer for the mapping from our domain entities (e.g. the `Address` in Listing 2.5) and entity relationships to these lower-level entities in the Azure Table Storage.

External services and applications. Access to and from external services and applications is possible via SOAP- and REST-based web services. For example, using the WCF framework the developers are able to expose the SaaS application to external applications. Furthermore, the Azure cloud storage provides a RESTful

Listing 2.6: Hotel entity in the Azure table storage.

```
public class HotelRow : TableServiceEntity
{
    public string HotelName { get; set; }
    public string Street { get; set; }
    public int Number { get; set; }
    public int PostalCode { get; set; }
    public string Place { get; set; }
    public string Country { get; set; }
}
```

API to enable direct access to data from external systems. As with Force.com, this enhances reusability of applications, data, and services outside Windows Azure.

In addition, the Service Bus is an AppFabric service that provides secure messaging and connectivity capabilities, enabling integration between different applications in the cloud as well as locally. It supports various communication and messaging protocols as well as patterns, ensuring reliable and scalable message delivery. For instance, the Service Bus can be used to create an hybrid application, composing Azure applications with internal components. However, this service is only available at an additional cost. Other announced AppFabric services that support the integration requirement are Integration and Composite App [135, 148].

2.4.4 Summary

Compared to Google App Engine and Force.com, Windows Azure offers the best portability support. It is fully compatible with the .NET programming model and frameworks, and also offers extensive integration support. Only the support for the Azure Table Storage is limited. For instance in the case of our hotel booking application, the use of SQL Azure is certainly recommended, because the domain model contains too many relationships.

Force.com is incompatible with the mainstream programming models used in the on-premise implementations of the application case. However, it copes with this issue by providing a wide set of integration capabilities with other programming languages, applications and (cloud) platforms. This enables the decomposition of applications: an application should not be entirely developed on top of Force.com, but it can be integrated with components on other platforms, locally as well as in the cloud.

While Force.com clearly focuses on business applications with its good built-in integration support, Google App Engine aims to facilitate development with good compatibility and to hide complexity. This compatibility, however, has its limitations. Integration support is especially focused on Google's own services, and does not provide built-in support to create SOAP- or REST-based web services (in contrast to Force.com and Azure).

2.5 Multi-tenancy

Multi-tenancy is an important enabler for SaaS applications to achieve better operational efficiency [44, 84]. This section has the following structure for the discussion of each platform. First we describe the general approach applied by the platform to provide multi-tenancy support. Next we discuss for each platform how it addresses the three identified subrequirements, i.e. data isolation, tenant-specific configuration and customization, and tenant-specific application management. Finally we summarize the main conclusions regarding multi-tenancy support by the three investigated PaaS platforms. Table 2.6 gives an overview of the multi-tenancy support provided by the different investigated PaaS platforms.

2.5.1 Force.com

Force.com offers built-in support for data isolation as well as tenant-specific configuration and customization by providing each tenant with a separate environment. In addition, it supports the tracking of tenant usage via the License Management Application (LMA).

General approach

Development and deployment (production) on Force.com typically takes place in an environment or organization, frequently shortened to *org* [90]. Such an environment provides a number of features for applications within that environment and ensures all tenant-specific data (including metadata) are separated from other orgs. The Force.com platform offers two ways to deliver applications depending on who the SaaS consumers are:

- *Single Org* model: the application runs in the org of the SaaS provider. This model is especially applied by organizations to build applications for internal use, only accessible by the own employees (i.e. SaaS provider and SaaS consumer are the same organization).
- *Distributed Org* model: the application is packaged and installed by SaaS consumers in their own Force.com environment (or org). This model is suitable for developing, running and distributing a multi-tenant SaaS application.

Table 2.6: Overview of multi-tenancy support in the three PaaS platforms

	Data isolation	**Tenant-specific configuration & customization**	**Tenant-specific application management**
Force.com	Every tenant has its own Force.com environment (org)	• App Distribution • Extension packages • Custom objects and code	License Management Application (LMA)
GAE	• Namespaces API • Manual mapping of requests to namespace	• Dependency Injection (Guice Provider) • Manual configuration management	n/a
Azure	No support: • Manual data isolation • No built-in tenant context • Manual mapping of requests to tenant context	• Only if data isolation is ensured • Dependency Injection (Ninject Provider) • Manual configuration management	n/a

Since we want to offer the hotel booking application as a multi-tenant SaaS application, only the distributed org model is relevant for this paper. All components that make up the application (i.e. application metadata) are bundled together in a package. This package can be made available to the Force.com market via AppExchange (part of the App Distribution core resource). This is a central directory service that enables tenants to find and install applications.

Packages exist in two forms. The unmanaged package form is intended for the distribution of applications or components, together with the source code. By installing an unmanaged package the tenant receives its own copy, which can be modified independently from the creator of the package. Managed packages, however, obfuscate most of the source code and permit package creators (i.e. SaaS providers) to offer upgrades.

Data isolation

By providing each tenant with a separate environment or org, Force.com inherently supports data isolation. When a tenant installs a reference to an application in its org (e.g. the hotel booking application), the application metadata is copied into its org. This application metadata is separately stored in the multi-tenant database, together with the tenant-specific metadata (configuration and customizations) and the application data (see Fig. 2.3).

Each org also specifies the different users and user groups of the tenant, as well as their roles and permissions. Login names of users are automatically mapped to a specific tenant and org. The creator and owner of the org receives the tenant administrator role, responsible for the management of the installed applications (e.g. configuration) and user management.

Tenant-specific configuration & customization

The metadata-driven architecture of the Force.com platform enables tenants to configure and customize the application to fit their specific requirements. To offer a customizable SaaS application as described in the case study, the SaaS provider can provide extension packages. These enable the creation of new functionality on top of a base managed package. Extension packages can also be managed or unmanaged. By means of these extension packages SaaS providers are able to offer multiple variants of their application. In the implementation of our hotel booking application, the price calculation strategy based on the customer profile can be implemented as an extension package. The tenant administrator can configure the application by selecting and installing the necessary extension packages.

In addition, the tenant administrator can further tailor the application by adding custom data objects and Apex code (i.e. tenant-specific metadata), or by integrating the application with other applications installed into his org. As mentioned above, managed packages obfuscate most of the source code. Therefore tenant-specific customizations are limited to a class of changes that do not hinder seamless upgrades (by the SaaS provider) from occuring.

The Force.com platform almost transparently supports the development and deployment of multi-tenant applications via the App Distribution. However, the SaaS provider has to take several trade-offs into account. Depending on which features of the platform are used in the application, tenants should have a specific Force.com edition. This can be partially solved by offering different versions via extension packages. In addition, some features cannot be packaged (yet) and require manual configuration by the tenant administrator.

Tenant-specific application management

The License Management Application (LMA) of Force.com facilitates tracking of installation and upgrades of the managed packages created by the SaaS provider. The LMA enables the SaaS provider to apply licensing on managed packages to control how many end users in a tenant organization can access the package and for how long. LMA is available as an application in AppExchange. Force.com, however, does not support tenant-specific performance and resource consumption monitoring.

2.5.2 Google App Engine

Built-in multi-tenancy support provided by Google App Engine (GAE) is limited to data isolation. Tenant-specific configuration and customization can be achieved by using dependency injection (as explained below), but this requires additional implementation work by the SaaS provider. GAE also lacks support for tenant-specific application management.

General approach

To support the development of multi-tenant applications, GAE provides the *Namespaces API*. A *namespace* is set globally for each tenant using the namespace manager. More specifically, the namespace corresponds to a unique tenant ID and represents the context containing the information of the tenant linked to the current request. The GAE APIs that support namespaces will automatically use this current namespace (e.g. datastore and memcache). However, it is still possible to change the current namespace locally while processing a request. Wrong usage can lead to unintended data leaks between different tenants.

Data isolation

The Namespaces API automatically supports the partitioning of data across tenants by specifying a unique namespace for each tenant. Thus, by assigning a unique namespace to each tenant (e.g. the tenant ID), data isolation is preserved. Internally, the datastore and memcache store, update and query data based on the current namespace. Global data, accessible by all tenants, can be stored in a separate namespace. However, the mapping of an incoming request to a specific namespace does not occur automatically. A practical solution is presented in the following paragraph.

In addition, GAE lacks support for proper user management of a multi-tenant application, for example to assign the tenant administrator role. The Google Accounts and the Users service are not sufficient to distinguish the tenant administrator from normal end users. Separate roles only exist for the SaaS provider (administrator and developer). Therefore SaaS developers should develop their own user management system (as part of the application) to allow the tenant administrator to configure the application. This is certainly necessary when access control is an important requirement.

Mapping to namespace. For each request we have to retrieve the tenant ID and set the namespace for that request, ensuring that the request will be processed in the context of that tenant (tenant-specific data and configuration). For the implementation of the hotel booking application on top of GAE, we determine the tenant ID based on the used Google Apps domain (URL) to access the SaaS application. Applications can be deployed to any Internet domain owned by the SaaS provider and registered in his Google Apps account. Each domain then corresponds to a namespace.

Incoming requests with a specific domain name (i.e. tenant ID) should be mapped to a namespace. Therefore we use Java Servlet Filters (cf. [77]). A filter dynamically intercepts requests and responses, enabling the inspection or transformation of the contained information. For every request our `TenantFilter` (Listing 2.7) checks whether the namespace is already set. If not, the tenant ID is retrieved from the Google Apps domain (line 17). Next, the namespace is set to this tenant ID (line 19).

Listing 2.7: TenantFilter to set the current namespace.

```java
@Singleton
public final class TenantFilter implements Filter {
 // The filter config with initialization data.
 private FilterConfig filterConfig = null;
 // init method is called by the container when filter is instantiated.
 public void init(FilterConfig filterConfig) throws ServletException {
  this.filterConfig = filterConfig;
 }
 public void destroy() {
  this.filterConfig = null;
 }
 public void doFilter(ServletRequest request, ServletResponse response,
    FilterChain chain) throws IOException, ServletException {
  // Only when namespace is not set.
  if (NamespaceManager.get() == null) {
    // retrieve tenant ID (Google Apps domain name)
    String tenantID = NamespaceManager.getGoogleAppsNamespace();
    // set namespace
    NamespaceManager.set(tenantID);
  }
  // invoke next filter (if available)
  chain.doFilter(request, response);
}}
```

Tenant-specific configuration & customization

Google App Engine does not support tenant-specific configuration and customization of applications. In our implementation, we achieved it by using Google's dependency injection framework, Guice (v3.0) [78]. We discuss this in depth in the following paragraph. The tenant-specific configuration settings are stored as part of the tenant data. However, in contrast to Force.com, there is no support for tenant administrators to manage this configuration themselves (via a configuration interface). Similarly, GAE lacks support for the SaaS provider to develop and manage the different features of an application. Both require additional development by the SaaS provider.

Tenant-aware dependency injection. *Dependency injection (DI)* is a design pattern where component dependencies are managed by an injector, rather than relying on the component itself to fill in the dependency [66]. By using such a DI framework [66] on top of GAE, we are able to support tenant-specific customization. Instead of instantiating the feature implementations directly in the application, the flow of control is inverted: the life cycle management of feature implementations is controlled by a dependency injector. This injector binds dependencies in the application to a specific implementation component. In the case study, this means that the dependency of the hotel booking application on `IPriceCalculatorStrategy` is bound to one of the available feature implementations (cf. `DefaultPriceCalculator` and `CustomerProfileBasedPriceCalculator` in Fig. 2.2).

Originally the configuration of a GAE application is stored in the `web.xml` file, which is shared by all tenants. By means of Guice, the configuration can be dynamically loaded. However, the dependency injector is shared by all tenants. This hinders the isolated execution of tenant-specific customizations: all dependencies are set globally and any modification would affect all tenants. This is a general problem with dependency injection because it does not support activation scopes.

To solve this issue, we added an extra level of indirection. Instead of injecting features, we inject a `Provider` for that feature. This way the servlets have a dependency to a provider of a feature instead of to the feature itself. However, the customizations that can be performed this way are limited to switching between implementations of an interface or abstract class. Listing 2.8 presents the `PriceCalculatorProvider` that decides based on the tenant-specific configuration which feature implementation of the `IPriceCalculatorStrategy` feature should be selected: normal price calculation or with discounts based on the customer profile (lines 19-21). This feature implementation is then instantiated (line 29). If no tenant-specific configuration for this feature is set, the default option is retrieved and instantiated (lines 24-27). The instantiated tenant-specific feature implementation is stored in the (namespace-enabled) Memcache to improve performance.

Tenant-specific application management

GAE applications can only be monitored on a per application basis, which is too coarse-grained for multi-tenant SaaS applications. The Java SDK of GAE includes a suite of tools for measuring the performance of applications, called Appstats. Appstats integrates with the application using a servlet filter to record events, and provides a web-based administrative interface for browsing statistics. To enable tenant-specific monitoring, the SaaS provider will have to manually filter and record the incoming requests per tenant, as far as this is possible. Additional implementation is also required to support billing and to track which features are used by the different tenants.

2.5.3 Windows Azure

Windows Azure offers no support for multi-tenancy. SaaS providers will have to provide this themselves. If data isolation can be ensured, tenant-specific configuration and customization can be achieved, for example by using dependency injection (cf. Google App Engine).

Listing 2.8: Provider to retrieve the tenant-specific IPriceCalculatorStrategy feature implementation.

```
import com.google.appengine.api.memcache.MemcacheService;
import com.google.appengine.api.memcache.MemcacheServiceFactory;
import com.google.inject.Provider;

public class PriceCalculatorProvider
               implements Provider<IPriceCalculatorStrategy> {

 private String bindingType;
 private MemcacheService cache;

 public PriceCalculatorProvider() {
  this.bindingType = IPriceCalculatorStrategy.class.getName();
  this.cache = MemcacheServiceFactory.getMemcacheService();
 }

 @Override
 public IPriceCalculatorStrategy get() {
  if(!cache.contains(bindingType)) {
   // Retrieve tenant-specific config for price calculator feature.
   Configuration conf =
     ConfigurationManager.INSTANCE.getTenantConfiguration(bindingType);
   try {
    if(conf == null) {
     // Tenant-specific configuration not set, retrieve default one.
     Class<?> def = ConfigurationManager.INSTANCE.
         getDefaultConfiguration(bindingType).getValueAsClass();
     cache.put(bindingType, def.newInstance());
    } else {
     cache.put(bindingType, conf.getValueAsClass().newInstance());
    }
   } catch (Exception e) {
    // Failed to create tenant-specific customization.
   }
  }
  // Retrieve tenant-specific price calculator feature from cache.
  return (IPriceCalculatorStrategy) cache.get(bindingType);
}}
```

General approach

Windows Azure does not provide support for developing multi-tenant applications. This lack of built-in support complicates the development and forces the SaaS provider to implement its own multi-tenancy strategy. In [44] and [136] (Part 2, Chapter "Hosting a Multi-tenant Application on Windows Azure") different options for implementing a multi-tenant application are presented, but this is limited to data isolation.

Data isolation

As part of the requirements of data isolation, we identified that two main components are required: (i) a *tenant context* containing the information of the tenant linked to the current request (via a unique tenant ID) (cf. namespace in GAE), and (ii) *multi-tenant data storage*. Neither is provided by Windows Azure. The same applies to tenant-aware user management (e.g. no tenant administrator role). The AppFabric Access Control service also does not have built-in support for multi-tenant applications. Like in GAE, this should be implemented by the SaaS provider. In the following paragraphs we present our solution to obtain a tenant context and a multi-tenant data storage.

Tenant context. Similar to Google App Engine, a filter is required to intercept the incoming requests and to set the current tenant context. It is also possible in Azure to assign a separate domain to each tenant of the SaaS application. However, custom domains are not supported when using SSL, because the IIS web server can only have one SSL certificate associated with a port (default port 443). In this case the tenant ID in Windows Azure is determined based on the context path of the URL: `https://<azure-account>.cloudapp.net/<application>/<tenant-id>/*`.

For the hotel booking application, we acquire the tenant ID from the context path via the `MvcApplication.RegisterRoutes` method provided by the MVC framework and add it to the request context (see lines 6-10 in Listing 2.9). Each route consists of a route name, a template URL, and a list of default parameters. The presented route sends the requests to the `BookingController`, calling the `Index()` method. In the same way as the controller and action parameters are added to the request context, the tenant ID is passed through with the "tenant" key.

Listing 2.9: RegisterRoutes method to acquire the tenant ID.

```
 1  public class MvcApplication : System.Web.HttpApplication
 2  {
 3   ...
 4   public static void RegisterRoutes(RouteCollection routes)
 5   {
 6     routes.MapRoute(
 7         "Default", // Route name
 8         "{controller}/{tenant}/{action}/{id}", // URL with parameters
 9         new { controller = "Booking", action = "Index",
10               id = UrlParameter.Optional }); // Parameter defaults
11     ...
12   }
13   ...
14  }
```

The next step consists of setting the current tenant context. To represent the tenant context we use `TenantController`, an abstract MVC controller class with the filter method `OnActionExecuting`. All other controllers (e.g. the `BookingController`) inherit from this `TenantController` and therefore have access to the tenant context. Listing 2.10 shows the `TenantController`: it retrieves the tenant ID from the request context by means of the "tenant" key (lines 20-22), verifies the tenant ID (lines 26-27), and then sets the current tenant (line 25). The `TenantStore` refers to the table storage containing information about all tenants. This approach is based on the Windows Azure Architecture Guide (Part 2) [136], but it is limited to MVC applications. Alternatively, a similar approach with namespaces like GAE can be used: an application-wide tenant context manager storing the tenant context of the current request.

Listing 2.10: TenantController containing the current tenant context.

```
 1  public abstract class TenantController : Controller
 2  {
 3    private readonly ITenantStore tenantStore;
 4
 5    public TenantController(ITenantStore tenantStore)
 6    {
 7      this.tenantStore = tenantStore;
 8    }
 9
10    public ITenantStore TenantStore
11    {
12      get { return this.tenantStore; }
13    }
14
15    public Tenant Tenant { get; set; }
16
17    protected override void OnActionExecuting(
18                ActionExecutingContext filterContext)
19    {
20     string tenantId;
21     if (filterContext.RouteData.Values["tenant"] != null)
22       tenantId = (string) filterContext.RouteData.Values["tenant"];
23     else
24       // throw exception because of missing tenant in request context
25     if (Tenant == null)
26     {
27       Tenant = TenantStore.GetTenant(tenantId);
28       if(Tenant == null) // or other validation check
29         // throw exception because of invalid tenant ID
30     }
31     base.OnActionExecuting(filterContext);
32  }}
```

Multi-tenant data storage. A multi-tenant data storage can be obtained by applying filters that intercept the calls to the storage API and inject the tenant ID from the associated tenant context. In addition, comparable interceptors are necessary for the caching service (distributed in-memory storage). This forms a tenant-aware layer on top of the data storage, which makes abstraction of the chosen data isolation strategy.

In [45], three distinct approaches for creating multi-tenant data architectures are identified. A common solution on Azure to achieve tenant-aware data isolation is to assign a separate SQL Azure database for each tenant. This is a simple approach for the SaaS provider, but rather expensive for the tenant. The selection of the right data isolation strategy also depends on the kind of data (e.g. number of relationships between the data entities), and the kind of queries on these data. We integrated tenant-aware data isolation into our implementation of a custom persistence layer for Azure Table Storage. Isolation is achieved by adding the tenant ID (retrieved from the tenant context) to the partition key of each entity.

Tenant-specific configuration & customization

Windows Azure offers no support for tenant-specific configuration and customization. However, if data isolation is ensured and the tenant IDs can be retrieved, for instance using the approach described above, it is possible to achieve tenant-specific customization via dependency injection (cf. our approach on GAE). Azure also lacks support for tenant administrators to manage the tenant-specific configuration and for SaaS providers to develop and manage the different features. The SaaS provider should develop and provide this, preferably as a reusable framework. The next paragraph describes our approach for tenant-specific customization using the Ninject dependency injection framework (v2.2) [102].

Tenant-aware dependency injection. To achieve tenant-specific customization using dependency injection, we have applied the same approach as on Google App Engine. The Ninject dependency injection framework is, like Guice, not tenant-aware. However, Ninject uses by default a provider, the `StandardProvider`. Each dependency is handled by this provider, and therefore there is no direct binding between a type (i.e. interface) and the injected implementation. A provider can decide based on a `IContext` parameter which implementation should be injected. This offers the perspective to make Ninject tenant-aware by passing the tenant ID via this context parameter. It requires additional development by the SaaS provider to ensure (i) the right tenant ID is added to the context parameter, perhaps by means of a dedicated `IContext` implementation, and (ii) the tenant ID is used by a tenant-aware provider to return the right customization based on the tenant-specific configuration.

Listing 2.11 presents our solution for tenant-specific customizations on top of Azure. We provided a `PriceCalculatorModule`, which is used to assign `PriceCalcula-torProvider` as the custom, tenant-aware provider for injecting the price calculation feature on a per tenant basis (lines 13-14). This `PriceCalculatorProvider` first tries to retrieve the tenant-specific configuration for the price calculation feature, or when not available, the default one (lines 26-32). Next, it can retrieve and instantiate the appropriate implementation for the `IPriceCalculatorStrategy` feature (lines 34-35). Another approach for tenant-specific customizations using MVC and Ninject is presented in [63], but it is not applied on top of a cloud platform.

Listing 2.11: Structure of the PriceCalculatorModule and the PriceCalculatorProvider on Azure.

```
 1  public class PriceCalculatorModule : NinjectModule
 2  {
 3   private readonly ITenantStore tenantStore;
 4
 5   public PriceCalculationModule(ITenantStore tenantStore)
 6   {
 7    this.tenantStore = tenantStore;
 8   }
 9
10   public override void Load()
11   {
12    // Register custom provider for the price calculation feature.
13    this.Bind<IPriceCalculatorStrategy>().
14      ToProvider(new PriceCalculatorProvider(tenantStore));
15  }}
16  public class PriceCalculatorProvider :
17                             Provider<IPriceCalculatorStrategy>
18  {
19   ...
20   protected override IPriceCalculatorStrategy
21                              CreateInstance(IContext context)
22   {
23     var bindingType = typeof(IPriceCalculatorStrategy).FullName;
24     var tenantId = GetCurrentTenant(context);
25     // Retrieve tenant-specific config for price calculator feature.
26     Configuration conf =
27       configurationManager.GetConfiguration(tenantId, bindingType);
28     if (conf == null)
29     {
30      // Tenant-specific configuration not set, retrieve default one.
31      conf = configurationManager.GetDefaultConfiguration(bindingType);
32     }
33     // Return tenant-specific instance for price calculator feature.
34     return Activator.CreateInstance(Type.GetType(conf.BindingValue))
35              as IPriceCalculatorStrategy;
36  }}
```

Tenant-specific application management

Similar to Google App Engine, Windows Azure only supports application-specific monitoring, which is too coarse-grained. To fullfil the requirement of tenant-specific application management, SaaS providers will have to implement this facility themselves.

2.5.4 Summary

Force.com is the only PaaS platform that provides built-in support for data isolation as well as tenant-specific configuration and customization. Concretely, the SaaS developer does not have to write additional code to support multi-tenancy (Table 2.7). The only condition is that both the SaaS provider and the tenants have an account (org) on Force.com. Multi-tenancy support on Google App Engine is limited to data isolation, while Windows Azure lacks any support for the development of multi-tenant SaaS applications. If data isolation is ensured (via Namespaces API or manually), we demonstrated that dependency injection can be used to support tenant-specific customizations, but in a limited way compared to Force.com (for instance, tenants cannot add custom code). In addition, the SaaS developers should implement support for tenant-aware user management (e.g. to assign the tenant administrator role), and for managing the different features and configurations (i.e. configuration interface). This development effort is considerable, at least 600 to 750 lines of code to achieve a minimal support for multi-tenancy (Table 2.7). In the case of GAE, the Namespaces API enables a rather modularized approach to add multi-tenancy support. On Azure, however, the additional code for multi-tenancy support is scattered across the different tiers of the application, adding up to the development and maintenance complexity. Finally, none of the investigated PaaS platforms fully addresses the tenant-specific application management requirement.

2.6 Tool support

Tool support plays an important role to achieve smooth adoption of PaaS platforms. We discuss for each platform how it tackles the three identified subrequirements concerning tool support: (i) a development environment, (ii) a local development server and storage, and (iii) a testing framework. Table 2.8 presents an overview of the tool support provided by the different investigated PaaS platforms.

2.6.1 Force.com

The main development environment of Force.com is a web interface and it does not provide a local development server or storage. Unit tests are written in the Apex language. To develop and test applications on top of Force.com, a (free) Development Environment (DE) account is required.

Table 2.7: Overview of the code size (in lines of code) of the full implementations of the hotel booking application on top of the different PaaS platforms. The last column indicates the ratio of the implementation for multi-tenancy support compared to the full implementation. Note: the code size of the Force.com implementation also includes the metadata that is generated by using the point-and-click interface.

	Type	**# files**		**# lines of code**		**% multi-tenancy**
Force.com	Apex	8		753		/
	XML (generated metadata)	24		615		
	VisualForce	6		229		
GAE	Java	41		1830		± 29%
	JSP	16		616		
	XML (config)	3		72		
	HTML	4		24		
Azure		*SQL*	*Table*	*SQL*	*Table*	± 21%
	C#	56	69	1791	2509	
	ASP.NET Web Page	23	23	919	919	
	XML (ASP.NET config)	9	9	197	187	

Table 2.8: Overview of tool support for the three investigated PaaS platforms

	Development environment	**Local development server & storage**	**Testing framework**
Force.com	• Web interface (main) • Force.com IDE (Eclipse plugin)	Always remote	• Apex Unit tests • 75% of all Apex code needs to be covered
GAE	• Eclipse plugin • App Engine SDK + Apache Ant	Development web server & datastore simulation	• JUnit & others • Google App Engine testing utilities
Azure	• Visual Studio plugin • Windows Azure SDK	• Compute & storage emulator • SQL (Express) Server	Visual Studio test tools

Development environment

Force.com offers two development environments: an online development environment inside the browser and an IDE based on Eclipse. The web interface supports SaaS developers with creating custom objects and workflows, modifying business objects, and writing Apex code. In addition, tenant administrators are able to administer the application (e.g. configuration, user management and security controls). Compared to the web interface the Force.com IDE is not mature yet: the provided functionality is limited to writing Apex code and Visualforce pages (UI), and some features do not work well, for instance auto-completion of SOQL statements or Visualforce code as well as refactoring. Therefore, SaaS developers are often forced to use the web interface instead of the IDE they are familiarized with.

Local development server & storage

To make the application resources accessible via the web interface, all code and metadata is synchronized with the Development Environment (DE) account on the server. The platform does not provide a local development server or storage: validation and execution only occur at the server side. This has as consequence that the IDE is less responsive than the web interface (each 'save' operation results in a synchronization) and prevents integration with the Eclipse debugger. In addition, this constant synchronization with the server complicates collaboration between multiple developers because there is no versioning support. This issue can be solved by using a Partner Development Environment which is a master repository environment to manage all the source code; each developer can check out to his own DE.

Testing framework

Force.com requires the creation and execution of unit tests to deploy Apex code to a production environment. These test classes and methods are written in the Apex language and verify whether a particular piece of code is working properly. At least 75% of the Apex code should be covered before deployment is possible. This encourages test-driven development and improves correctness of the Apex code. An example of a test method for the `CheckBookingPeriod` trigger is presented in Listing 2.12. Notice that the approach is similar to unit tests in common programming languages.

Listing 2.12: Test class with test method for the CheckBookingPeriod trigger.

```
@isTest
private class TravelApplicationTest {
 static testMethod void testBookingOverlap() {
  // Perform preparation of test method:
  // 1) Create and store an invoice.
  ...
  // 2) Create a new invoice with overlapping booking
  Invoice__c invoice = new Invoice__c(Status__c = 'Negotiating');
  insert invoice;
  Booking__c booking = new Booking__c(Invoice__c = invoice.Id,
    Start__c = date.newInstance(2012, 8, 13),
    End__c = date.newInstance(2012, 8, 18), ...);

  // Start the test.
  // The context changes to that of the CheckBookingPeriod trigger.
  Test.startTest();
  try {
    // Insert the overlapping booking
    insert booking;
  } catch(System.DmlException e) {
    // Assert that an exception is thrown: insertion of booking failed.
    System.assert(true);
  }
  // Stop the test, this changes the context back to test from trigger.
  Test.stopTest();
}}
```

2.6.2 Google App Engine

Google App Engine comes with an Eclipse plugin, a local development web server and datastore simulator, and testing utilities to support the writing of unit tests. A (free) Google account is only required to deploy the application in the cloud.

Development environment

To ease the development of Java App Engine applications, Google offers a plugin for Eclipse (Java EE version) to create, test and upload applications from within Eclipse. In contrast to the Force.com IDE, this plugin works smoothly. There is only a small issue with the (rather frequent) updates of the GAE libraries: when collaborating with multiple developers this can lead to inconsistencies.

Local development server & storage

The Google App Engine SDK includes a development web server that simulates the GAE Java runtime environment and the supporting services such as the datastore and the users service. While the Force.com IDE sends all code to an external server, this development server allows to extensively test GAE applications locally. The server can be executed in the Eclipse debugger via the Google plugin, or via the command line (using Apache Ant). However, the behaviour of applications running on the development server does not always correspond to their behaviour on top of GAE. For example, objects that are stored using the Memcache service should be serializable, however this is not required by the development server. Applications that run on the development server, therefore not necessarily work on top of GAE. Force.com does not have this problem, because applications are always validated and executed on the (remote) Force.com platform.

Testing framework

The SDK also contains testing utilities to support the writing of unit tests that have dependencies on Google App Engine services. The `LocalServiceTestHelper` class is a helper class that handles the setup of the environment. To configure the local GAE services, it requires a `LocalServiceTestConfig` instance as argument for each used service. For example, when using the datastore service, a `LocalDatastoreServiceTestConfig` instance should be passed as argument to the constructor. The GAE testing utilities are not tied to any specific testing framework, so it is possible to use JUnit, a widely used unit testing framework for Java. In contrast to Force.com, unit tests are not mandatory. Listing 2.13 shows the outline of a JUnit test case with dependencies on the Datastore and Memcache services.

2.6.3 Windows Azure

Development for Windows Azure is supported by a Visual Studio plugin and a compute and storage emulator. Visual Studio provides by default the necessary testing tools to write unit tests. Deployment of applications on Windows Azure requires an Azure account.

Development environment

The Windows Azure SDK is available in a standalone version or can be installed together with the Windows Azure Tools, a plugin for the Visual Studio IDE. These tools are also available for Eclipse to support Java development for Windows Azure. Moreover, it is possible to import existing .NET projects into a new Azure project. This supports the migration from an on-premise implementation to Azure.

Listing 2.13: Example of Google App Engine testing utilities.

```
public class HotelManagementTest {
 private final LocalServiceTestHelper helper =
   new LocalServiceTestHelper(new LocalDatastoreServiceTestConfig(),
     new LocalMemcacheServiceTestConfig());
 @Before
 public void setUp() throws Exception {
  // Setup configuration of local Datastore and Memcache
  helper.setUp();
 }
 @After
 public void tearDown() throws Exception {
  helper.tearDown();
 }
 @Test
 public void testAddRoom() {
  // Perform the AddRoom operation
  ...
  // Assert the correctness using the Datastore and Memcache services
  ...
 }
 ...
}
```

Local development server & storage

The Windows Azure SDK offers a compute and storage emulator. This enables SaaS developers to test their Windows Azure applications locally without an Azure account. There exists no local SQL Azure database, but applications that access SQL (Express) Server locally will largely work unchanged with SQL Azure. For example, distributed transactions across multiple databases are not supported by SQL Azure. Other differences are less important and only apply in exceptional cases. In contrast to Force.com and Google, Microsoft does not provide a free developer account to deploy and test applications in the cloud.

Testing framework

To create unit tests the existing testing tools provided by Visual Studio can be reused. However, there is no direct support, like GAE provides, to test an application against the storage tools of the Azure SDK. To effectively write unit tests for classes that interact with Windows Azure Storage, it should be possible to mockup the calls to the storage service. This can be achieved by writing a wrapper interface for `WindowsAzure.StorageClient` that can be reused by all classes and applications interacting with the storage service. This has been implemented in the sample application for part 2 of the Windows Azure Architecture Guide [136]. Another option is to setup the emulator with a default test configuration, like the Google App Engine test utilities do.

2.6.4 Summary

All investigated platforms provide good tool support in the form of a plugin for a commonly used IDE. However, in the case of Force.com the IDE is not sufficient to develop an entire application, but only supports Apex and Visualforce development. Google App Engine and Windows Azure also allow developers to use a standalone version of the SDK in combination with the command line. Another difference between Force.com and the two other PaaS platforms is the lack of a local development server. While GAE and Windows Azure support local testing of cloud applications, Force.com requires that all code is synchronized on their servers. Both differences are disadvantages caused by the metadata-driven architecture of the Force.com platform. Finally, all platforms support unit testing that is similar to the on-premise approach.

2.7 Discussion & challenges

In this section we discuss the different requirements for developing SaaS applications in the context of the different PaaS categories and relate these requirements to our results. Further, we discuss the impact of the specific case study on our findings and we briefly analyse the impact of our implementation decisions on these results. Finally, we itemize a set of open challenges that we have identified.

2.7.1 Application of results to PaaS categories

The main challenge for SaaS providers is to select the most appropriate PaaS platform (or category of PaaS) for their SaaS application. Our comparison has demonstrated that none of the investigated cloud platforms stands out above the others with respect to all three requirements concerning support for SaaS development.

Based on the results of this experience report, we can conclude that tool support is not a key differentiator with respect to SaaS development. Most PaaS platforms already offer good tool support in the form of a plugin for one of the common IDEs, and integration support with unit testing frameworks (e.g. JUnit). A point of attention is that the behaviour of the local development server should correspond with the cloud environment to enable developers to fully test the SaaS application. For example, we noticed this is not always the case with GAE.

Consequently, the selection between PaaS platforms can be reduced to the choice between portability and multi-tenancy support. The three PaaS platforms (of different categories) that we compared all relate differently to the portability and multi-tenancy requirements (see Fig. 2.6). The metadata-driven PaaS platforms (e.g. Force.com) are clearly the winner with respect to multi-tenancy. However, if portability takes precedence, PaaS category 1 is the best option, with the platforms of category 2 as runner-up. The following paragraphs discuss portability and multi-tenancy support for each of the PaaS categories.

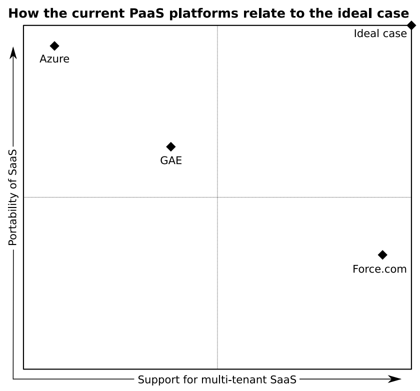


Figure 2.6: Overview of how the three representative PaaS platforms that we compared relate to the portability and multi-tenancy requirements. The data points are based on the different scores as defined in Section 2.2.3. The right top corner represents the ideal case of a PaaS platform that fully addresses both requirements. This figure clearly reveals the gap between the existing platforms and the ideal case.

PaaS category 1. The main goal of PaaS category 1 is clearly to support the provider's customers and developers in the migration from the on-premise to the cloud model. To achieve this goal, these PaaS providers try to support the standard APIs that are used in popular enterprise application servers and middleware frameworks. This is reflected by a high degree of portability, as clearly illustrated by Windows Azure with its high level of code reuse (Table 2.5 and Fig. 2.6). Typically, using PaaS platforms of this category results in code reuse of at least 90—95%. Moreover, the use of standard APIs not only facilitates the migration from on-premise to PaaS, but also between PaaS platforms of this category. Obviously, the latter is only valid as long as the same programming language and similar APIs are supported.

Because these PaaS platforms are based on existing solutions with standard APIs, they are often also available as a private cloud platform. For example, Red Hat OpenShift [175] (Java EE and JBoss applications) and Cloud Foundry [211] (Spring applications) are two open source PaaS platforms with both public and private cloud offerings. This way, SaaS providers have the option to keep the execution of their SaaS applications in-house.

To improve performance and scalability, these PaaS platforms are extended with typical cloud services, such as scalable storage and distributed caching. However, support to integrate with these services is limited or lacking, and this requires a significant amount of additional development by the SaaS provider. For example, we had to build a custom persistence layer to map our domain model to row entities that could be stored in the Azure Table Storage (Table 2.5). Some platforms of category 1, for instance Amazon Elastic Beanstalk [5], provide vendor-specific SDKs to make abstraction of the lower-level APIs for scalable storage services and to reduce the development overhead. However, such an approach compromises the portability.

Support for developing multi-tenant applications is completely lacking in PaaS category 1. The SaaS provider has to build the multi-tenancy support from scratch. In this regard, we have shown how to achieve data isolation and to enable tenant-specific customizations. The additional development effort is quite large (Table 2.7) and therefore not recommended when operational cost benefits are not the priority. Instead of developing the multi-tenancy support in-house, it can be acquired from third parties. For example, Apprenda [10] is a private PaaS platform for .NET web applications on top of SQL server. It supports the execution of multi-tenant applications: it ensures data isolation and enables simple customizations, for example in the user interface.

Since the need for support for traditional enterprise applications in the cloud is increasing, PaaS category 1 can play a crucial role in facilitating the migration of these enterprise applications to the cloud. This also explains the high amount of platforms in this category compared to the other categories (Table 2.2). But SaaS providers could benefit more from economies of scale, if more development support would be provided to integrate with scalable cloud solutions as well as to endow applications with multi-tenancy.

PaaS category 2. The focused PaaS platforms are built on top of custom application servers, aiming for higher elasticity and scalability. However, this high level of scalability is achieved at the expense of portability. For example, APIs need to be adapted and optimized. Often these platforms target a specific type of cloud applications and offer some support for the development of such applications. For example, both Google App Engine and the XAP Elastic Application Platform by GigaSpaces [76] provide an abstraction layer on top of their scalable cloud storage to support JPA, though with limitations. Because of this focus on specific types of cloud applications, the set of supported technologies is limited, and different results for portability are obtained depending on which type of application is implemented.

In the case of the hotel booking application on GAE, around 78% of the Java EE code could be reused, which seems quite acceptable.

With the rise of more established middleware providers in the PaaS field (Table 2.2), the PaaS platforms of category 2 try to extend their offerings to enhance portability. For example, we noticed that GAE gradually adds new Java classes to its white list and recently, Google added a MySQL database service to its offering (i.e. Google Cloud SQL). In addition, Google released a REST-based cloud storage API to access the datastore, thus improving integration. Both are still in beta phase. However, for each of these extensions, they still take non-functional requirements like scalability into account.

The focused PaaS platforms typically support data isolation for multi-tenant applications. We consider this to be a first step in the right direction. For example, GAE offers the Namespaces API to provide a multi-tenant database. However, development support for tenant-specific configuration and customization is lacking, so is a tenant-specific application management facility lacking as well. Similar to the platforms of category 1, the multi-tenancy support has to be custom developed, as illustrated in Section 2.5. Because a core requirement for multi-tenancy, i.e. data isolation, is already present, the additional development effort is less complex, but still substantial (Table 2.7).

PaaS category 3. The metadata-driven PaaS platforms, like Force.com, lack any compatibility with common programming models. This is mainly because, compared to PaaS categories 1 and 2, these platforms offer a composition and configuration interface at a higher abstraction layer. Even non-developers, who do not have any programming experience, could be able to build cloud applications by using a point-and-click interface to complete forms and models. Additional business logic and custom user interfaces are written using platform-specific programming/scripting languages. In our implementation, the development effort was acceptable (Table 2.7), because we tried to use the built-in features as much as possible, for example the standard web pages to access and manage the business objects (e.g. hotels and rooms) in the Force.com application. Only the booking process required the development of a custom user interface and corresponding controller (using VisualForce and Apex).

Moreover, this higher abstraction level limits the complexity of the applications and especially the processing logic that can be implemented. The lack of portability by the metadata-driven platforms is a core issue of this approach and cannot be bridged by the SaaS providers themselves. As the migration to PaaS platforms of this category requires a full rewrite, it inherently means that the application is also not easily portable to other PaaS platforms (even of the same category). Therefore, these platforms try to compensate it with extensive integration capabilities. With almost each new release, Force.com extended its set of toolkits to improve integration.

The metadata-driven PaaS platforms are the only PaaS category that offers full-fledged support for multi-tenancy, enabling tenants to customize SaaS applications to their preferences, even via custom code. It is remarkable that for example Salesforce.com, a company that is rather new in the role of middleware (platform) provider, beats the rest of the field in terms of multi-tenancy support. Yet the support for tenant-specific application management remains limited. For instance, Force.com does not support tenant-aware performance and resource consumption monitoring.

2.7.2 Impact of case study application and implementation decisions

The choice of the online hotel booking application as a case study has an influence on the results of this experience report. So do the implementation decisions for the on-premise application. We discuss both elements below.

Impact of case study application. The online hotel booking application is a component-based web application, that does not depend on external web services or applications. However, as discussed in Section 2.4, we did investigate the integration support provided by the diffferent PaaS platforms to implement service compositions. Using another case study, consisting of a typical service composition, would therefore yield similar conclusions. (Yet it would enable us to illustrate how to solve the detected integration issues.)

Further, the data model of the hotel booking application contains several relationships between the different entities. Such a data model complicates the use of NoSQL datastores, as illustrated by the Windows Azure implementation using the Azure Table Storage. Also GAE imposes limitations on entity relationships. Blogs and surveys are typical examples of applications that are suitable for such key-value datastores, e.g. [136]. These applications have a simple data model with no or a limited amount of relationships between the entities. In the case of GAE and Azure Table Storage, we tried to solve this issue by applying a hierarchical data model with unidirectional relationships. As shown by the GAE implementation, this approach can work quite well. Therefore we think that the selection of the hotel booking application with a more complex data model, did not have a major impact on the results of portability. In addition, this decision resulted in the documentation of GAE offering better development support on top of the Bigtable datastore compared to Azure with the Azure Table Storage, as the latter clearly focuses on structured data without entity relationships.

Impact of implementation decisions. We implemented two on-premise versions of the case study application (i.e. a Java EE and a .NET version), which we used as basis for the comparison of the three PaaS platforms with respect to the different requirements. The implementation decisions have especially impact on the portability requirement.

In the case of Force.com and other metadata-driven PaaS platforms, the selection of specific implementation technologies for the on-premise versions has no impact on the results. The metadata-driven architecture and the Salesforce.com-specific Apex language are incompatible with any common programming model or middleware framework, and therefore different implementation decisions do not improve the portability of the application.

The decision to use Enterprise JavaBeans (EJB) [210] in the Java version has a negative impact on the portability to GAE. GAE is focused on web applications and therefore the more heavy-weight component model of EJBs is not supported. However, the use of Java Servlets and JSPs for the front end of the application is in favour of GAE. For example, a front end consisting of Web Services would have shown worse results, because additional development is required for the handlers of SOAP messages. Consequently, we can claim that the mix of supported and unsupported technologies in the on-premise implementation generates a fair perspective on the portability support provided by GAE.

Portability is no issue for the platforms of category 1 and therefore our implementation decisions do not have a major impact on the results. Instead of the MVC framework, we could have easily used the Windows Communication Foundation (WCF) [137], a framework providing a unified platform for building service-oriented applications based on SOAP or the REST approach. The same applies if we had evaluated a Java-based PaaS from this category, e.g. Red Hat OpenShift [175].

2.7.3 Challenges

The main challenge that we extracted from the results of our work is *the need to combine portability with operational cost efficiency* (as shown in Fig. 2.6). None of the three categories of PaaS platforms sufficiently support both portability and development of multi-tenant SaaS applications. Currently, the focus of the existing platforms is on only one of these requirements. The challenge is to combine the best of two worlds: (i) maximum operational cost efficiency by supporting application-level multi-tenancy, as well as (ii) portability across different platforms because of the use of mainstream programming models and middleware frameworks. We consider opportunities to address this challenge by architecting a middleware layer for application-level multi-tenancy and by standardization. We discuss this in depth in the following paragraphs, and identify several subchallenges concerning the different requirements based on the results of this experience report on developing multi-tenant SaaS applications on top of PaaS.

Middleware for application-level multi-tenancy. By applying multi-tenancy at the application level, resource sharing between the different tenants and thus operational efficiency are maximized. However, except for the metadata-driven platforms, current PaaS platforms hardly provide any support for developing multi-tenant applications. When needed, this support is currently implemented for each application separately. Ideally, the *multi-tenancy concern is modularized into a reusable middleware service* that can be enabled/disabled when necessary by the SaaS provider and configured to the application-specific requirements. This way, SaaS providers are able to easily endow their SaaS applications with multi-tenancy, without much development overhead.

A first requirement for such a middleware layer is the availability of *development support for data isolation and tenant-specific customizations.* We have explained how this can be achieved on top of GAE and Azure via dependency injection, but this approach should be generalized to reuse it for every application. A first step to prototype such a middleware has been achieved in [214]. In addition, more research is required to support the same level of flexibility in multi-tenant applications as the metadata-driven platforms (e.g. Force.com) on top of the PaaS platforms of the other categories.

Besides development support for data isolation and customizable multi-tenant applications, major challenges still exist in achieving tenant-specific application management and performance isolation at runtime. None of the PaaS platforms that we investigated fully address these two requirements.

Tenant-specific application management requires the administration of SaaS applications to be isolated between the different tenants, for example to enable SaaS providers to monitor how many resources are consumed by a specific tenant (for billing and SLA compliance). In addition, this includes the need for APIs and tools to support application management, in particular for tenant-aware monitoring. Most of the PaaS platforms are limited to managing and monitoring complete applications or even VMs instead of on a per tenant basis. The monitored data is too coarsely-grained and not suitable for multi-tenant SaaS applications. This is clearly an open challenge that should be addressed by middleware as well as tool support.

Performance isolation is key to prevent one tenant from adversely affecting the performance of other tenants and to ensure that the performance of the different tenants comply with their service level agreements (SLAs) [84]. Typically, performance isolation is achieved by using virtualization (e.g. Windows Azure) or at the middleware level (e.g. GAE and Red Hat OpenShift). However, these approaches restrict resource sharing between different tenants and have a higher application management and maintenance cost. At the same time, more sharing complicates isolation between the tenants. Currently, SaaS providers are unable to guarantee any performance-related SLAs to their tenants when hosting a multi-tenant SaaS application, even in the case of the metadata-driven platforms. Therefore, this complexity to achieve performance isolation in multi-tenant applications should also be tackled at the middleware level. In the meantime, SaaS providers have to make a trade-off between operational efficiency and fairness in the use of resources. This decision depends on the kind of application and the tenant-specific requirements. Moreover, it is possible to deploy a combination of multi-tenant and single-tenant applications to address the different requirements.

Standardization. Standardization is an appropriate solution to improve portability of SaaS applications across different on-premise and cloud-based platforms (public as well as private) and to tackle vendor lock-in. Based on our observations in this experience report, we believe that the need for standardization especially exist in (i) the use of non-relational storage and typical cloud services like caching, (ii) the support for multi-tenancy, and (iii) application management and monitoring. The latter two are strongly related to the challenges of a middleware layer for application-level multi-tenancy, as mentioned above. Standardization is especially relevant for the APIs provided by such a middleware layer. The issue with non-relational storage is illustrated with the Azure Table Storage, where we had to implement an application-specific persistence layer ourselves. If we decide to replace Azure Table Storage with an external cloud storage service, we would have to adapt this custom persistence layer to the new API. Some platforms already offer a persistence layer for non-relational storage (e.g. GAE and Amazon Elastic Beanstalk), but these are vendor-specific approaches and compromise the portability.

In this regard, JSR 342, the Java EE 7 Specification [58], aims to enhance the suitability of the Java EE platform for cloud environments and can therefore be an interesting contribution to the standardization of Java-based PaaS platforms. The goal is to extend the Java EE platform with support for the PaaS model as well as a limited form of the SaaS model, while preserving as much as possible the current programming model. Concretely, the JSR proposes a.o. the following: (i) a definition of new platform roles for the PaaS model, (ii) metadata support for provisioning, QoS, isolation etc., (iii) potential standard APIs for non-relational databases, caching (JCache - JSR 107), management and monitoring, and (iv) multi-tenancy support in existing APIs, for example JPA. The latter is a first step to address the challenge of a middleware layer for application-level multi-tenancy, although it focuses mainly on data isolation. The proposed management and monitoring interfaces, however, are still at the granularity of complete applications (i.e. middleware-level multi-tenancy). The same applies to the specified isolation and QoS requirements.

With respect to integration support, standards exist in the form of SOAP- and REST-based web services, and as we explained, they are already commonly used by the different PaaS platforms. Only GAE did not have built-in support and required external libraries or additional development. Furthermore, more effort is required in providing development support and tools to improve integration with external cloud services (e.g. datastores) and platforms. This is important to tackle vendor lock-in. We notice that Force.com (with its toolkits), but especially the open source platforms are currently the front runners. An example of such a library is jclouds [203], an open source library that offers several API abstractions for cloud services like Amazon S3 and Azure Blob Storage. Finally, OCCI [155] comprises a set of specifications for cloud APIs to support integration, portability and interoperability. However, it focuses especially on IaaS.

2.8 Related work

This section illustrates the gap in the current state of the art with respect to the in-depth evaluation of cloud offerings based on practical case studies. The evaluation of cloud offerings has been limited to general comparisons between the different delivery models (i.e. IaaS, PaaS and SaaS), or discussions based on high-level concepts (e.g. computing architecture, storage and load balancing). In the context of IaaS, more in-depth studies and surveys have been conducted; this paper is to the best of our knowledge the first of its kind in the growing space of PaaS. Furthermore, this section discusses related work in the context of the main challenges that we identified based on our results: support for multi-tenancy in SaaS applications and portability across cloud platforms.

Surveys on cloud platforms. In the current state of the art, most research has been done in the context of IaaS with platforms such as Eucalyptus [153], OpenNebula [198], OpenStack [156] and CloudStack [46]. This results in surveys especially on IaaS platforms, e.g. [172]. Our work, however, focuses on PaaS platforms, and more specifically the development support for SaaS applications provided by these PaaS platforms.

Armbrust et al. [11] distinguish different classes of utility computing based on the level of abstraction. At one end of the spectrum they classify Amazon EC2 [2], and at the other extreme of the spectrum application-domain-specific platforms such as Google App Engine [77]. Windows Azure [134] is an intermediate point on their spectrum. They compare how these three platforms virtualize computation, storage, and networking and how scalability and high availability is ensured. Further, the authors discuss the economics of cloud computing and what the opportunities and challenges are for cloud computing in general, including the issue of data lock-in (i.e. data portability across different platforms). The focus of their work is especially on runtime properties of cloud computing, while we compare PaaS platforms based on requirements related to the development of SaaS applications, i.e. portability, support for multi-tenancy, and tool support.

Other work, such as [177, 178, 191], gives a general overview on cloud computing and discusses IaaS, PaaS as well as SaaS, including some examples. They are often limited to the enumeration of some high-level concepts (e.g. computing architecture, load balancing and storage), without any hands-on experience with the platforms. For example, the book "Enterprise Cloud Computing" [191] addresses cloud computing for enterprise applications, covering the common concepts and technologies like web services, virtualization and multi-tenancy. In addition, the author introduces several cloud platforms like Amazon EC2 [2], Google App Engine [77], and Windows Azure [134]. Furthermore, the work discusses extensively the concept of metadata-driven PaaS platforms, such as Force.com [182, 219] and InstantApps [192]. These metadata-driven platforms are called Dev 2.0 platforms [190] by the author. We focus on an in-depth comparison of PaaS platforms (including the Dev 2.0 platforms) with respect to support for SaaS application development, on the basis of a case study that involves the development of multi-tenant SaaS applications on top of different PaaS platforms. Therefore, our work is complementary.

In [176], Reese explains how to design, build and maintain web applications that can be deployed into the cloud, including how to move existing web applications to the cloud. In addition, the book discusses differences between traditional on-premise deployment and cloud computing in areas such as reliability, security and scalability. However, the focus is on building applications on top of IaaS with Amazon Web Services (AWS) [4] as primary example, and the work does not tackle the issue of multi-tenancy at the application level. We, however, evaluate the development support for SaaS applications on top of several PaaS platforms, with support for multi-tenancy as one of the key criteria to achieve higher cost efficiency.

Support for multi-tenancy in SaaS applications. The state of the art on multi-tenancy in SaaS applications can be subdivided into multi-tenant data storage (i.e. multi-tenancy at the data tier), and methodologies and patterns to achieve multi-tenancy. Notice that multi-tenancy is only one of the requirements that we investigated with respect to SaaS application development.

A key requirement to support multi-tenancy in SaaS applications is data isolation between the different tenants. Chong et al. [45] identified three approaches for managing multi-tenant data and described a number of patterns to realize a multi-tenant data architecture. In [15], a new schema-mapping technique for multi-tenant databases, called Chunk Folding, is presented. [16] compares five techniques for implementing flexible schemas for multi-tenant SaaS applications. Our work compares PaaS platforms with respect to development support for multi-tenant SaaS applications, including data isolation. The schema-mapping techniques in the state of the art can be applied by PaaS providers to support data isolation in the platforms, or by SaaS providers to implement this support themselves when it is absent.

In [44], Chong and Carraro present the four-level SaaS maturity model: (i) *single-tenant*, a separate custom application per tenant, (ii) *configurable single-tenant*, a separate instance of a configurable application for each tenant, (iii) *configurable multi-tenant*, a single instance serving every tenant with configurable metadata, and (iv) *scalable, configurable multi-tenant*, a load-balanced pool of identical multi-tenant instances, which are configurable to the tenant-specific requirements. Furthermore they describe a high-level architecture of a SaaS application, especially focusing on the addition of metadata services to manage the tenant-specific application configuration. Our work investigates the development of multi-tenant SaaS applications, customized to the tenant-specific requirements, on top of PaaS platforms. In principle, the PaaS platforms take care of the scalability concern. Therefore, our goal is to develop SaaS applications that correspond to level 4 in the SaaS maturity model (i.e. scalable, configurable multi-tenant).

Guo et al. [84] discuss the design and implementation principles to support the development of multi-tenant applications. The focus is on better isolation of security, performance, availability (i.e. fault isolation) and administration between the different tenants. Our work, however, focuses on the comparison of PaaS platforms, based on a practical case study, with respect to the provided development support for multi-tenant SaaS applications. The design principles presented by [84] can be applied in our work when the support for multi-tenancy by the PaaS platforms is inadequate.

Bezemer et al. [25] describe how to enable multi-tenancy in software services. They report their experiences with transforming an existing single-tenant software system into a multi-tenant one by means of their reengineering pattern [24]. This pattern requires three additional components: a multi-tenant database, tenant-specific authentication and configuration. The resulting SaaS application is hosted by the SaaS provider itself and is not deployed on top of a PaaS platform. In this paper, we compare the support offered by PaaS platforms to develop multi-tenant SaaS applications. One of our requirements regarding SaaS development is the need for multi-tenancy support. Therefore, the principles of the reengineering pattern complement our work and are used as requirements for the multi-tenancy support.

Portability across cloud platforms. Andrikopoulos et al. [7] analyse the migration of applications to the cloud based on a survey of the state of the art, identifying several challenges. A complete migration of an application to the cloud using the available cloud services is considered the most effective solution, however it requires the most re-engineering effort, showing the need for portability. In this work, we apply such a migration scenario and evaluate the support provided by PaaS platforms to develop SaaS applications based on a practical case study.

Mietzner et al. [139] propose an extension to the service component architecture (SCA) standard, adding variability descriptors and multi-tenancy patterns. This way, they try to combine portability with support for multi-tenancy. Similarly, [161] applies SCA to deploy applications on top of multiple heterogeneous PaaS and IaaS environments. Their federated PaaS infrastructure relies on their own FraSCAti application environment for SCA applications that should be deployed on top of each PaaS platform. In contrast to [139], their work does not offer support for multi-tenancy. However, SCA focuses on a limited set of application types, and applications can only be executed in a SCA-aware application environment (e.g. FraSCAti). In addition, SCA can only ensure portability with respect to computation. For example, it does not address the differences in the APIs of the different cloud services (e.g. storage, caching, etc.), which we identified as an important challenge in the current PaaS offerings.

The European mOSAIC project [166] develops a new platform with an open, independent API to support heterogeneous hybrid clouds. The goal is to deploy this platform on top of various existing PaaS platforms to enable the portability of component-based applications that are developed for their (new) platform. The latter requirement, however, hinders the migration of existing, on-premise applications to this platform. Moreover, support for multi-tenancy is not addressed by this work.

2.9 Conclusion

To benefit from economies of scale, software vendors aim to migrate from an on-premise deployment model towards a Software-as-a-Service (SaaS) model. PaaS platforms aim to facilitate the development and deployment of cloud applications by delivering a computing platform and solution stack as a service. As with any paradigm shift, many new and somehow competing technologies enter the arena; more and more PaaS technologies have emerged. This evolution demands for independent research on the relative value and performance of new types of technologies. Given the broad spectrum of competing technologies, this paper addresses a selection of three platforms that each can be considered to be representative for a larger category of PaaS platforms.

The paper covers an in-depth investigation of the support of Force.com, Google App Engine and Windows Azure for the development of multi-tenant SaaS applications. The paper addresses these three representative PaaS platforms based on a practical application case study. Google App Engine and Windows Azure provide better portability based on their compatibility with the mainstream development model, but these platforms lack built-in support for multi-tenancy. Force.com introduces a metadata-driven architecture and an interpreted programming language (Apex) to support the development of multi-tenant applications that are tailored to tenant-specific requirements. However, this approach hinders straightforward portability. A major research and development challenge consists of providing the same level of flexibility for multi-tenant applications (similar to Force.com), but on PaaS platforms that apply a more traditional and therefore portable programming model.

Clearly, application-level multi-tenancy remains an important challenge as PaaS platforms offer insufficient support. Modularizing all aspects of the multi-tenancy requirement into a reusable middleware layer is an essential track of ongoing and future work. Finally, there is a need for standardization to improve portability of SaaS applications across different (cloud) platforms and to tackle and avoid vendor lock-in. Standardization is especially relevant for typical cloud services like scalable storage, and also to support application-level multi-tenancy.

Besides SaaS development on top of PaaS platforms, it is definitely worth and necessary to consider and evaluate other aspects of current PaaS offerings, for example operational aspects (e.g. monitoring, updating and patching), as well as business aspects (e.g. contracting and pricing). In addition, it will remain important to follow-up on the evolution and improving quality of existing PaaS offerings, also with respect to portability, multi-tenancy and tool support. This paper therefore offers but one, yet important, contribution to the experience reporting and analysis that is required to guide and support the paradigm shift and transition to SaaS.

Acknowledgements

We thank the anonymous reviewers for their constructive comments to improve this paper. We also thank Wouter De Borger, Maarten Decat and Dimitri Van Landuyt for proofreading. This research is partially funded by the Research Fund KU Leuven, the iMinds CUSTOMSS project and by the EU FP7 project NESSoS. CUSTOMSS is a project co-funded by iMinds (Interdisciplinary institute for Technology), a research institute founded by the Flemish Government. Companies and organizations involved in the project are Agfa Healthcare, Televic Healthcare and UnifiedPost, with project support of IWT (government agency for Innovation by Science and Technology).

Chapter 3

Middleware framework for co-existing variants in multi-tenant SaaS

Current cloud-enabling middleware and PaaS offerings do not sufficiently support the development and customization of multi-tenant SaaS applications (see Chapter 2). To address this concern, this chapter presents an advanced middleware framework that builds upon these existing platforms and that provides the necessary support to realize the run-time and context-sensitive composition of tenant-specific variants within the same application instance. It is a framework in the sense that it provides both a versatile architecture that supports different implementations and deployments, and a set of reusable, loosely-coupled components that implement key parts of the architecture [185]. This versatility has been demonstrated in several prototypes, in the context of different applications and on top of diverse platforms, for example [74, 204, 213, 214]. Chapters 4 and 5 present two such instances of the architecture (each with a different focus), and as such contribute to the evaluation of this framework.

Section 3.1 defines the scope, requirements and constraints that shape this middleware framework. Subsequently, the architecture of the framework is presented in Section 3.2. Finally, Section 3.3 elaborates on the versatility of the presented middleware framework.

3.1 Architectural drivers

This section discusses the main architectural drivers that shape the middleware framework for co-existing variants in multi-tenant SaaS applications. Besides the main functional requirement of enabling tenant-aware customization, the scope of the middleware and the key non-functional requirements have a major impact on its architecture and constrain its openness.

3.1.1 Scope

The scope of the middleware is defined by (i) the stakeholders relevant to SaaS application development, management and customization, (ii) the supported types of SaaS applications, (iii) the customization goals and mechanisms.

Stakeholders. The focus is on two types of stakeholders: *SaaS providers* and *tenants*. SaaS providers design, develop and host SaaS applications on top of an existing PaaS offering or using cloud-enabling middleware. For these activities, the SaaS architect, developer and operator are the relevant employees of the SaaS provider.

As the different tenants have different and varying requirements, they want to be able to subscribe to a SaaS offering and to tailor the application to their specific requirements and preferences. In general, the task of managing the SaaS application on behalf of the tenant (e.g. user management, access control, configuration) is assigned to the *tenant administrator*. In the context of this dissertation and in line with the self-service characteristic of cloud computing, this specific role belongs to an internal user of the tenant organization.

Application constraints. This dissertation focuses on distributed enterprise SaaS applications, without putting any constraints on the types of applications that are supported. As explained in Section 1.3, two common types of enterprise SaaS applications have been used throughout this work, i.e. a request-driven, component-based multi-tier (web) application and a batch-driven, workflow-based service composition. Therefore, the middleware framework should support at least both types of applications (see Chapters 4 and 5). As these two types of applications are quite diverse, we expect the middleware framework to be sufficiently versatile.

A key constraint with respect to the supported SaaS applications is defined by the *feature-oriented application model.* More specifically, applications should be decomposable into features. These features consist of a set of modular software artifacts that can be easily composed to create tenant-specific variants of the application.

Customization goals. The middleware framework is open with respect to the customization types that are supported. Although different types of customization exist at different levels in the software stack, the scope is limited to application middleware. Thus, the focus of the middleware architecture is on enabling tenant-specific customizations (tailored to both the functional and non-functional requirements of the different tenants), which should be (fully) realized by or within the application middleware. However, this should not prevent the middleware from addressing concerns that are (mainly) tackled by the underpinning platform, such as availability and scalability: the middleware architecture should define interfaces that allow to control and direct the underpinning platform.

To validate the openness of the middleware framework, the following two chapters address two different types of customization. We applied a demand-driven approach to select these two customization types. The main customization requirement of tenants is related to the business logic, for example to tailor the functionality of the application to their specific business needs. Therefore, Chapter 4 focuses on middleware support to enable tenant-specific software variations in the core of the SaaS application and the middleware. A second important concern for tenants are the different requirements with respect to the expected performance (throughput as well as latency), as specified in service level agreements (SLAs). Chapter 5 aims to ensure compliance to these different tenant-specific SLAs while preventing that the behaviour of one tenant adversely affects the performance of the other tenants (i.e. performance isolation). As indicated by the gap analysis in Chapter 2, these two intended types of customization are not supported by the current PaaS offerings.

Furthermore, the middleware framework is open for existing customization mechanisms, if only the customizations are performed at run time. Moreover, SaaS architects and developers should be able to combine multiple mechanisms to support different types and levels of customization simultaneously.

3.1.2 Non-functional requirements

The proposed middleware solution for co-existing variants in SaaS applications should address the following non-functional requirements:

- Low operational costs are important for the SaaS provider in order to make a SaaS offering profitable (see Section 1.2). In this dissertation, we aim to achieve *cost efficiency* by supporting application-level multi-tenancy.

- The additional middleware layer should not introduce a high performance overhead. Thus, *performance* (i.e. throughput and latency) is key during the execution as well as the customization of the SaaS applications.
- As *scalability* is a core characteristic of cloud computing, it is key that the middleware does not compromise this scalability, at the operational as well as the management level.
- One of the benefits of the high degree of distribution is *high availability*. Therefore, distribution and replication of the middleware over multiple nodes (possibly at multiple geographically locations) should be feasible. Furthermore, the failure of some of the middleware components may not hinder its operation or that of the application (i.e. fault tolerance).

Because of its openness and versatility, the middleware framework does not (fully) tackle all these non-functional concerns. Different design decisions have to be made to address these requirements when creating a concrete instance of the middleware, for example depending on the selected application type(s), customization goal(s) and composition mechanism(s). Specific examples of such instances are presented in Chapters 4 and 5.

3.2 Architecture of the middleware framework for customizable multi-tenant SaaS

To enable the customization of multi-tenant SaaS applications, we introduce a framework for an application middleware layer between the underpinning base platform (i.e. a PaaS offering or existing cloud-enabling middleware) and the SaaS application (see Fig. 3.1). The requirements of the different tenants are specified in configurations. Driven by these tenant-specific configurations, the appropriate software variants are dynamically selected and composed into the SaaS application (cf. core customization mechanism in Fig. 3.1). In addition, the middleware layer continuously monitors the application as well as the execution environment, and verifies the results with the different co-existing configurations. If necessary, it responds by reconfiguring the application and/or the execution environment. This control loop (see Fig. 3.1) is necessary, for example, to ensure performance isolation in compliance to tenant-specific SLAs. Decoupling the control loop from the critical execution path is key for the performance and availability of the middleware.

This section presents the architecture of this middleware framework (see Fig. 3.2). The middleware serves as an extension to the underpinning cloud platform. First, we specify the assumptions and requirements with respect to this base platform. Then, we focus on the design of this middleware layer for co-existing variants.

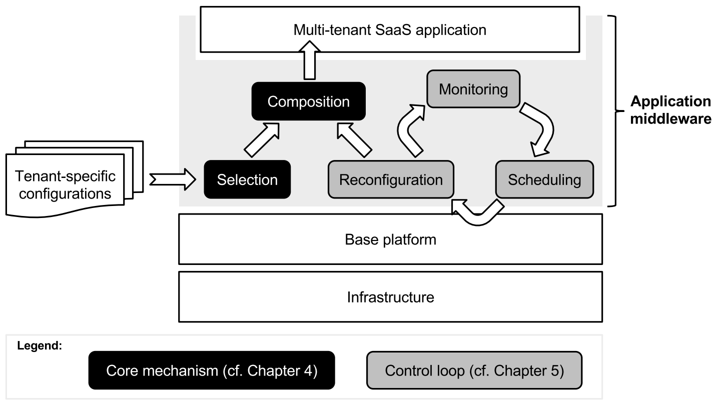

Multi-tenant SaaS application
Composition
Monitoring
Application middleware
Tenant-specific configurations
Selection
Reconfiguration
Scheduling
Base platform
Infrastructure
Legend:
Core mechanism (cf. Chapter 4)
Control loop (cf. Chapter 5)

Figure 3.1: High-level overview of the customization process, consisting of a core customization mechanism and a control loop, in the middleware for co-existing variants in multi-tenant SaaS applications.

3.2.1 Base platform

The middleware framework for co-existing variants relies on a base cloud platform (see bottom of Fig. 3.2) (i) to support distributed and scalable execution, possibly in an asynchronous way, and (ii) for easy and rapid access to application and configuration data across the distributed environment. Thus, the base platform should support (geo-)distribution and replication of application logic and data, as well as load balancing and dynamic resource provisioning in order to ensure high availability and scalability. Furthermore, it should provide (access to) common cloud (middleware) services such as distributed storage, caching and message queues.

Based on our hands-on experience and the analysis of the state of practice (see Chapter 2), we can state that existing PaaS offerings, or IaaS offerings in combination with cloud-enabling middleware, can be used as the underpinning base platform. The majority of these existing platforms fulfill the requirements of a distributed and scalable computing platform and storage service.

Furthermore, a multi-tenancy enablement layer offers basic multi-tenancy support by managing the current tenant context and by facilitating the tenant-aware isolation of data and configuration data. As indicated by the gap analysis in Chapter 2, current offerings hardly provide any support for multi-tenancy. Because from the point of view of the middleware framework we assume the presence of such a layer, we discuss in more detail the design of a basic multi-tenancy enablement layer in Chapter 4. However, some existing platforms do offer built-in support for tenant-aware data isolation, e.g. Google App Engine (GAE) [77].

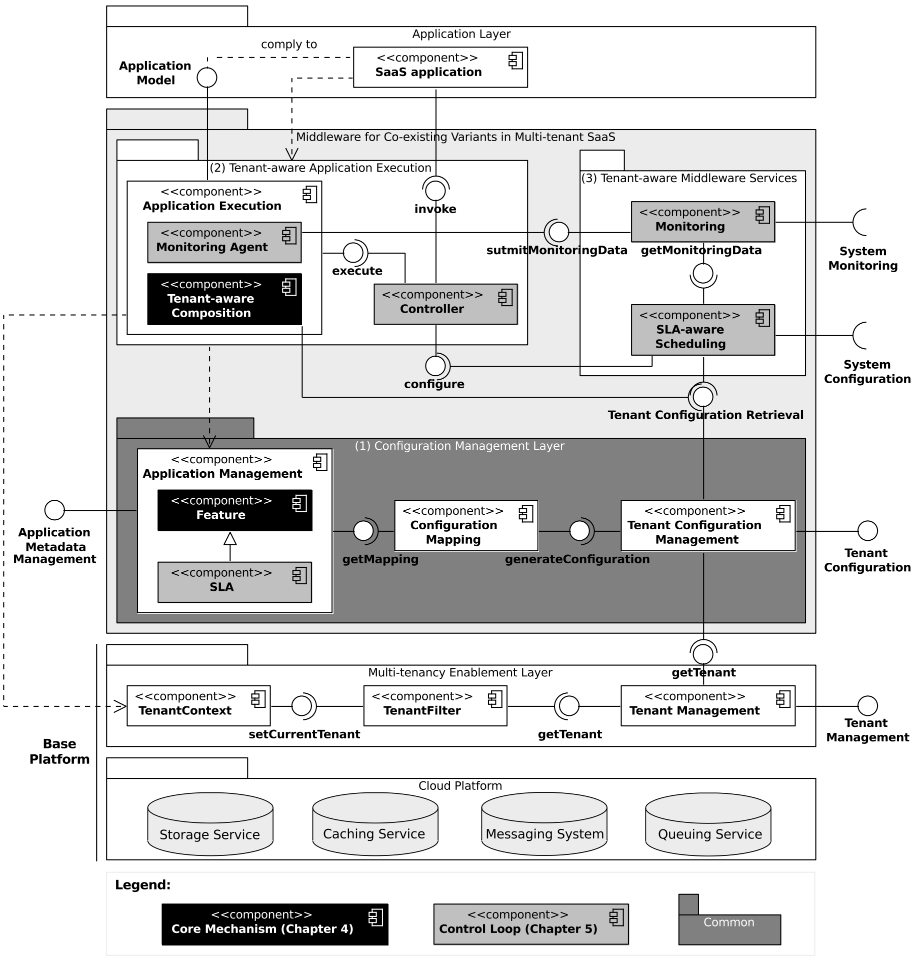

Application Layer
comply to
Application Model
<<component>> SaaS application
Middleware for Co-existing Variants in Multi-tenant SaaS
(2) Tenant-aware Application Execution
<<component>> Application Execution
<<component>> Monitoring Agent
<<component>> Tenant-aware Composition
invoke
execute
<<component>> Controller
configure
sutmitMonitoringData
(3) Tenant-aware Middleware Services
<<component>> Monitoring
getMonitoringData
<<component>> SLA-aware Scheduling
System Monitoring
System Configuration
Tenant Configuration Retrieval
(1) Configuration Management Layer
<<component>> Application Management
<<component>> Feature
<<component>> SLA
Application Metadata Management
getMapping
<<component>> Configuration Mapping
generateConfiguration
<<component>> Tenant Configuration Management
Tenant Configuration
Multi-tenancy Enablement Layer
getTenant
<<component>> TenantContext
setCurrentTenant
<<component>> TenantFilter
getTenant
<<component>> Tenant Management
Tenant Management
Base Platform
Cloud Platform
Storage Service
Caching Service
Messaging System
Queuing Service
Legend:
<<component>> Core Mechanism (Chapter 4)
<<component>> Control Loop (Chapter 5)
Common

Figure 3.2: Architecture of the middleware framework for co-existing variants in multi-tenant SaaS applications. The middleware layer serves as an extension to the underpinning base platform. This base platform (bottom layers) consists of a cloud platform that provides distribution and typical cloud services, and a multi-tenancy enablement layer.

3.2.2 A middleware layer for co-existing variants

The logical view of the middleware layer for co-existing variants consists of the following subsystems:

- the *configuration management facility* provides SaaS providers and tenants with APIs to manage the variability of the application as well as to manage and retrieve tenant-specific configurations,
- the *tenant-aware application execution environment* ensures the actual execution and composition of the multi-tenant SaaS application driven by the tenant-specific configurations,
- a *set of tenant-aware middleware services* monitor the SaaS application on a per-tenant basis and respond in an appropriate way, for example by (re)configuring the application and/or the execution environment.

Configuration management. The configuration management facility is a common layer in the middleware to facilitate the management and configuration of customizable SaaS applications. It offers several interfaces to the different stakeholders and to the other middleware components.

The `Application Management` component enables SaaS providers (the SaaS architect and developer) to manage the overall configuration and variability of the SaaS application. More specificially, via the `Application Metadata Management` interface, the SaaS provider can specify and update the metadata of the different features in the SaaS application, including how each feature maps to the implementation level, e.g. to specific software artifacts.

The `Tenant Configuration` interface provides tenant administrators with a service to customize the SaaS application to their preferences. This customization process is based on the selection and parameterization of features, resulting in a feature configuration. This feature configuration can be updated at any time by the tenant administrator. The `Configuration Mapping` component automatically translates these tenant-specific feature configurations into software-level configurations based on the mappings specified in the `Application Management` component. A version number is associated to these configurations to ensure consistency across the distributed SaaS application (cf. [121]): during the execution of a particular invocation on the application, the same version of the tenant-specific configuration is used throughout the (distributed) application.

Both the feature and the software configurations are managed by the `Tenant Configuration Management` component. This component also offers the `Tenant Configuration Retrieval` interface to the other middleware components.

This common middleware layer is further refined in Chapters 4 and 5, although with different focus and thus different names for the components, but with the same responsibilities. Chapter 4 achieves tenant-specific customization via variability in the business logic of the application (i.e. features map to software variants), while in Chapter 5 the focus is on the enforcement of performance SLAs (a non-functional feature). Chapter 6 describes how this layer supports the configuration process of the service line engineering method as well as the automated mapping of features to software configurations.

Tenant-aware application execution. The core of the middleware layer consists of the environment for tenant-aware execution and customization of the SaaS application. In addition, it also provides the feature-oriented application model that the SaaS developer has to use for implementing the application and its features. As explained in Section 3.1, the specific application model depends on the type of application, for example a component-based versus a workflow-based application.

Every application service or workflow relies on the `Application Execution` component for its execution. This execution is monitored by a `Monitoring Agent`. Furthermore, the `Tenant-aware Composition` component is responsible for deciding which software variants should be active during particular invocations. This component composes the application at run time based on the tenant-specific configurations and the current context, and ensures isolation between the different tenant-specific customizations. Chapter 4 elaborates on this mechanism for tenant-aware composition of software variants.

The `Controller` component controls the order in which invocations are processed. Often this is (or is based on) a load balancer that is part of the underpinning cloud platform. However, to ensure performance isolation (cf. Chapter 5), the middleware should be able to configure this component in order to manage the execution order.

Tenant-aware middleware services. The middleware layer requires an additional set of loosely-coupled middleware services to monitor the application on a per-tenant basis as well as to process the tenant-specific SLAs based on the monitored data. These two components are part of the cross-layer monitoring and cloud management services in Fig. 1.1.

More details on these services are provided in Chapter 5. More specifically, the focus of Chapter 5 is on performance isolation, thus the `Monitoring` and `SLA-aware Scheduling` components are used to determine whether the delivered performance is in compliance to the tenant-specific performance SLAs. If necessary, the `SLA-aware Scheduling` component reacts by (re)configuring the `Controller`.

However, in extension to the work in Chapter 5, the middleware framework allows using the same components to monitor and manage other non-functional concerns, for example availability and scalability. Evidently, this will require additional interfaces and support from the underpinning cloud platform, for example the `System Monitoring` and `System Configuration` interfaces in Fig. 3.2.

3.3 Versatility

This chapter has presented the overall architecture of the middleware framework for co-existing variants in multi-tenant SaaS applications. This is an open and versatile architecture, as it supports different implementations and deployments. The targeted application type(s), customization type(s), and software qualities introduce constraints on these implementation and deployment options, resulting into specific instances of the middleware (see Table 3.1). For example in Chapter 5, a centralized approach to control the workflow execution is recommended for performance reasons, while a distributed controller typically results in higher availability and scalability.

In the course of this doctoral program, the versatility of this middleware framework has been illustrated in the implementation of different middleware prototypes as well as by the use and extension of these prototypes by several MSc and PhD students in the context of research projects, master theses and courses (see Table 3.2). Moreover, the entire customization process by the middleware framework, consisting of the core customization mechanism as well as the control loop, have been validated in both application cases presented in Section 1.3.

Chapter 4 elaborates on the run-time composition of software variants tailored to the tenant-specific (functional) requirements in the context of multi-tier component-based SaaS applications (e.g. the hotel booking application). A prototype has been built on top of GAE and the run-time composition is achieved using dependency injection (DI) [66]. In addition to DI, we used context-oriented programming (COP) [92] as composition mechanism, and compared both approaches in terms of customizability and performance [204]. A similar comparison has been done by students. They added several new features (of different complexity) to the hotel booking application using one of these two composition mechanisms, and afterwards they filled in a questionnaire to evaluate the composition mechanism as well as the middleware. Furthermore, three students implemented different prototypes on top of JBoss AS [143], Windows Azure [23], and GAE [207], using DI and aspect-oriented programming (AOP). Finally, an implementation of the architecture has been developed by Gey et al. [74] to support customization of both the workflow and the individual services of the document processing application. This prototype runs on top of JBoss AS, and uses jBPM and Drools to enable customization at the workflow level.

Chapter 5 builds further on this by extending the middleware with a control loop to monitor and react on changing context parameters. More specifially, we added support for the enforcement of tenant-specific SLAs to ensure performance isolation in service-oriented (workflow-based) SaaS applications, such as the document processing application. In [213], we developed a middleware instance to support performance isolation in interactive component-based applications. Both solutions run on top of JBoss AS, but implement the `Controller` component differently (cf. Fig. 3.2).

Table 3.1: Overview of the key design decisions and principles (i) for the middleware framework, and (ii) for the two instances of this framework in Chapters 4 and 5.

	Design decisions & principles		
Architectural drivers	**Middleware framework**	**Chapter 4**	**Chapter 5**
Functional			
Application type		Component-based Request-driven	Workflow Batch-driven
Customization	Modularization	Run-time composition	Control loop
Non-functional			
Applicability	Extension to existing cloud platforms		
Availability	Distribution and replication via the base platform		
Cost efficiency	Application-level multi-tenancy		
Fault tolerance	Loose coupling Stateless components		Message-based comm.
Performance	Distributed cache		Centralized workflow engine Lightweight processes Message-based comm.
Scalability	Distributed and scalable base platform Load balancer Stateless components		Task queue

Table 3.2: Overview of the different prototypes that have been developed, extended and used to validate and illustrate the versatility of the middleware framework

Contribution	**Application**	**Platform & technology**	**Context**
Core mechanism	Hotel booking	GAE using DI	Chapter 4 [214], Course, Master thesis [207]
	Hotel booking	GAE using COP	[204], Course
	Hotel booking	GAE using AOP	Master thesis [207]
	Hotel booking	Azure using DI	Master thesis [23]
	Hotel booking	JBoss using DI	Master thesis[143]
	Document processing	JBoss using jBPM and Drools	[74], CUSTOMSS [53]
Control loop	Hotel booking	JBoss AS via load balancer	[213], Master thesis [143]
	Document processing	JBoss AS via task dispatcher	Chapter 5

Chapter 4

A middleware layer for flexible and cost-efficient multi-tenant applications *

Preamble

Chapter 3 has presented the middleware framework to support co-existing variants in multi-tenant SaaS applications. This chapter discusses this framework in more detail by focusing on the middleware support to enable the run-time and cost-efficient composition of software variants in multi-tier component-based applications. However, the general approach can also be applied to customize multi-tenant workflows, as shown by [73, 74]. The main contributions of this work are (i) the design and development of a middleware layer to improve the flexibility of multi-tenant SaaS applications (see Section 4.3), and (ii) the evaluation of this middleware layer showing minimal impact on the operational costs and engineering complexity (see Section 4.4). This evaluation has been performed on top of Google App Engine (GAE) using a prototype implementation of the hotel booking application that is presented in Section 1.3.

*The content of this chapter has been published in the proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware (Middleware 2011) [214].

A middleware layer for flexible and cost-efficient multi-tenant applications

Abstract

Application-level multi-tenancy is an architectural design principle for Software-as-a-Service applications to enable the hosting of multiple customers (or tenants) by a single application instance. Despite the operational cost and maintenance benefits of application-level multi-tenancy, the current middleware component models for multi-tenant application design are inflexible with respect to providing different software variations to different customers.

In this paper we show that this limitation can be solved by a multi-tenancy support layer that combines dependency injection with middleware support for tenant data isolation. Dependency injection enables injecting different software variations on a per tenant basis, while dedicated middleware support facilitates the separation of data and configuration metadata between tenants. We implemented a prototype on top of Google App Engine and we evaluated by means of a case study that the improved flexibility of our approach has little impact on operational costs and upfront application engineering costs.

4.1 Introduction

Context. An important trend in the landscape of service-oriented software has been the rise of the "Software-as-a-Service" (SaaS) delivery model [202] where software applications are created and sold as highly configurable web services. A well-known

SaaS provider delivers for instance a Customer Relationship Management (CRM) application [183] as a configurable service to a variety of customers that each have their specific preferences and required configurations.

SaaS applications differ from traditional application service provisioning (ASP) in the sense that economies of scale play a much more important role. A traditional application service provider typically manages one dedicated application instance per customer. In contrast, SaaS providers typically adopt a *multi-tenant architecture* [44], meaning that a shared application instance hosts multiple customers, which are called tenants. The primary benefit of this approach is that the *operational costs can be significantly reduced*: (i) hardware and software resources can be more cost-efficiently divided and multiplexed across customers, and (ii) the overall maintenance effort is seriously simplified because upgrading the application software can be performed for all tenants at once.

Problem. Application-level multi-tenancy comes however also with a number of disadvantages. More specifically, in this paper we focus on two challenges when implementing multi-tenancy at the application level. First *application engineering complexity* is increased. The engineering of multi-tenant application software is more complex than traditional single-tenant applications that are deployed per individual tenant. The primary cause is that the application developer should take measures to ensure isolation between different tenants with respect to the application configuration and data of each tenant [84]. Moreover, a tenant-specific management facility needs to be created such that application configuration management per tenant is separated from the core application management by the SaaS provider.

Secondly, in order to meet the unique requirements of the different tenants, the application must be *highly configurable and customizable.* The current state of practice in SaaS development is that configuration [44, 84] is preferred over customization which is considered too complex [201]. Configuration usually supports variance through setting pre-defined parameters for the data model, user interface and business rules of the application. Customization on the other hand involves software variations in the core of the SaaS application in order to address tenant-specific requirements that cannot be solved by means of configuration. Compared with configuration, customization is currently a much more costly approach for SaaS vendors because it introduces an additional layer of application engineering complexity and additional maintenance overhead.

Approach & contribution. This paper presents a software development and execution platform[1] for building and deploying customizable multi-tenant applications, narrowing down the gap between configuration and customization. More specifically, we present a multi-tenant middleware layer on top of Platform-as-a-Service (PaaS)

[1]Other aspects of SaaS applications such as SLA management, metering and billing are out of the scope of this paper.

platforms that (i) supports improved customization flexibility, (ii) preserves the operational cost benefits of the application-level multi-tenancy principle, and (iii) frees the application developer from a lot of initial application engineering costs for multi-tenancy.

We implement our middleware layer on top of Google App Engine (GAE) [77]. We extend the Guice dependency injection framework [78] with support for tenant-specific activation of software variations and use the scalable and high-performance datastore of GAE for storing and isolating tenant-specific application metadata. We evaluate the feasibility of our middleware layer by comparing a standard single-tenant and multi-tenant application with a flexible version that is developed using our middleware layer. This shows that the impact of our middleware layer on operational costs and additional application engineering complexity is minimal.

Structure of the paper. The remainder of this paper is structured as follows. Section 4.2 introduces the case study and motivates the need for a middleware that supports true application-level multi-tenancy with improved customization flexibility. Subsequently, Section 4.3 presents the architecture of our middleware layer and its implementation on top of Google App Engine. Section 4.4 presents the evaluation of our middleware architecture in the three dimensions of customization flexibility, operational costs, and initial engineering costs. Section 4.5 elaborates on related work and Section 4.6 concludes the paper.

4.2 Problem elaboration & motivation

This section first explores the design space of multi-tenant applications and positions our intended middleware architecture in this space. Subsequently our work is motivated by means of an application case. Finally, the main requirements for our middleware layer are derived from a customization scenario in this application case.

4.2.1 Multi-tenancy architectural strategies

Multi-tenancy aims to maximize resource sharing among customers of a SaaS application and to reduce operational costs. However different architectural strategies can be applied to achieve multi-tenancy. As shown in Fig. 4.1, multi-tenancy can be realized at the application level, middleware level or virtualized infrastructure level. Each approach makes a different trade-off between (i) minimizing operational costs (including infrastructural resources as well as maintenance cost), (ii) minimizing upfront application (re-)engineering costs, and (iii) maximizing flexibility to meet different customer requirements.

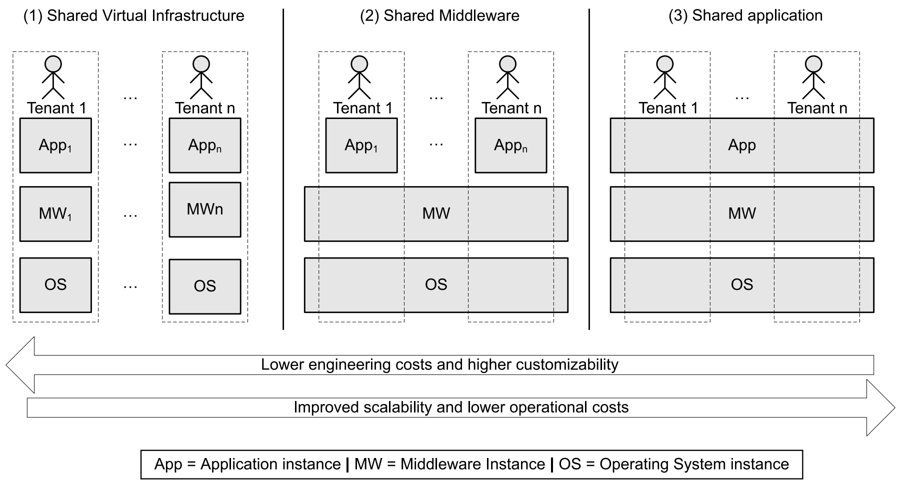
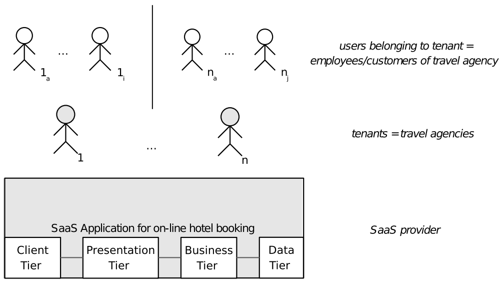

(1) Shared Virtual Infrastructure
(2) Shared Middleware
(3) Shared application
Tenant 1
Tenant n
App₁
Appₙ
MW₁
MWn
OS
App
MW
Lower engineering costs and higher customizability
Improved scalability and lower operational costs
App = Application instance | MW = Middleware Instance | OS = Operating System instance

Figure 4.1: Different architectural approaches to achieve multi-tenancy.

As stated in the introduction, application-level multi-tenancy maximizes the level of resource sharing but is also the least flexible choice with additional engineering overhead. At the other end of the spectrum, virtualization technology can be used to run multiple operating system partitions with dedicated application and middleware instances for each tenant on shared servers. The advantage of this approach is its increased flexibility and low upfront application engineering cost. However, fewer tenants can be hosted on a single server and maintaining separate application instances per tenant also has a much higher cost than with application-level multi-tenancy.

Middleware-level multi-tenancy [17, 34] uses a separate middleware platform that is able to host multiple tenants on top of a shared operating system, which may be either placed on a physical or virtualized hardware. In this way, the initial engineering complexity for multi-tenancy is shifted from the application level to a reusable middleware layer that also offers basic support for isolation of tenants. However, the component and deployment model of these middleware architectures still require that a separate application instance is deployed for each tenant which again implies a higher maintenance cost.

Our proposal is to create middleware support for building true multi-tenant applications with the flexibility to adapt to tenant-specific requirements. Because all tenants are served by the same instance of the application, this means that there is need for tenant-specific software variability in the application components. We assume that such multi-tenant application components do not maintain tenant-specific state, but that all tenant-specific state is stored in a (separate) database. To ensure scalability



users belonging to tenant = employees/customers of travel agency
tenants = travel agencies
SaaS provider
SaaS Application for on-line hotel booking
Client Tier
Presentation Tier
Business Tier
Data Tier

Figure 4.2: SaaS application for online hotel booking.

when user load increases, a pool of identical application instances with our middleware layer have to be created. Existing PaaS platforms already take care of this scalability requirement in a transparent way. For example, Google App Engine automatically scales up (and down) by creating extra instances as the load increases. We therefore propose to incept our middleware layer as an extension for PaaS platforms.

4.2.2 Motivating example

Consider the example of a SaaS provider for online hotel booking (see Fig. 4.2). The SaaS provider offers a highly configurable web service that travel agencies can use for booking hotels and flights on behalf of their customers. Travel agencies play in this example the role of tenant whereas employees and customers of a travel agency are considered the users that belong to a tenant. Employees are offered a customized user interface and customers of the travel agency can login to check the status of the travel items through a URL with a custom-made domain-name that corresponds with the travel agency. A special 'tenant administrator' role is assigned to someone who is responsible for configuring the SaaS application, setting up the application data and monitoring the overall service. This role can be played by an internal or external client of the SaaS provider or even resellers who are an intermediate business proxy. In the context of this simple example, the tenant administrator belongs to the ICT staff of a travel agency company.

4.2.3 Requirements derived from a customization scenario

Suppose that a particular travel agency wants to be able to offer price reductions to their returning customers. As such, the online hotel booking application should be extended with an additional service for managing customer profiles and a service for calculating price reductions. We assume that SaaS providers employ a business model where the base application is offered to tenants at no or low cost, but tenants incur an additional price for additional services. Based on this simple scenario, we can derive requirements with respect to core development, service customization and runtime support.

With respect to development, the application development team of the SaaS provider should be offered a simple way to *manage the different tenant-specific variations* as separate units of deployment that can be selectively bound to the core architecture of the application. Moreover, the overall 'multi-tenancy concern' should be well separated from the application layer.

With respect to customization, tenant administrators should be offered a *configuration facility* to select what software variations should be enabled for them (e.g. the price reduction service). In addition, this facility should also allow to specify specific configuration parameters (e.g. business rules for the price reduction service). This configuration data should be stored in the datastore of the SaaS provider in an isolated way under a specific tenant ID.

The runtime support of the middleware layer must provide support for *injecting software variations on a per tenant basis*. When a user (either customer or employee) logs in, the tenant to which the user belongs should be determined. Based on the acquired tenant ID, the multi-tenant middleware should then activate the appropriate software components to process the requests of the user. Another key requirement of the execution platform is that the tenant-specific software variations should be applied in an isolated way without affecting the service behavior that is delivered to other tenants.

4.3 Middleware support for tenant-specific customization

This section presents the overall architecture of our middleware layer to support tenant-specific customization of SaaS applications. The component model of our middleware layer targets multi-tier applications and structures the application into a core architecture with declared variability points for multi-tenant software variations. Building on top of this component model, the middleware layer consists of a support layer for tenant administrators and run-time support for injecting software variations on a per tenant basis.

In this paper we focus on the customization of component-based multi-tier applications, rather than business processes (e.g. BPEL). The latter requires a different approach where software variations are deployed as separate services, and per tenant a separate business process is responsible for the coarse-grained composition of the

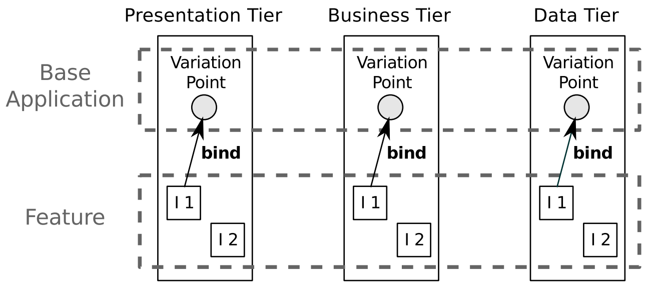


Figure 4.3: Illustration of the feature-based approach.

appropriate services. In the context of component-based applications, dependency injection (DI) [66] is a common composition mechanism. With standard DI however, separate object hierarchies are maintained per tenant in a shared address space which increases heap memory storage and supports only static binding of software variations. Therefore, we prefer a composition mechanism that allows in situ run-time rebinding of variations. This requires an extension to the DI mechanism.

This section is structured as follows. We first propose an extension to the multi-tier component model to make it tenant-aware. Next we describe in depth the architecture of our multi-tenancy support layer. Finally, the prototype implementation of this middleware layer on top of Google App Engine [77] is presented.

4.3.1 Tenant-aware component model

To cope with the different and varying tenant requirements, we apply a *feature-based approach*. Software variations are then expressed in terms of features. A feature is a distinctive functionality, service, quality or characteristic of a software system or systems in a domain [100]. Ideally these features are modular software units that can be easily composed into the base application. As illustrated in Fig. 4.3, variation points are specified in the base application, representing the locations where features should be composed. A feature can have several alternative implementations (e.g. `I1` and `I2` in the figure). Based on the tenant-specific configuration, one of the feature implementations is bound to the variation points across the different tiers.

Our extension to the component model supports the application developers of the SaaS provider to develop features as software modules. For each feature different implementations can be registered. A feature implementation consists of a set of software components (possibly at different tiers) and specifies how these components are bound to the base application. The concept of features is necessary to enable

Listing 4.1: Annotation of a variation point for price calculations.

```
...
 @MultiTenant
 private IPriceCalculatorStrategy priceCalculatorStrategy;
...
```

the SaaS provider to easily ensure the *consistency* of software variations across the different tiers of the SaaS application.

In addition, the developers need to be able to tag the locations in the base application where tenant-specific variation is allowed. To *annotate these variation points*, we introduce a new annotation: `@MultiTenant`. Listing 4.1 shows the annotation of a field with the price calculation service interface. This variation point initiates customization of the online hotel booking application based on the currently applicable tenant-specific configuration, for example price calculation with price reduction. Because a variation point can be bound by different features, the annotation has an optional parameter specifying the feature it belongs to. This enables developers to limit the variation point to a specific feature.

4.3.2 Architecture of the multi-tenancy support layer

The architecture of our middleware layer supporting flexible multi-tenant applications is presented in Fig. 4.4. This support layer consists of a *flexible middleware extension framework* to manage features, specify tenant-specific configurations and to dynamically activate the required variations on a per tenant basis via dependency injection. This approach relies on a *multi-tenancy enablement layer*, offering basic multi-tenancy support and facilitating the separation of data and configuration metadata. Our multi-tenancy support layer serves as an extension to middleware platforms, but especially to Platform-as-a-Service (PaaS) solutions. Possibly such a PaaS already offers built-in support for tenant data isolation.

Multi-tenancy enablement layer

The base for application-level multi-tenancy is isolation between the different tenants, such as isolation of data, performance and faults. To achieve tenant-specific customization the main requirement is isolation of data, more specifically configuration metadata. With the default single-tenant approach, the configuration of an application is specified in a global configuration file. In a multi-tenant context a global configuration file results in a uniform application for all tenants, preventing

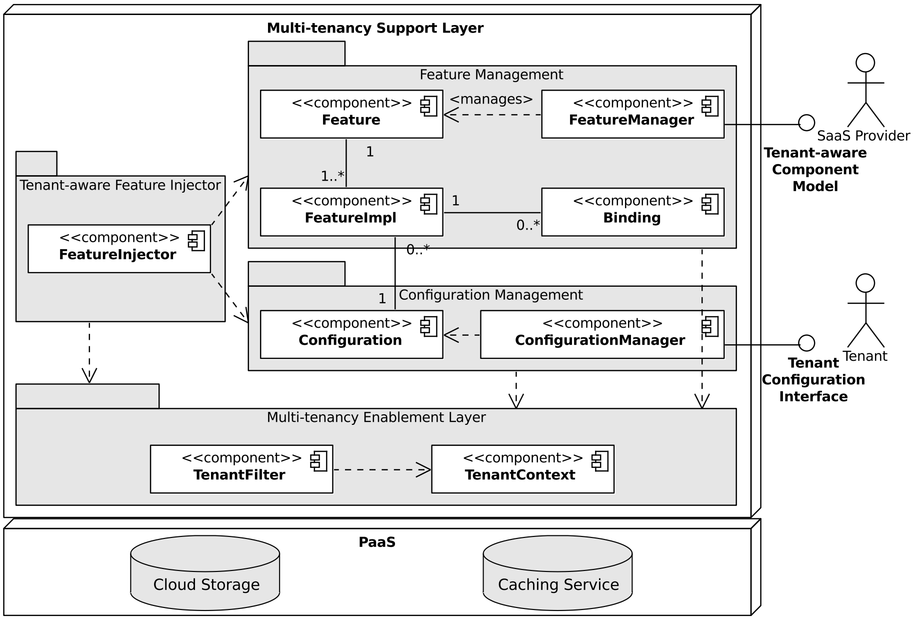

Multi-tenancy Support Layer
Feature Management
<<component>> Feature
<manages>
<<component>> FeatureManager
1
1..*
<<component>> FeatureImpl
1
0..*
<<component>> Binding
Tenant-aware Feature Injector
<<component>> FeatureInjector
0..*
1
Configuration Management
<<component>> Configuration
<<component>> ConfigurationManager
SaaS Provider
Tenant-aware Component Model
Tenant
Tenant Configuration Interface
Multi-tenancy Enablement Layer
<<component>> TenantFilter
<<component>> TenantContext
PaaS
Cloud Storage
Caching Service

Figure 4.4: Overview of the multi-tenancy support layer.

tenant-specific customization. Any change to the configuration would affect all tenants. Therefore tenant-specific configurations have to be stored separately and applied within the scope of a tenant, instead of globally.

To achieve tenant data isolation three main components are required: (i) the *tenant context* containing the information of the tenant linked to the current request (via a unique tenant ID), (ii) *tenant-specific authentication* to identify the tenant, and (iii) *multi-tenant data storage*. Incoming requests are filtered to retrieve the tenant ID (e.g. based on the request URL) and to set the current tenant context. Multi-tenant data storage can be obtained by applying filters that intercept the calls to the storage API and inject the tenant ID from the associated tenant context. In addition, comparable interceptors are necessary for the caching service (distributed in-memory storage). This allows to rapidly retrieve tenant-specific configurations, without large I/O performance overhead.

Flexible middleware extension framework

The flexible middleware extension layer provides the following functionality:

1. a *feature management facility* providing an API to manage the variability of the application and the available feature implementations,
2. a *configuration management facility* to manage the default and tenant-specific configurations,
3. a *feature injector* to dynamically inject the required software variations conforming the tenant-specific configurations.

Feature management. The `FeatureManager` manages the set of available features and their different implementations. A `Feature` specifies at least the following information: a unique identifier (e.g. feature name) and description for the feature, and the set of registered implementations for that feature.

A `FeatureImpl` contains the description of the feature implementation, a set of bindings, and a reference to the configuration interface of this implementation. Each `Binding` specifies the mapping from a variation point to a specific software component. This metadata about the features is globally accessible by both the SaaS provider and the tenants, and therefore should not be isolated. The `FeatureManager` offers a development API to enable the SaaS provider to create and register features and feature implementations, while the tenants are able to inspect the different features via the tenant configuration interface.

Configuration management. Since a feature can have multiple implementations, each tenant can specify its preference for a specific feature implementation via the tenant configuration interface. Such a `Configuration` description defines the mapping from a feature to a specific feature implementation, more specifically from a feature ID to a `FeatureImpl`. The different tenant-specific configurations are then managed by the `ConfigurationManager`. In contrast to the feature descriptions, the tenant-specific configurations are stored on a per tenant basis.

Furthermore, the SaaS provider has to specify a configuration containing for each feature the mapping to a default feature implementation. If a tenant does not specify his tenant-specific configuration, this default configuration will be automatically selected.

Tenant-aware feature injection. Based on the features registered in the `Feature-Manager` and the default as well as tenant-specific configurations, our multi-tenancy support layer has to activate the appropriate feature implementations when required. To achieve this we apply the *dependency injection (DI)* pattern [66]. Instead of instantiating the feature implementations directly in the application, the flow of control is inverted: the life cycle management of feature implementations is controlled by a dependency injector or provider. This injector binds dependencies in the application to an implementation file. Such a *binding* is traditionally but not necessarily a mapping between a type (generally an interface or abstract class) and an implementation type (a class or component). This concept of a binding between a dependency and an implementation corresponds to our `Binding` between a variation point and a software component, as specified in the `FeatureImpls`. As a result, in the above `ConfigurationManager` a tenant-specific configuration corresponds to a specific configuration of the DI framework.

For each variation point in the application the tenant-aware `FeatureInjector` decides at runtime which implementation needs to be used, based on the configuration that applies. First, the `FeatureInjector` intercepts the requests to a dependency and consults the `ConfigurationManager`. The latter queries the multi-tenant data storage using the tenant ID to retrieve the tenant-specific configuration. Subsequently, the right binding is obtained from the `Configuration`, specifying the mapping between the variation point and a specific software component. This software component is instantiated and injected in the application to further handle the request. If the appropriate binding is not available in the tenant-specific configuration, the default configuration is used. In case the feature ID parameter was given, the search to the appropriate binding can be narrowed down to the bindings of a specific feature implementation.

Finally the injected instance is stored in the cache in an isolated way using the tenant ID. For the following requests by this tenant that involve the same variation point, the `FeatureInjector` queries the cache. Using this tenant-aware caching service enables us to support flexible multi-tenant customization of a shared instance without the associated performance overhead.

4.3.3 Implementation

We implemented a prototype of our multi-tenancy support layer on top of Google App Engine (GAE) [77] (SDK 1.5.0), using the Java programming language and the Guice dependency injection framework [78] (v3.0). Google App Engine is a PaaS plaform to build and host traditional web applications developed with Java Servlets and Java Server Pages (JSP). GAE has built-in support for tenant data isolation via the Namespaces API. A separate namespace is assigned to each tenant. We only had to implement a `TenantFilter` to map incoming requests to a specific namespace and to configure that all requests have to go through this filter. For caching we use the Memcache service.

We chose Guice as DI framework because it is type-safe and compatible with GAE. However, it does not support the execution of tenant-specific injections: all dependencies are set globally. Any modification would affect all tenants. This is a general problem with dependency injection because it does not support activation scopes.

To solve this issue, we added an extra level of indirection. Instead of injecting features, we inject a `Provider` for that feature. This way the servlets have a dependency to a provider of a feature instead of to the feature itself. This generic `FeatureProvider` decides based on the tenant-specific configuration which feature implementation should be selected. However, the customizations that can be performed this way are limited to switching between implementations of an interface or abstract class.

4.4 Evaluation

The evaluation of our approach consists of several measurements of the operational and reengineering costs for our multi-tenancy support layer. In particular we want to measure the overhead introduced by the multi-tenancy support layer. We compare the results of our multi-tenancy support layer with a multi-instance, single-tenant approach and the default multi-tenant solution without flexibility.

We first describe the general methodology we applied. Next, a general cost model for the operational and reengineering costs of SaaS applications is specified. Finally we present the measurements we performed and compare the results with our cost model.

4.4.1 Methodology

In this evaluation we measure and compare the operational and engineering costs between a default and flexible single-tenant version, a default multi-tenant version (without flexibility), and a multi-tenant version using our multi-tenancy support layer. For these measurements we use the hotel booking application described in the case study. The source code of these four versions including our multi-tenancy support layer, is available on `http://distrinet.cs.kuleuven.be/projects/CUSTOMSS`.

To determine the operational costs the diferent versions of the application are deployed on top of Google App Engine (SDK 1.5.0), using the high replication datastore (default option). In the case of the single-tenant application, we deploy a separate application for each tenant, while both multi-tenant versions only need one application each. Each tenant is represented by 200 users who each execute a booking scenario. This booking scenario consists of 10 requests to the application: first several requests to search for hotels with free rooms in a given period, then creating a tentative booking in one hotel and finally the confirmation of the booking. The different users of one tenant execute the booking scenario sequentially, while the tenants run concurrently. Notice that it is not our goal to create a representative load for this application, but to compare the operational costs of the different versions under the same load. We retrieve the information about the execution cost via the GAE Administration Console. It provides a dashboard displaying the resource usage by the application. The focus of this comparison is on the relative differences between the execution costs, since the absolute numbers depend on the current (global) load on the GAE platform.

The reengineering costs are compared based on the quantity of source code used to develop the case study application for the different versions. We make a distinction between Java code, JSP pages (for the user interface), and configuration files (XML). The number of source lines of code are determined using David A. Wheeler's 'SLOCCount' application.

4.4.2 Cost model

The goal of the cost model is to define the metrics for our measurements, and to represent our hypothesis about the operational and reengineering costs associated with single-tenant and multi-tenant applications. In addition, it enables us to analyse the impact of customization flexibility on these costs.

Operational costs

The operational cost can be subdivided in (i) the application's execution cost (resource usage), (ii) the costs to maintain the application such as performing upgrades, and (iii) the administration cost, i.e. the cost to provision a new customer (tenant) with an application.

Execution cost. We use CPU time, memory and storage usage as the main execution cost drivers. Another important resource is network bandwidth. However, the introduction of multi-tenancy has no effect on the required bandwidth.

Let *t* be the number of tenants, *u* the number of active users per tenant, and $Cpu(t,u)$, $Mem(t,u)$ and $Sto(t,u)$ the total usage of respectively CPU, memory and storage. Then, in the case of a single-tenant application (ST),

$$\begin{aligned} Cpu_{ST}(t,u) &= t * f_{CpuST}(u) \\ Mem_{ST}(t,u) &= t * (M_0 + f_{MemST}(u)) \\ Sto_{ST}(t,u) &= t * (S_0 + f_{StoST}(u)) \end{aligned} \tag{4.1}$$

where $f_{CpuST}(u)$, $f_{MemST}(u)$ and $f_{StoST}(u)$ are functions of *u*, representing the usage of CPU, memory and storage by one single-tenant application instance. M_0 and S_0 are constants for the memory and storage usage by an idle instance.

In the multi-tenant case (MT) we introduce an extra parameter *i*, i.e. the number of identical multi-tenant instances managed by a load balancer (see SaaS maturity level 4 in [44]). Then,

$$\begin{aligned} Cpu_{MT}(t,u,i) &= t * (f_{CpuST}(u) + f_{CpuMT}(u)) \\ Mem_{MT}(t,u,i) &= i * M_0 + t * f_{MemST}(u) + f_{MemMT}(t) \\ Sto_{MT}(t,u,i) &= S_0 + t * f_{StoST}(u) + f_{StoMT}(t) \end{aligned} \tag{4.2}$$

where $f_{CpuMT}(u)$ is a function of *u*, representing the additional CPU necessary for tenant-specific authentication and isolation of the incoming requests. $f_{MemMT}(t)$ and $f_{StoMT}(t)$ are functions of *t* for the additional memory and storage required to store (global) data about the tenants, for instance the tenant's name and address.

Since the number of multi-tenant instances is limited compared to the number of tenants and the additional amount of memory and storage for multi-tenancy support is relatively small compared to the shared amount of memory and storage (M_0 and S_0), this results in:

$$\begin{aligned} i &\ll t \\ f_{MemMT}(t) &\ll (t-i) * M_0 \\ f_{StoMT}(t) &\ll t * S_0 \end{aligned} \tag{4.3}$$

Thus from Equations (4.1), (4.2) and (4.3), we can compare the execution costs of the single-tenant and multi-tenant versions:

$$\begin{aligned} Cpu_{ST}(t,u) &< Cpu_{MT}(t,u,i) \\ Mem_{ST}(t,u) &> Mem_{MT}(t,u,i) \\ Sto_{ST}(t,u) &> Sto_{MT}(t,u,i) \end{aligned} \tag{4.4}$$

As a result a multi-tenant application consumes less storage and memory than a single-tenant application, but requires more CPU. However, the latter is limited to authenticating the tenant and ensuring isolation.

Maintenance cost. The maintenance cost largely consists of the cost to develop and deploy upgrades to the application. Let f be the upgrade frequency, *i* the number of instances to upgrade, and $Upg(f, i)$ the total upgrade cost, then:

$$\begin{aligned} Upg_{ST}(f, t) &= f_{DevST}(f) + t * f_{DepST}(f) \\ Upg_{MT}(f, i) &= f_{DevST}(f) + i * f_{DepST}(f) \end{aligned} \tag{4.5}$$

where $f_{DevST}(f)$ and $f_{DepST}(f)$ are functions of *f*, representing the development and deployment cost of one single-tenant application instance. The number of single-tenant instances equals the number of tenants *t*. Often there is only one multi-tenant application instance that is automatically cloned to spread the load over multiple identical instances, resulting in *i* being equal to 1. Besides the application, the multi-tenancy support should also be upgraded, but since this is part of the middleware it should not be taken into account here.

Administration cost. For the SaaS provider the administration cost consists of two constant costs: (i) creating and configuring a new application instance (A_0), and (ii) provisioning a new tenant with an application (T_0), for instance by registering the tenant ID in the application and providing a URL to access the application. Let *t* be the number of tenants, then:

$$\begin{aligned} Adm_{ST}(t) &= t * (A_0 + T_0) \\ Adm_{MT}(t) &= A_0 + t * T_0 \end{aligned} \tag{4.6}$$

Reengineering costs

When migrating an application to the cloud, reengineering is required to make use of the available cloud services, for example storage. In addition, making an application multi-tenant results in an additional reengineering cost. The latter is the difference in reengineering costs between a single-tenant and a multi-tenant application, and is dependent on the middleware platform that is used. For example, when an API for multi-tenancy is provided, this reengineering cost stays limited. Without this support, additional development is required to provide tenant-specific authentication and to ensure isolation between the different tenants.

Impact of flexibility

Our multi-tenancy support layer provides multi-tenant SaaS applications with the flexibility to adapt to the different and varying requirements of the tenants. However, this also has an effect on the operational and reengineering costs.

Operational costs. The tenant-specific configuration of single-tenant applications can be set at deployment time. Therefore the effect of tenant-specific variations have a negligible effect on the execution cost of single-tenant applications. Only the base storage S_0 will increase with the core application and its features. In the case of the flexible multi-tenant application, CPU usage $f_{CpuMT}(u)$ (see Eq. (4.2)) will increase because the tenant-specific configuration should be retrieved and activated by the `FeatureInjector`. Further, additional memory ($f_{MemMT}(t)$) and storage ($f_{StoMT}(t)$) is required to store this tenant-specific configuration and the different feature implementations. Though, these differences are not in such quantity that they will affect Eq. (4.4).

The impact of adding flexibility on the maintenance cost will be especially noticeable in the upgrade frequency *f*, because the features also have to be maintained. Since the tenant-specific configuration of a single-tenant application is set at deployment time, changes to this configuration will require additional work for the SaaS provider (C_0). We add an extra parameter *c*, the (average) number of tenant-specific configuration changes which cannot be done by the tenant. Tenants of a multi-tenant application can set their tenant-specific configuration themselves. This results in no maintenance overhead for the SaaS provider.

$$Upg_{ST}(f, t, c) = t * (f_{UpgST}(f) + c * C_0) \tag{4.7}$$

For the administration cost, flexibility only affects the initial configuration of the application (A_0 in Eq. (4.6)) for both versions. In the single-tenant case this consists of setting the tenant-specific configuration, while the SaaS provider needs to specify the default configuration for the multi-tenant application.

Reengineering costs. To add the necessary flexibility, multi-tenant applications require development support for the application developers, support to retrieve and activate the tenant-specific configurations when needed, and a configuration interface to let tenants specify their configuration based on the set of available features. Our multi-tenancy support layer provides this support: multi-tenant applications only have to interact with it. This still results in additional but limited reengineering cost, for example to define the variation points, register the features and specify the default configuration. In a single-tenant application additional reengineering is only needed to facilitate the instantiation of a tenant-specific configuration.

Providing tenants with the flexibility to customize the application, also requires the development of the different software variations. However, this is part of the core application development cost and therefore is not taken into account as reengineering cost.

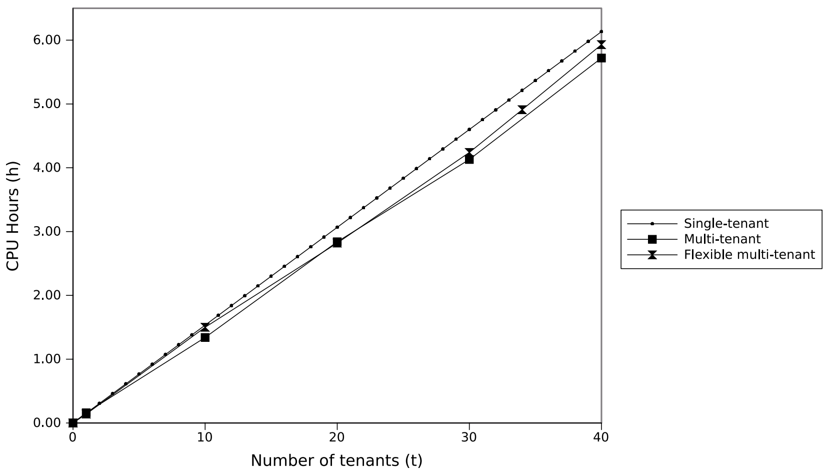


Figure 4.5: Overview of the CPU usage by the different versions.

4.4.3 Measurements

We focus on the execution cost of running the different versions on top of Google App Engine, and the reengineering cost. Since the maintenance and administration costs are hard to measure, we refer to our cost model for more details.

Execution cost. To determine the execution cost we run the four different versions of our case study application on top of GAE: a single-tenant version, a multi-tenant version, a single-tenant version with variability, and a multi-tenant version using our multi-tenancy support layer. However, we noticed that there is no difference in execution cost between the two single-tenant versions, since all variability is hard-coded. Therefore we only show the results of the default single-tenant version. Furthermore, the storage cost is not measured. Because the case study is not a data intensive application, data usage is too limited to make any conclusions about the storage cost.

In Fig. 4.5 we present the evolution of the average CPU usage with an increasing number of tenants. The CPU usage by the single-tenant version is linearly proportional to the number of tenants, as in Eq. (4.1). We also notice that the CPU usage by both multi-tenant versions is also rather linear, but lower than the single-tenant application, which differs with our cost model (see Eq. (4.2)). However, our cost model represents the usage of CPU by the application, while on GAE the CPU time for the runtime environment is included. This is an additional cost per application and therefore has more influence on the single-tenant version. We can conclude that the

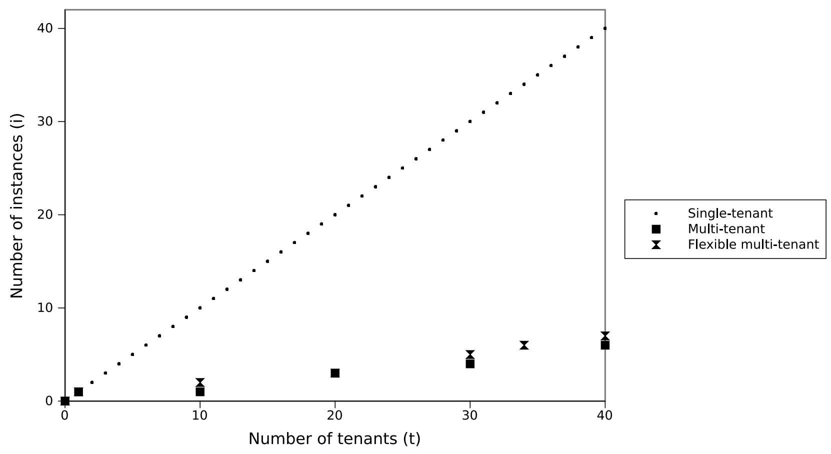


Figure 4.6: Overview of the number of instances used by the different versions.

multi-tenant versions require less CPU time than the single-tenant application, and that our multi-tenancy support layer shows limited overhead compared to the default multi-tenant version.

The total memory usage cannot be measured precisely, because several other factors despite the application binaries add or reduce memory consumption: a rising number of requests triggers an increase in memory because a new instance (i.e. process required to handle the incoming requests) is started to provide better load balancing, and once the requests decline, instances become idle and are removed to release memory (M_0 in Eq. (4.1) and (4.2) is 0). Therefore, we use the average number of instances to represent the maximal possible memory usage. Figure 4.6 shows the evolution of the average number of application instances when increasing the number of tenants. As can be seen, the difference between the single-tenant and multi-tenant versions is significant. The number of instances for both multi-tenant versions increases only slightly with the number of tenants.

Reengineering cost. Table 4.1 shows the quantity of source code used to develop the case study application. The engineering cost to develop multi-tenancy support is not taken into account, because this is part of the middleware. The differences in lines of source code between the single-tenant and multi-tenant versions is the reengineering cost required to let the application use the multi-tenancy support.

In the default multi-tenant version without flexibility, the developer only has to write 8 extra lines of configuration compared to the single-tenant version. This is to specify that the `TenantFilter` should be used, which uses the Namespaces API of Google

Table 4.1: Overview of the source lines of code (sloc) of the different versions

	Java	JSP	XML (config)
Default single-tenant	915	514	131
Default multi-tenant	915	514	139
Flexible single-tenant	1016	514	131
Flexible multi-tenant	1090	514	74

App Engine to ensure data isolation.

When using our multi-tenancy support layer, the difference with the flexible single-tenant application is bigger. However, the majority of these 74 extra lines of Java code are required to use Guice, and not to use our layer. Moreover, the use of Guice resulted in a decrease of configuration lines. Furthermore, in the flexible single-tenant version the configuration is hardcoded and not user friendly. Making this more accessible for the developers to configure will result in more reengineering cost. Finally, we can conclude that adding flexibility to multi-tenant applications by means of our multi-tenancy support layer requires a limited reengineering cost. This cost consists of creating and registering features and their feature implementations, and defining the default configuration.

4.5 Related work

Related work can be divided into three domains: a) middleware support for developing multi-tenant applications, b) work on customization of multi-tenant SaaS applications, and c) adaptive middleware.

Middleware support for multi-tenancy. Multi-tenancy is a key enabler to deliver SaaS applications with high cost effectiveness. The current state of the art especially focuses on approaches to support isolation in multi-tenant software applications [34, 84]. For instance, Guo et al. [84] discuss design and implementation principles for application-level multi-tenancy, exploring different approaches to achieve better isolation of security, performance, availability and administration among tenants.

Only a few Platform-as-a-Service (PaaS) solutions offer support to build multi-tenant applications. Google App Engine (GAE) [77] facilitates the development of multi-tenant applications via the Namespaces API. Application data is partitioned across tenants by specifying a unique namespace string for each tenant (the tenant ID). These namespaces are supported by several GAE services, such as the datastore and the caching service, enabling tenant data isolation in a transparent way. The Namespaces

API is also supported by GAE's open-source implementation AppScale [43]. Other PaaS platforms supporting tenant data isolation are Apprenda SaaSGrid [10] and GigaSpaces SaaS-Enablement platform [76]. None of these platforms directly support tenant-specific customizations and therefore do not offer the same flexibility as our solution. Note that these platforms can also be used as underpinning PaaS for our approach.

In the traditional middleware space JSR 342, the Java EE 7 Specification [58], aims to enhance the suitability of the Java EE platform for cloud environments, including support for multi-tenancy. A descriptor for application metadata will enable developers to describe certain cloud-related characteristics of applications, for example by tagging them as multi-tenant or by specifying the sharing of resources. This extension of the component model with cloud-specific application metadata focuses on persistence and security. Our multi-tenancy support layer, however, offers a way to annotate points of tenant-specific variation, increasing the flexibility of multi-tenant applications, and thus is complementary.

Customization of multi-tenant SaaS applications. Although tenant-specific customizations are an important requirement [25, 44, 201], it is not trival to adapt the business logic and data to the requirements of the different tenants [84], especially in Java or .NET, the programming languages commonly used for enterprise applications.

Bezemer et al. [25] applied their multi-tenancy reengineering pattern to enable multi-tenancy in software services. This pattern requires three additional components: a multi-tenant database, tenant-specific authentication and configuration. Configuration is however limited to the look-and-feel and workflows.

In [141] variability modeling techniques from software product line engineering (SPLE) [168] are applied to support the management of variability in service-oriented SaaS applications. Application templates describe the variability via variability descriptors. Our work focuses on the realization of tenant-specific customizations in SaaS applications, which is not covered by this work.

Existing approaches for dynamic customization of multi-tenant SaaS applications utilize dynamic interpreted languages [147, 183]. However, we focus on customization of enterprise multi-tier applications, which are commonly written in statically typed languages such as Java or C#. In this context, a dynamic software adaptation approach such as dynamic aspect weaving or dynamic component reconfiguration is preferred.

Adaptive middleware. The state of the art in adaptive middleware [31, 52, 103, 111, 180] has mostly focused on adapting applications to one usage context at a time. This means that application software is adapted by replacing an old configuration to a new configuration. In other words, the existing configuration interfaces of adaptive middleware are inherently oriented towards the dimension of the application owner or end user, but have no good ways of managing software variations on behalf of tenants. Adaptive middleware techniques include reflection and aspect-oriented development. The following paragraphs more closely relate our work to these two techniques.

Reflective middleware platforms, such as DynamicTAO [104] and OpenORB [52], provide a configuration interface to inspect and adapt the structure of applications and middleware at runtime. However, these adaptations are based on a global configuration and result in the replacement of components, thus affecting all tenants. They do not allow adaptations scoped to a specific tenant.

Aspect-oriented frameworks such as JAC [165], JBoss AOP [96] and Spring AOP [199], have improved the modularization and customization capabilities of middleware platforms and applications. By means of a declarative configuration application-specific or user-specific extensions can be weaved in where necessary. Currently also dynamic and distributed aspect weaving are supported [111, 165, 179], including in a reliable and atomic manner [151, 205]. These AO-techniques are therefore suitable for usage in a multi-tenant context. Lasagne is an aspect-oriented middleware [206] that supports concurrent, co-existing configurations of the same application instance. This approach is however limited to traditional client-server architectures and does not support customization of multi-tenant software. Still, aspect-oriented software development (AOSD) [65] looks a promising alternative for dependency injection to support tenant-specific injections of crosscutting feature implementations.

4.6 Conclusion

This paper presented a reusable middleware layer on top of an existing PaaS platform to support customizable multi-tenant applications while maintaining the operational cost benefits of true application-level multi-tenancy. We have implemented a prototype on top of Google App Engine and extended the Guice dependency injection framework to achieve activation of software variations on a per tenant basis. This prototype shows improved flexibility with a minimal impact on operational costs for the SaaS provider.

Dependency injection proved to be useful to support the customization of multi-tenant applications. However, adding new features requires the introduction of new variations points in the core application. In addition, for each variation point only one software variation can be injected at a time. This complicates more advanced customizations, such as feature combinations. In this respect, AOSD is a more powerful alternative which we will investigate in the future.

A future research challenge with respect to application-level multi-tenancy is adding support for tenant-specific monitoring and ensuring performance isolation between different tenants. When performing our measurements we experienced that GAE lacks performance isolation between the different tenants. Especially when a number of tenants heavily uses the shared application, this results in a denial of service for the end users of certain tenants. Additional support from the operating system and middleware layers is needed to ensure this performance isolation. Furthermore, tenant-specific monitoring enables SaaS providers to better check and guarantee the necessary SLAs.

Acknowledgements

We thank the reviewers for their helpful comments to improve this paper. This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund KU Leuven.

Chapter 5

Middleware for performance isolation in application-level multi-tenancy [*]

Preamble

This chapter builds further upon Chapter 4 in the sense that it extends the middleware with support for performance isolation in service-oriented multi-tenant SaaS applications in compliance to tenant-specific service level agreements (SLAs). Thus, it realizes the control loop that has been presented in the overall architecture of the middleware framework (cf. Chapter 3), in order to monitor the performance of the application and the execution environment, and to reconfigure these appropriately if necessary. The main contributions of this reusable middleware architecture for run-time SLA enforcement (see Section 5.3) are (i) its support for co-existing performance requirements of the different tenants, (ii) its focus on multi-tenant SaaS applications providing high performance, cost efficiency, scalability and fault tolerance, and (iii) its openness for different algorithms to support different types of applications. A prototype of this middleware layer has been implemented on top of a private cloud platform, based on a JBoss AS cluster. The evaluation in the context of the document processing SaaS application (as introduced in Section 1.3) shows limited performance overhead (see Section 5.4). A preliminary version of this work [213] has been evaluated using the hotel booking application (cf. Section 1.3).

[*] The content of this chapter has been bundled in a paper draft, which has been submitted to the International Conference on Middleware (2014).

Middleware for performance isolation in application-level multi-tenancy

Abstract

SaaS providers typically adopt a multi-tenant architecture to leverage economies of scale: by maximizing the sharing of resources among multiple customer organizations, called tenants, operational costs are reduced, and application management and maintenance can be simplified. Application-level multi-tenancy maximizes the level of resource sharing but complicates performance isolation between tenants. Supporting different tenant-specific variants and, in particular, ensuring compliance with co-existing performance requirements both remain a challenge.

This paper presents a middleware architecture that enables SaaS providers to efficiently address co-existing performance requirements in service-oriented multi-tenant SaaS applications. It can manage a combination of performance SLAs in terms of latency, throughput and deadlines, and enables rapid response on changing circumstances, while preserving the resource usage efficiency of application-level multi-tenancy. Furthermore, it is open to specific policies and (scheduling) algorithms. We have implemented a prototype on top of a private cloud platform, based on OpenStack and JBoss. The evaluation shows the effectiveness of our solution in the context of an industry-relevant SaaS case study, with a very small performance overhead.

5.1 Introduction

Software as a Service (SaaS) is a software deployment model that delivers software applications as online, on-demand services. The obvious driver is leveraging economies of scale by maximizing resource utilization through sharing. Typically, SaaS providers adopt a multi-tenant architecture to achieve these economies of scale [44, 84]. Multi-tenancy is an architectural design principle that enables SaaS providers to achieve high operational cost efficiency by sharing hardware and software resources among multiple customer organizations, called tenants, and by simplifying the maintenance and management effort of the software system.

Multi-tenancy can be realized at different levels: (i) at infrastructure level via virtualization, (ii) at the middleware level by sharing the operating system and middleware, and (iii) at the application level. Maximum cost efficiency is achieved in principle in the case of *application-level multi-tenancy* [44, 214]: the underpinning infrastructure, operating system, middleware, and the application are shared between the different tenants. End users from different tenants are simultaneously served by a single application instance on top of shared infrastructure [84]. Moreover, updates only have to be applied to the shared application instance, giving all tenants automatically access to the most recent version of the application.

However, this high degree of resource sharing complicates *performance isolation* between the different tenants. The objective of performance isolation includes ensuring fairness when distributing resources among the different tenants, i.e. the behaviour of one tenant cannot adversely affect the performance of the other tenants [84, 107]. In addition, the SaaS provider must ensure compliance with the different, tenant-specific (performance-related) service level agreement (SLA). These SLAs can differ depending on the tenants' requirements, typically resulting in a variety of price settings. Our work addresses the creation of an open, efficient middleware framework that extends the stack of system-level support software and that facilitates this type of enforcement by offering all the necessary mechanisms, while being open to strategies by plugging in scheduling algorithms that can be specific for a particular SaaS application setting.

This type of enforcement is typically not supported in standard middleware platforms (being Platform as a Service (PaaS) or more general-purpose cloud-enabling middleware technology). Yet it is unrealistic to expect SaaS providers re-engineering a specific solution in each new application type. Furthermore, current solutions for performance management in SaaS are based on load prediction, workload planning, resource provisioning, etc., typically in terms of lower-level system parameters. Such approaches are not sufficient to support performance isolation in multi-tenant SaaS applications, as resources are shared (and thus cannot be allocated to specific tenants) and the process of workload planning and dynamic resource provisioning is not responsive enough to address unexpected high-priority requests (possibly resulting in SLA violations).

This paper presents a middleware architecture to support SaaS providers to build and deploy multi-tenant SaaS applications, endowed with SLA-driven performance isolation. More specifically, we provide an architecture for a middleware layer that complements PaaS platforms and that (i) enables the run-time enforcement of tenant-specific SLAs over distributed environments for multi-tenant SaaS, while (ii) preserving high performance as well as the operational cost benefits of application-level multi-tenancy , and that (iii) can be reused, thus assisting the SaaS provider when developing SLA-aware multi-tenant SaaS applications. This middleware is an enabling technology for performance isolation in that it offers a framework with application-level monitoring and management facilities, while being open to specific policies and (scheduling) algorithms. We have implemented this middleware on top of a private PaaS platform that has been built using OpenStack, a JBoss AS cluster and an Apache Cassandra database cluster. We evaluate the effectiveness of our solution by comparing our SLA-aware approach with a default system implementing a FIFO strategy, and with an approach implementing fair distribution. In addition, we measure the performance overhead to show that the impact of our middleware architecture is small and acceptable.

The remainder of this paper is structured as follows. Section 5.2 presents a case study that motivates the importance of SLA-driven performance isolation in multi-tenant SaaS applications and defines the requirements for a middleware solution. Section 5.3 presents the architecture of the proposed middleware layer. In Section 5.4, our middleware is evaluated with respect to its effectiveness and to the additional performance overhead. Section 5.5 discusses related work and Section 5.6 concludes the paper.

5.2 Case study & challenges

This section elaborates on a multi-tenant SaaS application in the domain of online business document processing, and identifies the main challenges for this paper driven by a set of scenarios that illustrate the requirements for performance isolation.

5.2.1 Business document processing as an online service

A SaaS provider offers B2B document processing facilities to a wide range of companies. This multi-tenant SaaS application supports the creation, the business-specific processing, the archival and the delivery of various types of business documents, such as invoices, payslips and leaflets. In addition, members of the customer organizations (i.e. the tenants and their respective end users) can view and manage documents via a web-based interface, and download these documents from a storage service. This SaaS application is deployed on top of a private cloud platform that is managed by the SaaS provider.

Application model. The SaaS application implements a service-oriented architecture, consisting of several software services that can execute the different steps of a document processing application. These so-called subservices are distributed over multiple nodes to enable scaling. The core abstraction in the application architecture is the concept of a *task*. Many cloud applications are modelled as a collection of interdependent tasks that synchronize indirectly via a workflow engine, and/or via a common data structure containing intermediate results (Fig. 5.1). Many examples show that asynchronous application models enable scalability (e.g. [3, 115, 120, 126, 134]). Each step in the application corresponds to a *task type*, for example document generation and archival. The sequence of task types is described by a *workflow*. Different workflows exist to address the different tenant-specific requirements. A *job* is an instance of a workflow, containing the full description of work that is assigned to the SaaS application by a particular tenant. It consists of one or more tasks to be executed. Each task refers to a specific task type and contains a set of input parameters (e.g. input file and output format). A software subservice of the application (called *worker*) is responsible for executing tasks of a particular type. When an end user of a tenant submits a job to the application, the different tasks of the job are placed into a queue. The workers asynchronously retrieve tasks (of a specific task type) from the queue and execute these tasks. Workers can transparently be added or removed at run time for up- and downscaling.

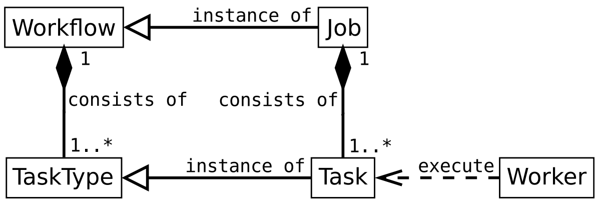


Figure 5.1: Application model of the document processing application.

Workflow example. A basic workflow of document processing consists of the following tasks (as depicted in Fig. 5.2). First, an end user of a tenant starts a new job by submitting a batch of raw input data via the portal of the document processing application. Upon reception, the `BatchPreprocessing` subservice parses and subdivides the input data into multiple raw data records with associated metadata. The latter contains information about the expected output document and the intended recipient(s) (e.g. name and address). In the document generation phase, each data record is separately transformed into the correct output format(s) by first applying a template and then rendering the document. The final steps of the basic workflow consist of packaging and digitally archiving the output documents. More sophisticated workflows include document signing and distribution via the appropriate delivery channel (e.g. printing, fax or email). Notice that multiple jobs of different tenants are processed concurrently by the shared, multi-tenant subservices of this SaaS application.

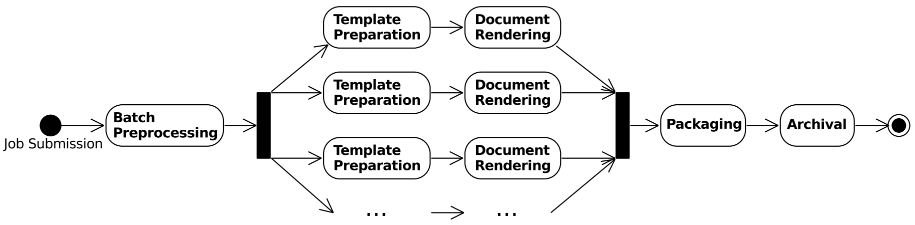


Figure 5.2: Example of a basic document processing flow (consisting of five task types) for a single job. In a multi-tenant SaaS application, multiple jobs of different tenants are processed concurrently.

5.2.2 Illustration: Scenarios for SLA-driven performance isolation

The document processing application processes different types of workflows for different tenants. Each of the workflows may impose different performance requirements (deadlines, throughput, specific response times for specific jobs, etc.). The following scenarios illustrate the need for performance isolation in multi-tenant SaaS applications.

Scenario #1: Minimum throughput per tenant. A common scenario consists of multiple tenants submitting jobs of different sizes simultaneously, which should be processed as fast as possible (e.g. generating invoices, leaflets, or other administrative documents). In a multi-tenant application, the tasks of these jobs are typically placed in a queue and processed sequentially by the different workers (i.e. FIFO). Although multiple jobs and tasks can be processed concurrently, it basically means that each tenant has to wait until the tasks of previously submitted jobs are processed (or at least are being processed). Especially in the case of large jobs, the impact on the other (waiting) tenants is large and can thus cause SLA violation. Therefore, to ensure compliance with the different SLAs, the SaaS provider should be able to let his application monitor the throughput of the diferent tenants and dynamically decide which tasks should be processed next. For example, a tenant with a premium throughput SLA should be able to consume a bigger share of the available resources to process his jobs.

Scenario #2: Deadlines for specific jobs. Besides the just-in-time submission, jobs can be of a recurring nature, e.g. processing payslips and invoices at the end of the month, typically in the form of large document batches. The SLAs related to these jobs define a (soft) deadline. Therefore, the load caused by these jobs can be predicted to a certain level, enabling the SaaS provider to plan the necessary capacity upfront. However, in a multi-tenant application the different workers are shared by all tenants and thus additional (unexpected) jobs can be submitted any time. In addition, the exact load is hard to predict and depends on many parameters (e.g. varying amounts of documents to process). Therefore, the progress should be monitored at run time to ensure that the different deadlines are met. In case SLA compliance is in danger, the application should be able to immediately react by dynamically re-prioritizing jobs and/or individual tasks.

Scenario #3: Minimal response time. Sometimes smaller jobs, consisting of only a few input documents, require a guaranteed response time (almost comparable to interactive applications). For example, premium tenants may have a special SLA for a limited number of emergency cases that may occur (e.g. per month) and that should get a prioritized treatment. This scenario can materialize at any time, even during moments of peak loads. Again, the application should be able to prioritize the processing of such a small job across all its workers. Evidently, such an SLA is also associated with a high penalty in case of violation.

5.2.3 Challenges

The different scenarios presented above occur *simultaneously*, resulting in many *co-existing SLAs*. The SaaS provider aims to ensure these different SLAs, while minimizing the resource cost and the penalties. We have identified the following challenges to achieve SLA-driven performance isolation in distributed multi-tenant SaaS applications:

- *Support for co-existing performance requirements:* Tenants can impose performance requirements concerning response time, throughput, deadlines, or a combination of these elements. Therefore, there should be support to simultaneously manage and enforce multiple tenant-specific SLAs.

- *The need for a reusable middleware framework for run-time SLA enforcement:* By providing a reusable middleware framework, SaaS providers are able to (relatively) easily endow their multi-tenant SaaS application(s) with support for SLA enforcement. This middleware takes care of the monitoring of the different (distributed) subservices of the application, and the verification as well as the actual run-time enforcement of the SLAs throughout the SaaS application. In addition, the middleware framework should be open for different algorithms to support different kinds of applications.
- *High performance and cost efficiency:* The middleware solution for SLA-driven performance isolation should select the next job/task to execute rapidly, without having to wait until sufficient resources become idle and without the overhead of workload planning or dynamic resource provisioning. This is critical to enable the enforcement of premium throughput or low response time. Furthermore, the preservation of the cost efficiency (i.e. maximum resource sharing) and scalability benefits of application-level multi-tenancy is also an important non-functional concern. The middleware should have minimal impact on these benefits. Finally, measures should be taken against failures.

In reality, SaaS applications often do not have (immediate) access to unlimited resources. It takes a significant amount of time to allocate extra resources [6, 125], and private cloud environments offer an inherently limited capacity. Sometimes the SaaS provider does not even have sufficient control over the total amount of resources (e.g. on top of a PaaS platform that promises automatic scaling). Under all circumstances, even for very small low-latency jobs and unexpected high-throughput jobs during peak loads, SaaS applications should try to deliver their SLAs, if necessary by taking into account the different penalties that may apply.

5.3 A middleware architecture for tenant-aware SLA enforcement

The proposed middleware solution operates as an indirection between the application logic and the underpinning cloud platform. The concept diagram of this solution is sketched in Fig. 5.3. The middleware for performance isolation services the SaaS application that consists of a workflow description and multiple multi-tenant workers. Furthermore, it relies on distributed storage to ensure availability of data, and on a distributed execution platform, which may be extended with basic support for application-level multi-tenancy. Such a multi-tenancy enablement layer facilitates at least the isolation of tenant-specific data and configuration metadata (cf. [34, 214]). Existing PaaS offerings can be used as underpinning cloud platform, because they fulfill the requirements of a scalable computing platform and storage service, and sometimes also of tenant-aware data isolation (e.g. Google App Engine [77]).

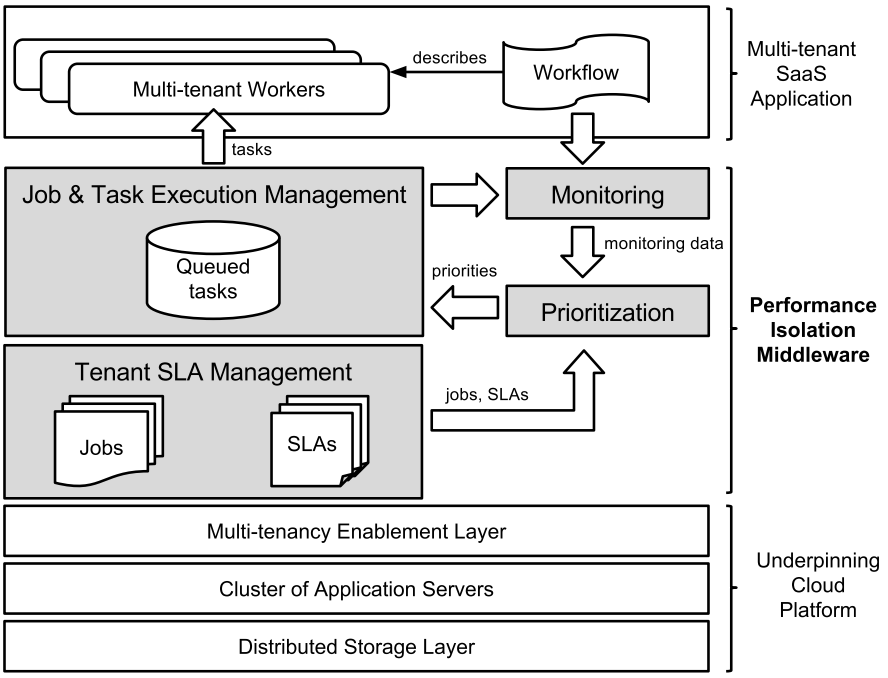

Multi-tenant Workers
describes
Workflow
Multi-tenant SaaS Application
tasks
Job & Task Execution Management
Queued tasks
Monitoring
monitoring data
priorities
Prioritization
Performance Isolation Middleware
Tenant SLA Management
jobs, SLAs
Jobs
SLAs
Multi-tenancy Enablement Layer
Cluster of Application Servers
Distributed Storage Layer
Underpinning Cloud Platform

Figure 5.3: Concept diagram of the proposed middleware solution.

The middleware for enforcing performance SLAs is decomposed in 4 elements, that each fulfill an essential role in achieving the objective of servicing multiple tenants in a fair and SLA-compliant way, while preserving high performance and cost efficiency:

- Decisions are made by the *Prioritization* service. These decisions are based on the jobs and tasks that must be delivered, on the agreements (SLAs) with the corresponding tenants, and on observations of the application and the actual workload. The three remaining elements actually deliver and manage this information:
- The *Tenant SLA Management* layer maintains an overview of the various jobs and the corresponding SLAs that must be delivered for each of the tenants that the SaaS provider is servicing at a particular moment in time. This layer obviously offers external interfaces to inspect and update these configurations and SLAs.
- The *Monitoring* service collects data from the present jobs (workflow instances) and tasks (workers), as well as from the system to assess the workload.

- The *Job and Task Execution Management* layer collects all jobs and therefore also all tasks that must be allocated to the workers in the cloud environment. This layer embeds a centralized, in-memory queue of tasks that must be executed. Selection of tasks occurs by applying the priorities assigned by the above mentioned *Prioritization* service.

The basic operation of the proposed middleware layer is based on a relatively simple concept with various workers (subservices) that retrieve tasks from the Job and Task Execution Management layer. These tasks have been prioritized by the Prioritization service, which in its turn consults the Monitoring Service and the Tenant SLA Management layer to make the appropriate decisions. Notice that the Prioritization service relies on a scheduling algorithm that can be provided and/or customized by the party that instantiates our middleware solution for a specific SaaS context. Communication between these 4 elements occurs in an asynchronous way via a messaging system (pub/sub) or via the distributed storage to minimize coupling and to improve fault tolerance.

In the next subsections, we elaborate on each of the elements introduced above.

5.3.1 Tenant SLA management

The base layer of the SLA enforcement middleware (see (1) in Fig. 5.4) is deployed on every node and consists of a set of facilities to manage the core attributes of our task-driven application model, i.e. workflows, jobs, tasks and tenant-specific SLAs. These attributes are stored and sufficiently replicated in the distributed storage to ensure availability.

The `ConfigurationManager` enables the SaaS provider to manage the configuration of the SaaS application. Such a configuration specifies the different workflows and the various task types, the mapping of task types to service implementations (i.e. workers), the way to access the different middleware services, and the algorithms (and their implementations) for prioritizing tasks.

All jobs (submitted, on-going and finalized) are managed by the `JobManager`. A `Job` specifies at least the following information: a unique identifier, its workflow definition, the tenant owning it, and its creation time. Optionally, tenants can specify input parameters for the execution process (e.g. localizing where raw input data is stored), a start time for the job execution, and they can associate SLAs to the specific job (e.g. a deadline).

The `SLAManager` component is responsible for managing the different SLAs. Every `SLA` specifies its unique identifier, the workflow it applies to, and its type. The middleware supports multiple types of performance SLAs: *response time, throughput*

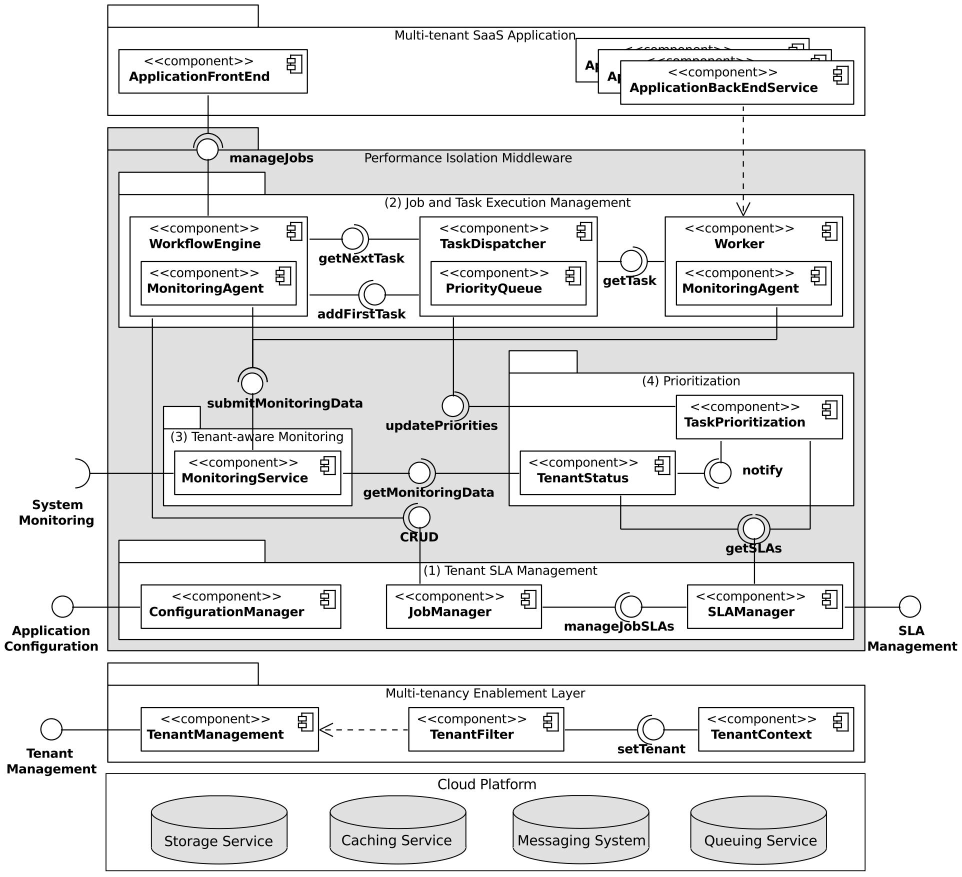


Figure 5.4: Overview of the performance management middleware for multi-tenant SaaS applications. The middleware layer serves as an extension to PaaS offerings. Tenants submit new jobs (e.g. via a front end), which are then executed by the different back-end services of the SaaS application. The middleware decides which tasks may execute.

and deadline, which require different input parameters. Furthermore, these SLAs can apply to a specific job (e.g. a deadline), to all jobs of a particular tenant (e.g. premium throughput), or to an entire application. Optionally, an SLA can define the penalty for the SaaS provider in case of violation, and a limit defining the minimum percentage of jobs the SLA applies to (e.g. "minimum response time for 90% of the jobs").

5.3.2 High-performance job and task execution

The core of the middleware architecture (see (2) in Fig. 5.4) consists of three components that ensure the execution of jobs and their associated tasks: a `WorkflowEngine`, a `TaskDispatcher`, and multiple `Workers`.

Stateless workflow engine. The workflow engine is responsible for the creation and proper execution of workflow instances (i.e. jobs). Therefore, it provides an interface to the application(s) to create and submit jobs, optionally with job-specific SLAs. When a job is submitted, it is persisted using the JobManager and a notification is sent using the underpinning pub/sub infrastructure. Other components, internal as well as external, can subscribe for these notifications, for example to schedule jobs with a specified start time.

The decision to actually start the execution of a job can be taken by the workflow engine itself when the (optional) start time of the job has passed, by the tenant (via the application), or by an external component (e.g. at the scheduled point of time). In Scenarios #1 and #2 (see Section 5.2.2), jobs are typically started immediately after submission. When a job is allowed to start, the workflow engine sents out a notification, creates an instance of the first task type in the job's workflow, and submits it to the task dispatcher (together with its input parameters). In our middleware, the Prioritization service is subscribed for this event to keep track of the active jobs.

Furthermore, the workflow engine manages the execution flow of the different workflows. When a task is finished, it determines the following task in the workflow for the particular job using the appropriate workflow definition. The available workflow definitions are cached in memory to improve the performance of this workflow execution.

Priority-driven task dispatcher. The task dispatcher manages all tasks that are allowed to execute (waiting) and those that are in progress. Concretely, these tasks are stored in an in-memory `PriorityQueue`. By means of such a queue, the task dispatcher controls which of the waiting tasks should execute next. We apply a *centralized and in-memory approach* because it simplifies the design and because of performance reasons. An alternative consists of a distributed queue (e.g. using distributed caching or storage), however this requires strong consistency across the different nodes to ensure tasks are executed only once. Our experiments with the latter approach resulted in poor performance.

Concretely, the `PriorityQueue` offers a getter operation to return tasks based on the priorities (i.e. weights) assigned to the different tasks by the Prioritization service. The higher the priority, the more chance a task will be selected and executed first (cf. weighted round robin). It is critical for the performance and efficient resource usage that the task selection procedure occurs without interruption or waiting: the next task is selected immediately to keep the available workers busy. Notice that this approach is open for extension, allowing the task dispatcher to use queues that take additional constraints into account, such as security policies, cost, and data as well as worker location.

When a task is finished, the task dispatcher retrieves the following task for the particular job from the workflow engine and adds it to the queue. Possible return values of a finished task are added as input for the next task (e.g. (location of) intermediate results). This process is repeated until the whole job is finished; the latter triggers again an event for external subscribers, for example the Prioritization service.

Multi-tenant workers. The actual execution of tasks is performed by `Workers`. These workers can be task-specific (cf. the document processing case study), or uniform (in which case they can execute any task type). The performance isolation middleware provides an API to support the SaaS developer with the development of the workers: each back-end subservice in the SaaS application should extend the existing `Worker` component (see Listing 5.1), and provide the `work` method with an application-specific implementation, for example the generation of documents from raw input data.

Listing 5.1: Template for the implementation of a worker. The SaaS developer should provide a proper implementation for the `work` method. Optionally, the worker can be made task-specific, by assigning a specific task type in the constructor.

```
public class <Worker Name> extends Worker {
 public <Worker Name>(String taskType) {
  super(taskType);
 }

 @Override
 public TaskResult work(Task task) {
  TaskResult result = new TaskResult(task);
  // add application-specific implementation
  ...
  result.addReturnValue(...);
  return result;
 }
}
```

To provide more flexibility with respect to elastic scaling and maintenance, the workers have to request a new task themselves: the task dispatcher should not manage the list of workers that are available for work or that should not receive new tasks (e.g. because they are terminating). When a `Worker` is started, it automatically sends a request for an executable task (of a specific type) to the dispatcher, and when the execution is finished, it returns the result and requests a new task. In case there is no waiting task, the worker will wait for a limited time and send a new request. This is fully managed by our middleware. Notice that the `Worker` component is tenant-aware: the associated tenant ID is passed with every task throughout the application. Using the underpinning multi-tenancy enablement layer, data isolation is ensured during the execution of a particular tenant's task.

In the case of the document processing application, the batch of raw input data is split up in several chunks of data that need to be further processed into separate documents (see Fig. 5.2). With our approach, this means that the workflow engine spawns multiple tasks after the `BatchPreprocessing` task, which can be processed in parallel. In addition, the middleware provides a `JoinWorker` that is able to join the results of multiple tasks into a single result, for example to package multiple sets of newly generated documents.

5.3.3 Tenant-aware monitoring

To enable SLA enforcement in multi-tenant SaaS applications, the middleware keeps track of the performance level delivered to the different tenants. Each `Worker` has a built-in `MonitoringAgent` (see (2) in Fig. 5.4) to observe the execution of tasks. This modular extension does not require any changes to the application code itself. For each task, the tenant ID, job ID and task ID associated with the task are retrieved and different performance metrics can be monitored, for example processing time. Furthermore, a monitoring agent is integrated into the `WorkflowEngine`. The workflow engine is the proper location to measure the end-to-end response time for each job. The monitoring data gathered by the different monitoring agents is stored into the distributed storage (or caching) service of the underpinning platform.

Since the monitoring agents already gather all execution information on the various jobs and their associated tasks, they are the appropriate components to manage the *execution log*. The execution log contains a record of the start and end times of the different jobs and tasks, as well as the current status of the jobs (e.g. waiting for execution, in progress, finalized, failed, etc.). This log is stored and replicated in the distributed storage service. Each log line is stored in a separate row (i) to enable concurrent write access by the different monitoring agents, and (ii) to support querying, for example for recovery or to retrieve failed jobs.

The `MonitoringService` (see (3) in Fig. 5.4) retrieves, via the distributed storage, the monitoring data and aggregates this information to provide an overview of the global resource usage and performance level per tenant and per job. In addition, it can process the historical data on task execution times to calculate the average execution time per task type (e.g. to support the enforcement of deadline SLAs). This aggregated information is persisted into the storage service and can be used to evaluate SLA compliance.

5.3.4 Prioritization

The middleware part responsible for SLA-driven prioritization (see (4) in Fig. 5.4) consists of two components: (i) the `TenantStatus` service that verifies whether the performance level delivered to the different tenants complies with their SLAs, and (ii) the `TaskPrioritization` service that updates the priorities assigned to each tenant, job and task.

`TenantStatus` uses as input the different tenant-specific SLAs as well as the aggregated monitoring data. It continuously analyses the monitoring data to determine for each tenant whether the delivered performance level is in compliance with the applicable SLA(s). Based on the results of this analysis, tenants are assigned the appropriate behavioural *status*, for example aggressive (i.e. tenant consuming much more resources than specified in its SLA(s)), normal, borderline (i.e. SLA(s) almost violated), or penalized (i.e. SLA(s) violated).

If a tenant is classified as aggressive but the performance delivered to the other tenants is not affected by this aggressive behaviour (i.e. no SLAs are violated), this means that the aggressive tenant is using spare resources and no action should be taken. However, if some SLAs are (almost) violated, e.g. because of aggressive tenants or limited resources, the `TenantStatus` service notifies the `TaskPrioritization` service that action should be taken.

The `TaskPrioritization` service determines the priorities of each tenant, job and task, based on the different SLAs, the status labels that are assigned to tenants, and the current set of active jobs (i.e. these jobs have tasks in the queue). The latter is kept up-to-date via the notifications from the workflow engine. Priorities are then translated into weights, which are used by the `TaskDispatcher` and its `PriorityQueue` to decide which task should execute next. We use the underpinning asynchronous messaging infrastructure to send these priorities to the task dispatcher. This avoids unnecessary communication overhead (e.g. because of polling), and the task dispatcher is only interrupted when there is an actual update.

The use of a specific algorithm to assign priorities based on SLAs depends heavily on the kind of application. Therefore, different algorithms for prioritization can be plugged in into our middleware using the strategy design pattern [69]. Developers who select (or designers of) such algorithms are offered APIs to access the information that is offered by the `MonitoringService` and by the `SLAManager`. Furthermore, they can subscribe for notifications from the `WorkflowEngine` and the `TenantStatus`.

As output, they have to provide a mapping from tenants, jobs and tasks to weights, and deliver it to the `TaskDispatcher`.

5.3.5 Deployment aspects

This subsection explains how the different components of the middleware architecture are deployed on different nodes or as separate processes in a distributed environment. We focus especially on the aspects that affect *scalability and fault tolerance*. Typically, a node consists of a virtual machine (VM) running an application server as part of a cluster or it is an instance of a Platform as a Service (PaaS). Furthermore, a monitoring infrastructure and a (dynamic) resource management system (outside of our middleware) are needed to detect failing nodes or processes and to restart them, possibly elsewhere.

One or more `Workers` can be deployed per node, depending on the complexity of the task or whether a task requires specific hardware (e.g. for encryption). Our approach does not affect the scalability of the workers, so multiple instances of the same worker can be deployed to balance the load.

The `TaskDispatcher` and the `WorkflowEngine` components are typically deployed together on a single node to limit the cost of remote communication. This setup creates a single point of failure and possibly also a performance bottleneck. However, these are lightweight processes that can restart almost instantly in case of failure. Furthermore, a new workflow engine process can be started elsewhere, because the list of available workflows is replicated. The same applies to the task dispatcher: the state of its queue can almost completely be rebuilt based on the execution log (i.e. all unfinished tasks and jobs). Only tasks that are still being executed, cannot be determined. Therefore, the task dispatcher has two options: (i) add all unfinished tasks immediately to the queue and perform some tasks twice, or (ii) wait for a limited time and collect the results of tasks that were still executing. The last option is only recommended for long-running or expensive tasks.

Our architecture offers the opportunity to replicate the workflow engine: it is a stateless component that caches a copy of the workflows in memory, while workflows are only seldomly updated. In contrast, preserving the consistency of the priority queue of the task dispatcher is crucial and therefore hinders scaling. However, separate queues can be created per task type, which enables the SaaS provider to spread the load and improve the performance of each queue.

Finally, the monitoring and prioritization components are deployed on separate nodes. In this case, active replication is not recommended: both components need all available data to perform their work, which hinders the distribution of the load. However, these components cannot become a performance bottleneck, because the task dispatcher and the workers can operate independently from these components. Furthermore, the operational results of both components (i.e. monitoring data and priorities) are persisted in the distributed datastore, allowing a backup service to immediately continue the work in case of failure.

5.4 Evaluation

The evaluation of our middleware architecture for performance isolation includes (i) the demonstration of its SLA enforcement capabilities, and (ii) the measurement of the performance overhead that the middleware introduces. We compare the results of our SLA-aware solution with a default system that implements a *FIFO* (First In, First Out) strategy without any sophistication, and with an approach that implements a straightforward, *fair* distribution of resources over the different tenants.

This section first presents the prototype implementation. Next the setup and the evaluation scenario that are used for demonstration and further evaluation are described. Subsequently, the results of this demonstration (i.e. actual tenant-specific SLA enforcement) and the performance evaluation are presented. Finally, a discussion section addresses the general applicability and the limitations of our middleware solution.

5.4.1 Prototype implementation

We have developed a Java-based prototype of our middleware architecture. JBoss AS is used as application container; RabbitMQ as pub/sub infrastructure. The different components of the middleware are implemented as RESTful web services using the JBoss RESTEasy framework. For the distributed storage we selected Apache Cassandra [112], because it was designed for high write throughput, availability (via replication) and scalability. Furthermore the consistency level is tunable: strong consistency can be achieved when necessary, but comes at the expense of availability. The communication with this database occurs in our prototype via Astyanax, a Java-based client library for Cassandra.

The order of tasks (or messages) cannot be changed in a standard queue. Therefore, the implementation of the `PriorityQueue` should offer support to query tasks based on task type, on tenant ID, and on job ID. In addition, it should ensure consistency, even in the context of distributed joins. For example the document processing application may only create a zip file when all documents of a specific batch have been generated. Existing queuing services ensure consistency, however they often offer no or only limited filtering support (e.g. based on a single parameter). Using such services would inherently require the deployment of multiple queues, for example one queue per task type. We therefore have provided our own implementation for the prototype.

For evaluation purposes, we have developed two (relatively simple) prioritization algorithms for our middleware: (i) a strategy that ensures fair distribution over all tenants without taking SLAs or monitoring data into account, and (ii) an SLA-aware strategy, that makes decisions based on the different tenant-specific SLAs and on the available monitoring data.

Furthermore, we have implemented the document processing application, as described in the case study, on top of our middleware prototype. The workflow combines six different task types, each task implemented by a different worker: the five activities that have been introduced in the example workflow (see Fig. 5.2) and a `JoinTask` that is responsible for the join operation. In this instance of the document processing application, CSV data is used as input to generate PDF documents (via XSL-FO), which are then aggregated into a zip file. The web front end of the multi-tenant SaaS application is developed with Java Servlets and Java Server Pages (JSP). This front end is used to manage the application configuration and to submit new jobs.

5.4.2 Evaluation scenario & setup

This part presents an evaluation scenario to demonstrate the operational capability and the effectiveness of our performance isolation middleware (see Table 5.1). The scenario must contain a realistic mixture of different tenant-specific SLAs (e.g. some addressing throughput, some other imposing deadlines) and tenant behaviours (average resource needs versus very high, up to excessive needs[1]). We composed this scenario in order to maximally utilize the available throughput. Such a scenario is well-suited to demonstrate and verify the ability of the middleware solution to fairly distribute the platform capacity despite different tenant behaviours, and to ensure compliance with the different tenant-specific SLAs. The different tenants that contribute to the scenario are specified in Table 5.1.

We have executed this scenario in three deployment contexts, i.e. in a configuraton of the performance isolation middleware that implements (i) FIFO, (ii) fair share, and (iii) SLA-awareness. The former only uses the task and job execution manager of our middleware and a basic FIFO queue that retrieves the first task in the queue (i.e. no monitoring and prioritization), while the other two use the full middleware but with different prioritization algorithms (as sketched above).

[1]A tenant that imposes excessive needs for resources is said to expose *aggressive behaviour*, as opposed to *normal* behaviour. In our scenario, aggressive behaviour of a tenant is represented by the submission of multiple jobs at the same time, thus Tenants 1, 7 and 8.

Table 5.1: Overview of the evaluation scenario. All jobs have the same size and contain raw input data for the generation of 800 PDF documents. Tenants with high resource needs, which corresponds to aggressive behaviour, submit multiple jobs at the same time. The default SLA specifies a minimum throughput of 1 document per second per tenant.

	Jobs	Start	SLA
Tenant 1	Jobs 1, 2, 3	at 0s	Minimum throughput: 1 document/s
Tenant 2	Job 4	at 5s	Premium throughput: 4 documents/s
Tenant 3	Job 5	at 5s	Deadline: at 2700s
Tenant 4	Job 6	at 5s	Deadline: at 1150s
Tenant 5	Job 7	at 5s	Minimum throughput: 1 document/s
Tenant 6	Job 8	at 5s	Minimum throughput: 1 document/s
Tenant 7	Jobs 9, 10, 11, 12	at 300s	Minimum throughput: 1 document/s
Tenant 8	Jobs 13, 14, 15, 16, 17	at 500s	Premium throughput: 4 document/s

The evaluation environment is set up as follows. The prototype of the middleware and the document processing application are deployed on a private PaaS platform using a JBoss AS 7.1.1 cluster and an Apache Cassandra 1.2 database cluster. The underpinning infrastructure consists of three servers, each based on two Intel Xeon 6- to 8-core processors and 64 GB of memory. This infrastructure is virtualized using OpenStack (Folsom release). We have created one instance per worker of the application and we have deployed these on separate virtual machines. Two additional virtual machines are allocated, a first one to the task dispatcher and the workflow engine, and the second one to the monitoring service and the SLA-driven prioritization services. Thus, 8 different virtual machines are used in total.

5.4.3 Demonstration of SLA enforcement

The execution of the evaluation scenario for the three different cases clearly shows the effectiveness of our SLA enforcement middleware (see Fig. 5.5). In the *FIFO case*, the job that is started as first, completely runs through the workflow first (see Fig. 5.5(a)). All jobs are completed one by one, resulting in several SLA violations. For example, the aggressive behaviour of Tenant 1 has an impact on the performance of the service that has been delivered to the other tenants, i.e. the tasks of other tenants have to wait for Tenant 1's tasks to finish. The same effect is visible when observing the impact of Tenant 7 on the execution of the jobs that are generated by Tenant 8. The lack of performance isolation thus causes the SLAs of Tenants 2 and 8 to be violated, although these tenants pay for premium throughput. In addition, the application was also not able to deliver the minimum throughput to Tenant 6.

The *fair share case* shows that our middleware is able to enforce fair distribution over the different tenants (see Fig. 5.5(b)). Although Tenant 1 submits three jobs at the same time, its total share of resource consumption is equal to the one of other tenants that have a job running. In addition, the resources available to Tenant 1 are also fairly distributed over the jobs of Tenant 1. Consequently, the jobs of Tenants 2 to 6 finish at about the same time, and the Tenants 1, 7 and 8 do not benefit from their aggressive behaviour. At the end of the scenario more resources are available because less tenants have a running job, which allows Tenants 7 and 8 to consume more resources. However, fair distribution is not sufficient to ensure SLA compliance: the application does not deliver the required throughput to comply with the SLAs of Tenants 2 and 8. In case one additional tenant would have submitted a job, then the fair share approach would also not have been sufficient to deliver the minimal throughput to Tenants 5 and 6.

Finally, in the *SLA-aware case*, our middleware ensures compliance with the different tenant-specific SLAs as defined in the scenario (see Fig. 5.5(c)). In contrast to the previous cases, Tenant 2 receives a higher share of the resources in accordance to the SLA that offers premium throughput. Consequently, this job is finished first. In addition, our middleware ensures a minimal throughput for Tenants 1, 5 and 6, while the deadlines specified in the SLAs of Tenants 3 and 4 are met. Notice the rather large differences between the results of the SLA-aware approach and the other cases: because of the premium throughput, the jobs of Tenant 8 finish sooner than the jobs of Tenant 7, and the execution of the job for Tenant 3 has been spread over a much longer period compared to the other cases (because his SLA allows it). Again, aggressive tenants cannot adversely affect the performance delivered to the other tenants.

To demonstrate that our middleware can also address jobs with very low response times, we added an extra job to the basic scenario: Job 18 by Tenant 9, containing 2 documents, starting after 25s (i.e. a period of heavy load) with a required response time of 10 seconds. Fig. 5.6 shows that our middleware effectively is able to support low response times, even when the application is under heavy load.

5.4.4 Performance overhead

We evaluated the performance overhead by measuring the global throughput during the execution of the entire evaluation scenario (see row 1 in Table 5.2). Obviously,

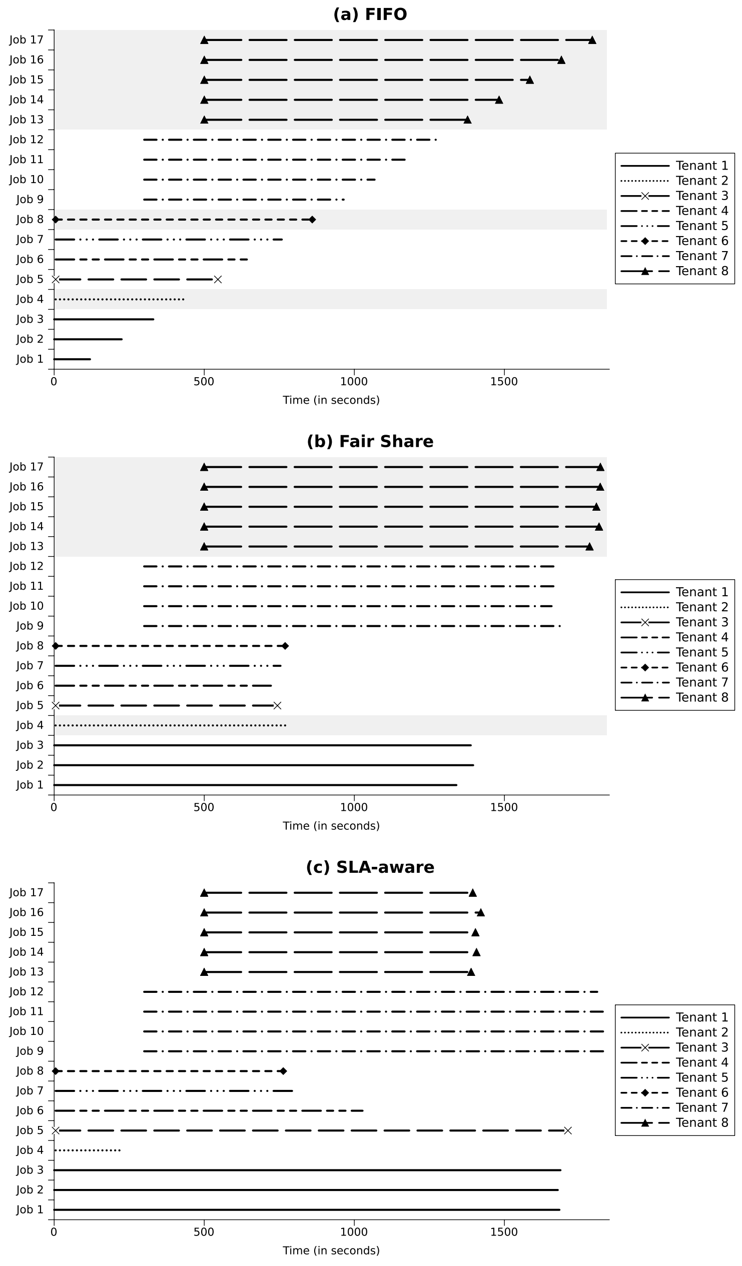


Figure 5.5: Graphs presenting the execution of the tenant-specific jobs according to the evaluation scenario, respectively applying a FIFO approach (Fig. 5.5(a)), fair distribution of resources per tenant (Fig. 5.5(b)), and SLA-driven prioritization (Fig. 5.5(c)). SLA violations are highlighted in grey.

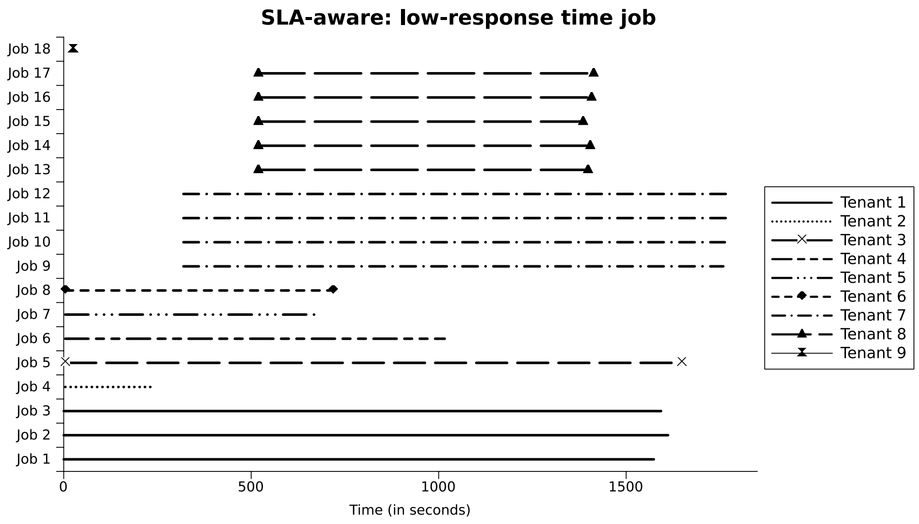


Figure 5.6: Graph presenting the execution of the evaluation scenario, including an additional job with low response time (by Tenant 9).

the FIFO version has the highest throughput, because it has no overhead caused by monitoring and prioritization and it has a basic FIFO queue. However, our performance isolation middleware introduces only a limited overhead of about 2%. Moreover, there is no difference in performance between the fair share and the SLA-aware algorithm. The main causes of the performance overhead are the selection of the next task based on priorities and the additional remote communication with the messaging system, which are similar for the fair share and the SLA-aware versions. The impact of the monitoring service is minimal.

Table 5.2: Overview of the average global throughput over 15 runs each (in terms of documents per second) by the FIFO version, and by the two versions using our middleware. The first row corresponds to the default setup for the evaluation scenario, while the second row presents the results for the upscaled version with two workers for document rendering.

	FIFO	**Fair share**	**SLA-aware**
Default	7.56 (σ 0.08)	7.40 (σ 0.15)	7.41 (σ 0.09)
Upscaled	19.56 (σ 0.05)	19.37 (σ 0.15)	19.46 (σ 0.13)

These results also confirm that the evaluation scenario completely consumes the available throughput: the SLAs of Tenants 1, 2, 5 and 6 already require a throughput of at least 7 documents per second. Combined with the jobs of Tenants 3 and 4, the scenario reaches the average of 7.5 documents per second.

The bottleneck in the evaluation setup is the worker for rendering PDF documents using XSL-FO. Therefore, we deployed an additional worker for this task. As a result, the SLAs were easily delivered (see Fig. 5.7), but the average throughput of this upscaled version more than doubled (see row 2 in Table 5.2), which is theoretically impossible. However, we consequently came to these results, also when alternating with the default evaluation setup and the different cases. This effect is caused by the underpinning cloud platform, probably because multiple task workers are deployed on the same physical machine and influence each other. Furthermore, these results indicate that the relative performance overhead of our middleware decreases with an increasing throughput. This means that it is a rather static cost for our scenario, independent from the throughput.

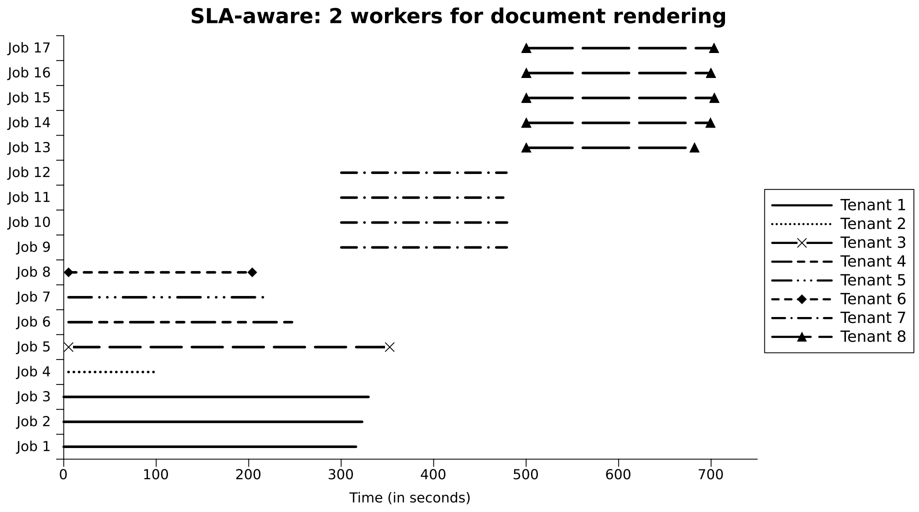


Figure 5.7: Graph presenting the execution of the upscaled version of the evaluation scenario, i.e. with two workers deployed for document rendering.

Finally, to evaluate the scalability of the task dispatcher and the priority queue, we deployed 22 simple workers running sleep 0 on 11 virtual machines (one worker per CPU). In this configuration, the rate of tasks processed by the middleware reached up to 300 tasks per second in average (over 100 runs with each 25000 concurrent tasks submitted). Furthermore, it receives updates from the prioritization service and polling requests from idle workers. The task dispatcher can easily keep up with this workload. Moreover, the throughput can be further increased by bundling tasks.

5.4.5 Discussion

In this section, we further analyse the results of the evaluation, and discuss the limitations of our approach and the lessons learnt.

Analysis of results. The evaluation shows *promising results* with respect to the architecture of our middleware solution: it enables SaaS providers to enforce different tenant-specific SLAs, while only a limited performance overhead is introduced. As explained above, this performance overhead is mainly caused (i) by the task selection in the `PriorityQueue`, and (ii) by the additional remote communication with the messaging system. The performance overhead introduced by the latter depends on the frequency of changes in the workload that is presented to the SaaS application, i.e. when the number of tenants changes, when new jobs arrive, etc. Few changes and longer-running tasks will cause lower overhead, but often do not require rapid response to changing requirements to ensure performance isolation. Therefore, we do believe that our evaluation scenario is a representative sample to demonstrate the feasibility of our solution: 17 jobs spawning many tasks of a small size, and this under high workload and with different tenant-specific SLAs that must be respected within a single, limited time frame.

It remains possible however, that the performance isolation middleware can be further optimized by limiting the frequency of its interventions: in other words, should workload changes occur at a very high frequency, then it may be better to schedule the middleware activities periodically. This is a topic of further research.

The results also show that the implementation of our SLA-aware prioritization algorithm is conservative. For example, the jobs with a deadline are finished too soon, which prevents other jobs from finishing earlier. However, the design of a prioritization algorithm is not the focus of this work. In the state of the art, many algorithms are available that can be plugged into our middleware to improve the prioritization process, e.g. [107, 142, 218].

General applicability. In this paper, we focused on service-oriented multi-tenant SaaS applications that execute a workflow consisting of different subservices (workers). More specifically, we aimed at SaaS applications that process a mixture of small and medium-sized batches with different performance SLAs in terms of latency, throughput and deadlines. Document processing is a typical example of such an application, as illustrated in Section 5.2.2. Such applications require a rapid response from the performance isolation middleware to ensure the different SLAs, even under changing circumstances.

However, our architecture can also be used in a simplified form for interactive SaaS applications, as shown in previous work [213]. In this case, there is no need for a job manager and a workflow engine. Requests are directly sent to the task dispatcher. Obviously, the prioritization component should be adapted to this type of applications, for example with a different prioritization algorithm and update frequency. Evidently, for (very) long-running workflow processes, e.g. e-science applications, the more typical approaches of resource provisioning and workload planning offer the most suitable solution.

Another limitation of the current architecture is that the performance isolation is limited to the actual execution of the tasks. However, workers can depend on other services provided by the cloud platform or third parties, for example the distributed storage service. Although there exist solutions for performance isolation in multi-tenant storage (e.g. [193]), it is definitely worth to analyse the impact of this limitation, in particular when the application depends on services that cannot be controlled by the SaaS provider, for example as part of a public PaaS offering. This is an avenue of future work.

Scalability. In this work, we assume that the SaaS provider predefines a set of SLAs that he is able to support and that tenants can select when consuming the service. Custom SLAs are possible, but only after negotiation with the SaaS provider and typically at a high price. This approach enables the SaaS provider to stay in control and to ensure that sufficient resources are available. By applying application-level multi-tenancy, these resources can be maximally shared among the different tenants, resulting in a reduction of the operational costs.

However, the SaaS application should be able to scale with an increasing number of tenants in order to deliver the different SLAs. Our middleware aims to enforce the tenant-specific SLAs, while maximally utilizing the available resources, but it does not automatically provision additional resources when necessary. In case of insufficient resources, SLAs will be violated. As a consequence, the functioning of the middleware will be limited to minimizing the penalties for the SaaS provider. Therefore, it is crucial to also deploy an (external) resource management system and monitoring infrastructure to dynamically add and remove resources when necessary. Optionally, these systems can integrate with our middleware to better estimate the needs, for example by subscribing for the different notifications that are generated during execution.

Furthermore, as we explained above, we experienced that a resource management system not necessarily provides the optimal deployment of the different components and workers in a virtualized environment, and thus can have an impact on the scalability of the application.

5.5 Related work

We discuss related work by addressing three important subdomains: (i) performance isolation in multi-tenant SaaS applications, (ii) resource provisioning, scheduling and cloud auto-scaling, and (iii) QoS-aware service composition.

Performance isolation in multi-tenant SaaS. Typically, performance isolation in SaaS applications is achieved by assigning separate virtual machines to the different tenants [85, 197]. However, this approach restricts resource sharing (only hardware is shared) and has a higher application management and maintenance cost [44, 84, 214].

SPIN [117] uses an anomaly detection model to predict an instable status in the execution environment of a multi-tenant application. When an anomaly report is triggered, aggressive tenants are identified based on their resource consumption. The focus of this work is the algorithm for the identification of aggressive tenants to ensure fair distribution of resources over the tenants. Tenant-specific SLAs are not taken into account. Our work, however, focuses on the development of a middleware architecture to enforce tenant-specific SLAs in distributed multi-tenant SaaS applications. Different algorithms can be plugged in, for example to improve the detection of aggressive tenants.

In previous work [213], we identified and illustrated the challenges of performance isolation in the context of interactive, multi-tenant web applications, and proposes a middleware architecture to enforce fair distribution of resources. In contrast, our work focuses on multi-tenant SaaS applications with a service-oriented architecture and a task-driven application model. Moreover, we take tenant-specific SLAs into account.

Pisces [193] provides performance isolation and fairness in multi-tenant cloud storage. An important difference with multi-tenant SaaS applications is that requests are scheduled driven by the partitioning and replication of the data across the different nodes. In addition, the approach does not take tenant-specific SLAs into account. Related to this work, [39] presents a QoS architecture to manage performance in NoSQL distributed storage systems. While our work focuses on application-level multi-tenancy, the integration with such approaches for multi-tenant storage seems appealing in light of ensuring system-wide performance isolation.

CPI^2 [225] is a system to detect and handle CPU performance isolation faults in shared compute clusters. Concretely, it tackles the problem of performance interference between different applications that share resources (i.e. CPU cycles). Our work focuses on multi-tenant SaaS applications, thus on sharing the same application among multiple customer organizations. Furthermore, our middleware enforces performance isolation at the application level, while CPI^2 works at the CPU level.

However, CPI^2 is still relevant as the different subservices of a distributed multi-tenant SaaS application can share resources (deployment on the same physical machine) and thus interfere.

Resource provisioning, scheduling & cloud auto-scaling. A considerable amount of research has been done with respect to dynamic resource provisioning in virtualized environments, workload scheduling with deadline and cost constraints, and cloud auto-scaling [27, 28, 33, 36, 110, 115, 126, 160, 189, 218]. Typically, these approaches aim to meet job-specific deadlines (cf. Scenario #2 in Section 5.2.2). The majority of these approaches consider a public or hybrid cloud environment and try to handle increasing workload by acquiring resources in the (unlimited) cloud, while minimizing the costs. Often this auto-scaling is driven by load prediction mechanisms. Other approaches focus on environments with limited capacity and schedule the workload after a reservation of capacity, which is not suitable to handle jobs that require low response times (cf. Scenario #1) and not applicable in the context of application-level multi-tenancy.

Our work is complementary to these approaches as our performance isolation middleware focuses on cloud environments that are fully shared by the different tenants (application-level multi-tenancy), while taking into account different tenant-specific SLAs, characterized by response time, throughput or deadlines. Our middleware cannot acquire additional resources itself, but it does support the scaling of the SaaS application and its workers driven by an external auto-scaling or resource management system. Thus, the integration of these existing approaches for auto-scaling and task scheduling with our work offers SaaS providers the mechanisms to enforce different tenant-specific SLAs (for the long and the short term), while being able to scale in a cost-efficient way.

QoS-aware service composition. In the domain of service composition, automated composition of Web services is often based upon QoS properties of the different services [122, 220, 223]. Non-functional properties such as security, reliability, and performance are used by a matchmaking algorithm to select the most appropriate (third-party) service. In contrast, we consider a multi-tenant context where all services are shared by the different tenants. Instead of dynamically adapting the service composition by selecting different services to ensure SLA compliance, SLAs have to be enforced on the (limited set of) subservices of the multi-tenant SaaS application.

Furthermore, several approaches exist for dynamically predicting SLA violations in service-oriented applications, for example [116, 119], which is complementary to our tenant status service. However, notice that such approaches are not tenant-aware.

5.6 Conclusion

This paper presents the architecture and evaluation of a performance isolation middleware that enables SaaS providers to manage and enforce co-existing SLAs of the different tenants in service-oriented multi-tenant SaaS applications. Driven by the monitoring data from the different application services and the different SLAs, the middleware embeds the mechanism to dynamically prioritize tasks, and to enforce timely execution while servicing multiple tenants. Our solution does not promote any particular algorithm, it is a framework that enables plugging in the necessary algorithms.

We have implemented a prototype on top of a private cloud platform, built using OpenStack, a JBoss AS cluster and an Apache Cassandra database cluster. We have evaluated the effectiveness of our solution by means of a case study, showing promising results in ensuring compliance with the different tenant-specific SLAs. Moreover, the prototype shows that our middleware introduces only a limited and acceptable performance overhead, and thus preserves the cost-efficiency benefits of application-level multi-tenancy.

Chapter 6

Efficient customization of multi-tenant Software-as-a-Service applications with service lines *

Preamble

In the previous three chapters, we have presented a middleware framework to enable and realize the tenant-specific customization of SaaS applications. However, the SaaS model as well as leveraging this advanced middleware for hosting a customizable SaaS application have an impact on the applied software engineering method. Therefore, this chapter focuses on the necessary engineering support for SaaS architects and developers. Because of the customization needs, we have proposed a method similar to SPLE, called service line engineering (SLE), for the efficient development, operation and management of customizable SaaS applications. More specifically, it addresses the management complexity of co-existing tenant-specific configurations and reduces the effort to provision tenants significantly. Section 6.3 introduces the SLE concepts and method, while Section 6.4 briefly describes the validation of this method in the context of the document processing SaaS application (cf. Section 1.3). [216] provides more details on the practical application of this method. Section 6.5 evaluates the efficiency benefits in the form of a comparative cost analysis of the required effort, and discusses the strengths and limitations of the presented method.

*The content of this chapter has been published in the 91th volume of the Journal of Systems and Software [217].

Efficient customization of multi-tenant Software-as-a-Service applications with service lines

Abstract

Application-level multi-tenancy is an architectural approach for Software-as-a-Service (SaaS) applications which enables high operational cost efficiency by sharing one application instance among multiple customer organizations (the so-called tenants). However, the focus on increased resource sharing typically results in a one-size-fits-all approach. In principle, the shared application instance satisfies only the requirements common to all tenants, without supporting potentially different and varying requirements of these tenants. As a consequence, multi-tenant SaaS applications are inherently limited in terms of flexibility and variability.

This paper presents an integrated service engineering method, called *service line engineering*, that supports co-existing tenant-specific configurations and that facilitates the development and management of customizable, multi-tenant SaaS applications, without compromising scalability. Specifically, the method spans the design, implementation, configuration, composition, operations and maintenance of a SaaS application that bundles all variations that are based on a common core.

We validate this work by illustrating the benefits of our method in the development of a real-world SaaS offering for document processing. We explicitly show that the effort to configure and compose an application variant for each individual tenant is significantly reduced, though at the expense of a higher initial development effort.

6.1 Introduction

Software as a Service (SaaS) has become a common software delivery model. It is a form of cloud computing that involves offering software services in an online and on-demand fashion (with the Internet as the main delivery mechanism). One of the key enablers in cloud computing to achieve economies of scale is *multi-tenancy* [44, 84]: the sharing of resources among a large group of customer organizations, called *tenants*. This architectural concept can be applied at various levels of the software stack: at the infrastructure level (i.e. virtualization), at the OS and middleware level, and even at the application level. Each approach makes a different trade-off between (i) maximizing scalability and operational cost benefits (including hardware and software resource usage as well as maintenance effort), and (ii) maximizing flexibility to meet the potentially different and varying tenant-specific requirements [214].

This paper focuses on application-level multi-tenancy. Application-level multi-tenancy achieves the highest degree of resource sharing between tenants [44, 214]. End users from different tenants are simultaneously served by a single application instance on top of shared infrastructure. However, when compared to infrastructure-level and middleware-level multi-tenancy, the inherent limitations in variability form a crucial disadvantage. The high degree of resource sharing typically results in a one-size-fits-all approach: the multi-tenant application only satisfies the requirements that are common to all tenants. Support for different and varying requirements of the different tenants is lacking.

To shorten the time-to-market, the initial development and release cycles of a SaaS application are typically focused on the needs of the first customer organizations (tenants). As the SaaS offering becomes more successful, an increasing amount of variations (ranging from minimal to substantial) is implemented to service new tenants and an increasing amount of tenant-specific configurations therefore have to co-exist at run time. This easily leads to an explosion of relatively small variations in the implementation as well as in the different configurations. This real-world scenario is experienced in many specific business cases; we have identified a lack in *methodical support for the development and customization of multi-tenant applications*. This lack of support can be characterized by two essential challenges:

First, SaaS providers need to be able to *manage and reuse the different configurations and software variations in an efficient way*, without compromising scalability; e.g. by avoiding additional overhead when provisioning new tenants.

Secondly, part of realizing the scalability benefits of SaaS is achieved by *self-service*: shifting some of the configuration efforts to the tenant side, e.g. by allowing the tenant to manage his tenant-specific requirements and by automating the run-time configuration process. Therefore, tenants require *additional support to manage the configuration in a tenant-driven customization approach*.

In the state of the art, some work has been performed to combine the benefits of software product line engineering (SPLE) [48, 168] with those of multi-tenancy to facilitate the customization of SaaS applications tailored to the tenant-specific needs [138, 141, 186, 214]. We define a *service line* as a specific concept that leverages on the notion of software product lines by offering a shared application instance that still is dynamically customizable to different tenant-specific requirements. However, none of the current service engineering approaches offer a complete customization process for multi-tenant applications. Moreover, customization is often limited to specific technologies or to specific types of applications.

Our solution is a feature-oriented method that is highly integrated, in the sense that the feature-level variability that is introduced in the early development stages is consistently and explicitly supported in each of the subsequent development stages, also in the run-time environment. This is a key difference with respect to traditional SPLE. Instead of delivering a dedicated, separate application product for each tenant (cf. the application engineering phase in SPLE), the entire service line (including all variations) is instantiated and deployed only once and simultaneously shared by all tenants. Specific software variants are activated at run time within one single SaaS application instance.

The main contribution of this paper is an integrated service line engineering method that focuses on addressing variability up front without compromising the scalability of SaaS applications. This method starts with the initial development phases (requirements engineering, architectural design and implementation) of the service line, but also focuses on the deployment, run-time configuration and composition, the operations and maintenance. The method is generic in the sense that each stage is open for existing work in the state of the art to be leveraged upon, yet it imposes some specific constraints (e.g. composability and traceability of features) and requires some enablers (e.g. multi-tenancy, feature-level versioning, tenant-level configuration interfaces) for service line variability.

We have validated this method in the development of an industry-level SaaS application in the domain of online B2B document processing. To this end, we closely collaborated with an industrial SaaS provider in the context of a collaborative research project [53]. Our evaluation focuses on illustrating (i) the efficiency benefits with respect to addressing the management complexity of many co-existing tenant-specific configurations, and (ii) the trade-off between the *early* effort required to design and implement the initial service line, and the *late* effort required to configure and compose application variants, to provision new tenants as well as to update and maintain the service line as a whole.

The structure of this paper is as follows. Section 6.2 motivates this work and elaborates on the problem statement. Section 6.3 articulates the concept of a service line and describes the service line engineering method. In Section 6.4, this method is applied on a SaaS application in the domain of document processing. We evaluate and discuss our work in terms of the efficiency benefits associated with service lines in this specific SaaS application in Section 6.5. In Section 6.6, related work is discussed and Section 6.7 concludes the paper.

6.2 Problem elaboration

The motivation for this paper is based on our extensive insight into the current state of practice of a number of industrial SaaS providers (which was obtained in the context of several applied research projects, e.g. [53]). In this section, we first present our characterization of this current state of practice of developing, operating, and maintaining multi-tenant SaaS applications. Then, we list the main challenges that are addressed in this paper.

6.2.1 State of the practice

We present a number of development and management activities that occur in the lifecycle of a multi-tenant SaaS application. Specifically, we focus on the customization and management challenges that SaaS providers face to efficiently offer and manage their offerings.

The following stakeholders are involved in these scenarios. The *SaaS architect*, *SaaS developer* and *SaaS operator* are employees of the SaaS provider. In addition, each tenant should assign a *tenant administrator* role to the person responsible for managing the SaaS application on behalf of the tenant organization (e.g. for user and configuration management). In theory, this role can be assigned to an internal user of the tenant or to the SaaS provider, or even to value-added resellers who are a business channel between tenants and the SaaS provider. In practice, the tenant administrator is often an employee of the SaaS provider, as detailed technical knowledge of the SaaS application is required.

Scenario #1: Initial development of the SaaS application. This scenario entails the initial design and development of a multi-tenant SaaS application by the SaaS architect and the SaaS developers. The SaaS operator is responsible for deploying and managing the (running) implementation of this SaaS application.

To shorten the initial time-to-market, the first development cycles of a SaaS application are typically focused strongly on the needs of the first customer organizations (tenants). As a result, the SaaS application typically supports limited variability beyond the scope of the initial tenants.

Scenario #2: Provision a new tenant. In this scenario, a new tenant wants to use the SaaS application and customize it to its requirements. We assume that the SaaS application already covers the requirements of the new tenant, e.g. because these requirements are very similar to those of already provisioned tenants. (Scenario #4 covers the case where new requirements have to be supported.)

Based on his (technical) knowledge about the application, the tenant administrator has to manually translate the tenant's requirements into a software configuration for the multi-tenant SaaS application. Obviously when certain requirements of the new tenant are similar to those of other tenants, parts of the existing configuration can be reused. In practice, this is done in a weakly controlled and error-prone manner (e.g. by copy-pasting configuration fragments). As a consequence, validation and testing of the newly-created configuration is of high importance. With an increasing number of tenants, this manual approach easily leads to an explosion of relatively small variations in the tenant-specific configurations.

Scenario #3: Update tenant-specific configuration. The requirements of a tenant can change over time. To satisfy these varying requirements, the tenant-specific configurations have to be updated.

Every time the requirements of a tenant change, the tenant administrator has to update the configuration to fit the new requirements and to ensure correct behaviour. Again, this is an error-prone manual task, and it might lead to divergence of configuration fragments that were initially equal (e.g. by copy-pasting), resulting into even less reuse and manageability of configurations.

Scenario #4: Support new requirements. In this scenario, some requirements of existing or new tenants are not yet covered by the current SaaS application. Therefore, the SaaS architect and developers have to add support for new requirements. This implies that new variations are introduced into the SaaS application.

Adding support for new requirements can have an impact on the existing workflows, services, components and interactions of the SaaS application. Such changes will ripple to the tenant-specific configurations, even to the configurations of tenants whose requirements did not change. The SaaS provider has no view on configurations that are affected, and the SaaS operator will have to verify the configurations of all tenants, while the tenant administrator will have to double-check specific configurations. This manual task obviously does not scale well with the large amount of tenants.

Scenario #5: Update and maintain SaaS application. The SaaS operator is also responsible for the maintenance of the SaaS application as well as the underpinning platform, and for keeping these up-to-date. For example, the current implementation might have to be upgraded to a new version of the platform or libraries, bugs have to be fixed, or performance improvements have to be introduced. It is a key advantage of SaaS applications that these updates can be performed more frequently than in an on-premise setting, i.e. maintenance happens continuously. On the one hand, such changes can be limited to the internals of a single software component, but on the other hand they can also affect the complete architecture of the SaaS application and even conflict with the requirements of some tenants. Similarly to the previous scenario, the SaaS operator and tenant administrator will have to verify the configurations and correct them if needed.

In the case that the updates conflict with the current preferences of some tenants, the tenant administrator cannot solve the issue by updating the configuration. A solution for this issue is to support both the older and the latest version of the changed software artifacts simultaneously, possibly for a limited time only (transition period). However, this implies that the SaaS operator should maintain different versions, and this easily leads to a rapidly growing number of relatively small variations in the implementation as well as in the different tenant configurations.

6.2.2 Challenges

Based on the above characterization of the current state of practice, and based on the state of the art (which is presented in Section 6.6), we have identified two key challenges to support tenant-specific customization in multi-tenant SaaS applications:

1. *Efficient development, management and reuse of software variations in a SaaS offering.* As illustrated above, variability in multi-tenant SaaS applications may have fundamentally different sources (tenant's requirements, evolution, maintenance, etc.), but is inherently present. Currently, software variations and tenant-specific requirements are introduced and managed in an ad-hoc, often manual and error-prone fashion, and we have shown above that this has some obvious disadvantages. For example, whenever a tenant's requirement changes, the development process has to be re-executed for this tenant, and the ensuing changes might cause ripple effects that affect potentially all tenants. Especially when the SaaS offering evolves and the amount of supported variations grows, the cost of variability (expressed in terms of development, maintenance, and configuration effort) grows accordingly.

   Because of the high impact of variability on the SaaS application, it cannot be realized as an afterthought in the development process. Essential qualities such as maintainability and evolvability of the SaaS application, as well as the reusability and modularity of specific software variants must be taken into account throughout the entire development life-cycle of the SaaS application (in accordance with design principles such as attribute-driven design [22]).

2. *Efficient, tenant-driven customization of the multi-tenant SaaS application.* In the current state of practice, the tenant administrator has to analyse the tenant's requirements, create technical configurations, validate, test and activate these configurations manually in the SaaS application. One of the main advantages of the SaaS paradigm however is related to the fact that some of the configuration complexity can be outsourced to the tenant himself (typically called *self-service* [201]). This is essential to allow the SaaS provider to benefit from cloud scalability benefits; i.e. by reducing the time to provision new tenants.

   Specifically, dedicated configuration interfaces must be offered to the tenant that (i) are sufficiently expressive to allow him to create valid configurations of the SaaS application that meet his requirements (i.e. expressed in terms of the variations supported by the SaaS provider), but (ii) that hide the technical inner workings of the SaaS application from the tenant (i.e. provide abstraction). In addition, some of the activities of the tenant customization process (generation of technical configurations, validation, run-time activation of configurations) have to be automated to achieve this level of self-service.

In this paper, we address these challenges by introducing an integrated service line engineering method which is presented in the next section.

6.3 Service line engineering: Concepts & method

This section presents our *service line engineering (SLE) method* that supports tenant-level variability in multi-tenant SaaS applications without compromising the essential benefits of scale associated with cloud computing. We define a *service line* as a SaaS application that is built as a software product line consisting of customizable services that can be dynamically selected and configured based on the tenant-specific requirements, with the major difference that *one single instance is supporting the different application variants.*

The SLE method is feature-oriented and highly integrated, in the sense that the feature-level variability that is introduced in the early development stages is consistently and explicitly supported in each of the subsequent development stages, also in the run-time environment. Fig. 6.1 presents the individual development and management activities and their input and output artifacts of relevance to the presented method.

The initial investment is in Service Line Development and Deployment, which consists of the following activities:

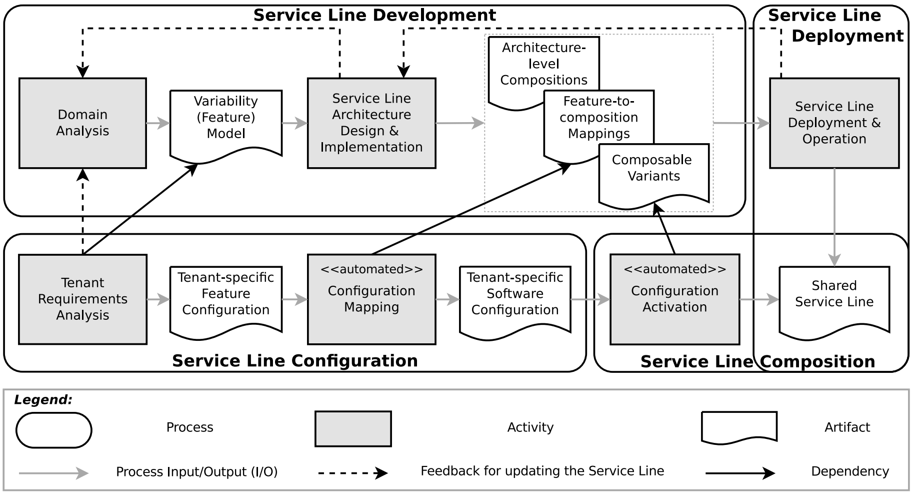

Service Line Development
Service Line Deployment
Domain Analysis
Variability (Feature) Model
Service Line Architecture Design & Implementation
Architecture-level Compositions
Feature-to-composition Mappings
Composable Variants
Service Line Deployment & Operation
Tenant Requirements Analysis
Tenant-specific Feature Configuration
<<automated>> Configuration Mapping
Tenant-specific Software Configuration
<<automated>> Configuration Activation
Shared Service Line
Service Line Configuration
Service Line Composition
Legend:
Process
Activity
Artifact
Process Input/Output (I/O)
Feedback for updating the Service Line
Dependency

Figure 6.1: Service Line Engineering Method.

1. **Domain Analysis** is quite similar to SPLE, yet it introduces a feature-based variability model that *includes SLAs* and *supports versioning* of the feature model itself, both of which are key requirements for service lines.
2. During **Service Line Architecture Design & Implementation**, variability is made explicit at the level of the service line architecture by creating separate and explicit *variability-supporting views*. Abstraction is made of (i) the level of granularity of the service line compositions supported in the architecture and (ii) the underpinning SaaS middleware and dynamic composition technologies that will be used. Additionally, the traceability and composability of features and their corresponding software variants are taken into account as key architectural requirements for the service line. The implementation support is based on *feature-to-software-composition mapping specifications* that describe the mapping between feature models and the selected features, and specific components in the full architecture of the service line. These mappings are the basis for automation.
3. **Service Line Deployment** involves the instantiation of the service line on a powerful platform, so that the impact of technical developments to complement and extend PaaS, IaaS and middleware technology can be limited. We present some *key requirements and criteria* for these types of platforms, without going into the details of comparing a snapshot of some popular, contemporary technologies. After initial service line development and deployment, the focus shifts towards **Service Line Operation**. To avoid service disruption while conducting essential evolution and maintenance activities, the service line and the underpinning middleware should support non-trivial dynamic adaptations far beyond the classical upscaling

or downscaling of resources typically cited in a SaaS context, e.g. updating the underpinning platform or supporting new features.

Subsequently, Service Line Configuration and Composition become relatively straightforward by leveraging on the investment of service line development and deployment.

1. **Tenant Requirements Analysis** leverages on the feature model (defined during domain analysis): a feature-oriented configuration interface is offered to the tenants, and verification of the selected set of features is done by validating the constraints of the feature model.
2. **Configuration Mapping** generates an application variant, in fact a tenant-specific configuration that extends the service line implementation, on the basis of the mapping specifications that have been defined in the service line architecture.
3. Finally, **Configuration Activation** occurs at run time. When requests arrive that can be attributed to a specific tenant, the running service line is dynamically adapted to match the tenant-specific configuration. This becomes feasible because of (i) the composability and traceability of variants, (ii) the modularity of tenant-specific configurations that co-exist in the service line, and (iii) the capabilities of the underpinning platform.

As with many software engineering approaches that deal with evolving requirements [64, 154], it is an essential property of the presented SLE method that it is iterative. This is represented by the multiple feedback loops in Fig. 6.1. Many of these feedback loops are driven by the scenarios presented in Section 6.2.1, for example new requirements that are discovered during the tenant requirements analysis often lead to a re-iteration over the previous activities. These loops and the activities they pertain to, are discussed below.

6.3.1 Domain analysis

Next to regular requirements elicitation and analysis techniques (obtaining functional and non-functional requirements), engineering a service line starts with the activity of domain analysis to obtain the essential characteristics of SaaS applications in a particular domain (e.g. document processing). Variability analysis is a part of domain analysis that focuses on eliciting the typical commonalities and variabilities within the service line. We focus on tenant-level variability in this activity, i.e. differences in tenant requirements (sometimes also called *external variability* [168]). A key output of this activity is the *variability model* that represents the domain-specific commonalities and variabilities in a service line.

In this paper (and in alignment with traditional SPLE), we apply a feature-based approach [99, 100] to model variability, and therefore the relevant output variability models we discuss in this paper are feature models. The state of the art covers several approaches for feature modeling and feature models can be represented in different ways, from text-based (e.g. FDL [60] and TVL [47]) to graphical in the form of feature diagrams (e.g. FeatureIDE [101] and S.P.L.O.T. [131]). Furthermore, CVL [88] is a language for variability modeling that is submitted for standardization.

This activity does not differ substantially from domain analysis in traditional SPLE approaches. To build the initial feature model, requirements are gathered from (internal) domain experts, as well as representative tenants. Often these tenants can be divided into groups with similar requirements. The resulting features are specified in terms of domain-specific functionality and business rules at the abstraction level of the tenants.

However, two essential differences (relative to SPLE) must be highlighted that can be attributed to the multi-tenant SaaS context in which the feature models will be embedded:

1. The context of multi-tenancy and SaaS introduces *additional variability, especially driven by non-functional requirements*. Because there is a single deployment of the service line that is shared by all tenants and fully managed by the SaaS provider, tenants want to impose additional availability and performance requirements to gain more control over the delivered service. Furthermore, some tenants may require strict isolation of their data for security reasons, for instance in a separate database (or even a separate execution environment). This additional variability, which does not exist in traditional deployments, also needs to be explicitly represented in the feature model, more specifically in the form of service level agreements (SLAs).
2. Also, as an operational service line typically services many tenants simultaneously, actual down-time should be minimized. In the context of evolution, instead of undeploying (taking off-line) the service line or its features, a more preferred tactic is to deploy the updated features and reconfigure individual tenants to make use of these updated features. An essential enabler for this is *versioning support* in the feature model.

6.3.2 Service line architecture design & implementation

The next activities in the service line development process are the design of the service line architecture by the SaaS architect and the actual implementation of this architecture, including all variations, by the SaaS developers. During these activities, a number of important aspects have to be considered by the SaaS architect: (i) the use of architectural formalisms that support variability and variability management, and (ii) composability and traceability of individual features and their corresponding software variants in the service line architecture. These aspects *constrain* the architect in the following ways.

Architectural support for variability

The design of the service line architecture results in the typical architectural views (cf. [108]), showing the building blocks of the service line (structural view), the deployment of the service line (deployment view), processes and component behaviours (process view), etc. The service line architecture describes all possible variations in the multi-tenant SaaS application.

Specific variations are seldomly contained within one architectural view, especially when it comes to non-functional features (e.g. performance SLAs). For example, to realize performance-level variability, the architect has to introduce performance monitoring components in the structural view, model different resource allocations (e.g. nodes, CPU percentages, physical connections, etc.) in the deployment view, and introduce additional behaviour in the process view, e.g. to document the protocol to dynamically upscale. It is clear that when the amount of features increases drastically, documenting these variations in the complete architecture, in and across these architectural views will lead to complex, composed models in which it is hard to distinguish the base architecture from specific, individual variations.

Therefore, it is an essential decision in the creation of our method to express service line variations in separate *variability-supporting views.* These views enable the SaaS architect to manage and design different variations and versions in the service line separately. To realize this, the *variation points* in the service line as well as the *binding* between these and actual *variants* (i.e. alternative implementation artifacts) are therefore made explicit in the architectural description.

The current state of the art presents several (meta-models for) variability views that can be used in our method by the SaaS architect, for example [68] and [82]. However, these solutions lack support for a multi-tenant SaaS context, i.e. they do not offer support for (i) co-existing configurations that are activated on a per-tenant basis, and (ii) versioning to enable service line evolution.

Composability and traceability of features

At the level of the architecture, a feature is represented by a set of *(architectural) components.* Depending on the selection of middleware technology, such a component can be implemented by different implementation-level artifacts, such as classes, software components, web services and workflows, etc. It is an important concern during the development of service lines to ensure the *composability* of features and their corresponding software variants; i.e. to make sure that separately developed feature implementations remain compatible (syntactically and semantically), so that they can later be composed to a working system. To support this concern, the base architecture has to be built around stable interfaces [208], and components should be self-contained (modular) and preferably be stateless. The latter is especially relevant in a multi-tenant context because it simplifies dynamic customization (no need for quiescence) and thus limits the performance overhead. Modularity makes components interchangeable and enables SaaS developers to reuse components in different compositions and thus in the implementation of different features.

Furthermore, the SaaS provider should be able to associate the different variants in the service line architecture with the features in the feature model resulting from the domain analysis, but also with particular software artifacts in the implementation, and even at run time with the tenant-specific configurations and the dynamic compositions in the deployed service line. Maintaining these *traceability* links is thus crucial to combine the different activities of our method in an integrated process. Furthermore, rigorously documenting the traceability links between different artifacts will enable the creation of advanced management and development tools, for example to signal inconsistencies to the developer (e.g. when certain features exist in the feature model, but no implementation is available). In general, these concrete associations between features, architectural components and software artifacts enable the SaaS provider (i) to ensure consistency across the service line and to analyse the impact of changes (cf. Scenarios #1, #4, and #5), and (ii) to support self-service by automating the configuration of the service line based on the tenant-specific requirements (cf. Scenarios #2 and #3).

To achieve traceability, the SaaS architect has to explicitly define *feature-to-software-composition mapping* specifications for each feature in the feature model. In the state of the art, several approaches exist to support variability and traceability in software architectures [82] and throughout the process of software product line engineering [18, 38]. However, these approaches require some modifications to enable integration with our method, e.g. versioning support.

Notice that non-functional requirements do not necessarily map on particular software artifacts. For example, performance SLAs can provide input for a monitoring service, and availability SLAs can be translated to a replication strategy. Although these features depend on the presence of particular middleware services, they do not necessarily have an impact on the service line implementation itself. In contrast, security is an example of a non-functional requirement that typically maps on particular components (e.g. for encryption, authentication or authorization).

6.3.3 Service line deployment & operation

After developing the service line, the SaaS operator creates an instance of the service line and deploys it on top of the cloud infrastructure of the SaaS provider. After this deployment process, the service line is accessible by the tenants (typically via the Internet) as a customizable, multi-tenant SaaS application.

The SaaS provider has to decide which environment is most appropriate to host the service line. For example, the SaaS provider can choose to deploy the service line on top of an existing PaaS or IaaS platform, or to set up his own cloud infrastructure. The former option requires no initial investment (i.e. infrastructure acquirement) and less maintenance, but the SaaS provider has less control over the cloud platform and thus needs to take these limitations into account. In addition, the necessary middleware support should be available or installed by the operator.

This process also includes *operating* the service line: aspects related to monitoring the running service line (e.g. to verify SLA compliance, to optimize resource usage across tenants, etc.), as well as managing and maintaining the different co-existing versions of the components, for example upgrading features, disabling certain variants, etc. (cf. Scenarios #4 and #5). Specific management tools can be envisioned to operate a running service line.

This selection of a suitable SaaS middleware platform that underpins the service line has a major impact on the architectural design, the implementation, as well as the deployment and operation process. We therefore have defined a feedback loop to the development process of our method (see Fig. 6.1). The middleware platform typically provides a number of enabling services for the development (e.g. component technology, libraries, etc.), deployment, management and operation of service lines. We require at least the following elements of the SaaS middleware:

1. *Support for versioning of features and feature implementations.* Versioning functionalities enable different versions of service line artifacts (e.g. modules, libraries, workflows, etc.) to co-exist. This is essential to realize, test and manage partial upgrades in service lines that potentially service a wide range of tenants. In addition, it supports the traceability concern. Furthermore, a common practice in cloud computing to reduce the risk of failure during upgrades, is to perform rolling upgrades [61]. The SaaS middleware should offer the necessary support to enable such a gradual roll-out of upgrades.
2. *Basic support for multi-tenancy.* Multi-tenancy is a core characteristic of a service line. [84] and [25] discuss guidelines and approaches to develop a multi-tenancy enablement layer for multi-tenant SaaS applications. Such a middleware layer offers the necessary support for application-level multi-tenancy, for example to ensure data isolation between the different tenants. Each incoming request is associated

with the corresponding tenant, and this context information is processed with the request throughout the SaaS application and the middleware platform (e.g. storage service). Some existing PaaS platforms (e.g. Google App Engine) already offer built-in support for tenant data isolation.

3. *Support for dynamic (run-time) composition.* Because a service line is inherently multi-tenant and thus shared by all tenants, the appropriate variants are activated at run time. Therefore, the implementation of the service line should support run-time (re-)composition. Our method is open for existing composition mechanisms, as the service line architecture makes abstraction of the heterogeneity of (composition) technologies provided by the underpinning platform. More specifically, SaaS developers should be able to combine multiple composition mechanisms in the service line implementation (e.g. both service and component composition). In the domain of adaptive and reconfigurable middleware, several technologies exist to support dynamic composition, for example using reflection and dynamic aspect-oriented programming (AOP) [165, 169, 199, 206]. However, these dynamic composition mechanisms should also be tenant-aware. For example, [214] supports tenant-aware customization using dependency injection [66], and [138] focuses on the customization of (BPEL-based) workflows based on tenant-specific requirements.

6.3.4 Tenant requirements analysis

During the activity of tenant requirements analysis, the tenant administrator is offered a configuration interface that enables the expression of the tenant's preferences and requirements. This configuration interface is based on the feature model that was created during the domain analysis activity (cf. Section 6.3.1). By selecting the appropriate features and configuring specific feature attributes, a *tenant-specific feature configuration* is created for each tenant. Because the feature model is specified in terms of domain-specific functionality and business rules, the tenant administrator does not need any technical knowledge and can thus be an employee of the tenant (thus enabling self-service). Existing tools for feature modeling often also support feature-based configuration (e.g. [101, 131]). Basically, such a configuration contains the IDs of the selected features.

The feature configuration meta-model describes the different concepts and their relations with respect to the tenant-specific configuration of service lines in terms of features (see Fig. 6.2). For each `Service Line`, a `Tenant` can specify a `Feature Configuration`. A `Feature Configuration` is derived from a feature model (not depicted in Fig. 6.2). It contains the set of features selected by the tenant administrator and, when applicable, specifies the value for a feature attribute to parameterize it. Similar to features, feature configurations are also versioned. This allows tenants to revert to earlier configurations when desired.

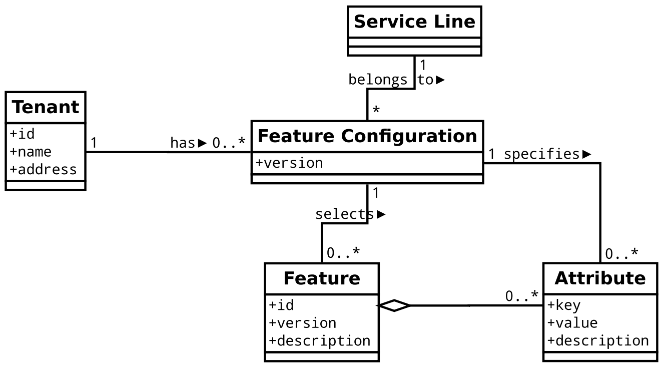


Figure 6.2: Meta-model for tenant-specific feature configurations.

Finally, the feature selection process is subject to the constraints and dependencies as defined in the feature model. After the tenant administrator has specified the feature configuration, the relations between the different selected features are verified to ensure that no conflicts exist (e.g. using Binary Decision Diagrams like [200]). Next, the feature configuration is used as input for the next activity of the service line configuration process.

When the tenant has new requirements (i.e. not yet supported by the service line (cf. Scenario #4 of Section 6.2.1)), a new development iteration of the service line might be started, potentially including updates to the feature model, the service line architecture and implementation, and finally updates to the service line deployment. Throughout the method, a number of software engineering principles have been applied to limit the potential impact of new requirements: (i) abstraction: because a feature model is hierarchical, lower-level details are typically handled near its leaves, and the coarse-grained structure of the feature model is expected to remain relatively stable, (ii) similarly, the base architecture is defined once and expected to remain stable over the life-time of the service line, while variations are described in separation of the architectural views describing the base architecture (cf. variability views), and (iii) fine-grained variations can lead to small version increments of individual components, and these can be activated at run time and on a per-tenant basis.

Evidently, the impact of new requirements will depend highly on the nature of the new requirements. In Section 6.5.2, we further discuss how new requirements can be dealt with in the context of service lines.

6.3.5 Configuration mapping

In traditional SPLE, a feature configuration leads to the instantiation of a specific product instance from the product family. Although the feature configuration is a key artifact in the process to instantiation of a single product, it is not visible in the end result (i.e. it has been compiled away during the process). In the context of multi-tenant SaaS applications however, a tenant-specific feature configuration (cf. Fig. 6.2) has to be translated to an actual *software configuration* that specifies the composition of architectural components in the service line (as expressed in the variability views). These tenant-specific software configurations co-exist in the running service line. Not only will different tenants have different software configurations, but also different configurations of a single tenant may co-exist to reflect e.g. different versions, or different choices for different end users. Maintaining these co-existing configurations introduces substantial management overhead.

The next activity in the configuration process therefore entails the *automated transformation* of a tenant-specific feature configuration to a software configuration. A software configuration defines for a particular tenant which specific variants should be bound to the different variation points. Features are thus translated into a corresponding configuration of software artifacts. Automation of this activity is possible by applying the mapping specifications as defined during the design of the service line architecture (see Section 6.3.2). This automation is essential to preserve cloud scalability and to ensure consistency.

Before activating the automatically generated software configuration, an optimization step might be required. Architectural components can have multiple variation points, which can be fulfilled by different features (i.e. by specifying bindings). Therefore, multiple occurrences of the same architectural component in a software configuration have to be eliminated by integrating and combining the different bindings. This optimization step can be incorporated as part of the automated transformation.

6.3.6 Configuration activation

While in the application engineering phase of the SPLE method a dedicated product instance is delivered and deployed for each tenant, the service line composition activity involves adapting the service line at run time to match the tenant-specific software configuration. Concretely, after the tenant administrator has specified a feature configuration, the automatically generated software configurations are immediately effective and the end users associated with that tenant can start using the service line application.

At run time, it is decided on a per-request basis which software variants should be activated, depending on the configuration of the tenant associated with the current request. As mentioned in Section 6.3.3, the service line (or the middleware on top of which it is built) should offer both explicit support for multi-tenancy and dynamic composition to realize this.

In a typical implementation strategy for tenant-aware dynamic composition (e.g. [214]), the service line middleware intercepts each incoming request and links it to the appropriate tenant (via the associated context information). Next, the corresponding software configuration is fetched. Each time a variation point is reached during the execution of a request, the appropriate variant, and thus software artifacts, are dynamically composed into the service line.

6.4 Service line engineering in practice

The service line engineering (SLE) method presented in this paper has been realized in the context of a collaborative research project with industry, called CUSTOMSS [53]. The consortium of this project comprises of three European B2B SaaS providers, active in the domains of healthcare, medical image processing and document processing. Although we applied the SLE method and its principles in all three SaaS applications, the main validation was conducted in the context of document processing in close collaboration with the SaaS provider. We introduce this application in Section 6.4.1.

We adopted a prototype-driven validation approach, in the sense that we applied our SLE method to develop a prototype (that represents a significant subset of the entire document processing SaaS application), and we leveraged the resulting prototype to validate the practical feasibility and applicability (in this section), and to evaluate the benefits of SLE (in Section 6.5).

Sections 6.4.2 to 6.4.5 present an activity-per-activity account of how we applied the SLE method to develop the prototype. For each activity, we provide the relevant artifacts, and briefly discuss the key design decisions. During development, we had to bridge some gaps by complementing our generic method with some auxiliary elements to make it effective in practice: (i) a variability meta-model that supports co-existing configurations, (ii) an approach to define reusable feature-to-software-composition mappings, and (iii) a basic set of middleware services to facilitate the development and management of service lines. Throughout this section, we highlight these auxiliary elements.

6.4.1 Document processing SaaS application

UnifiedPost[1] is a European SaaS provider that offers B2B document processing facilities to a wide range of companies in very different application domains. This multi-tenant SaaS application supports the creation and generation, the business-specific processing and the storage of millions of business documents per day, such as invoices and payslips, even up to printing and distributing. This is a fairly large-scale and complex SaaS application, that currently services around 150 different tenant organizations. In the current operation, UnifiedPost distinguishes between roughly 25 clusters of tenants, grouping companies that have reasonably similar requirements. These clusters all represent different high-level and parameterizable variations of the application.

The prototype focuses on servicing one of these tenant clusters and in the next paragraph, we present two examples of tenant companies[2] in that cluster that have different document processing requirements. For readability, we elaborate on a limited part of the UnifiedPost application, and we show only a fraction of the existing variations. Nonetheless, these are sufficiently representative examples of realistic variations within one cluster of tenants of the document processing SaaS application.

Due to space limitations, this section only presents a strongly summarized account of the entire validation effort. Please refer to [216] for a detailed account of applying our method to develop the document processing SaaS application, as well as for further details on the auxiliary solutions that we developed. The source code of the prototype, including all feature mapping specifications, and the supporting middleware services is publicly available.

Running example. *Tenant A* is a temporary employment agency that uses UnifiedPost's document processing application to process the payslips of all its employees. On a regular basis, Tenant A provides a set of payslip documents, along with some metadata, and requires the SaaS application to print the payslips and distribute them among the different employees based on the associated metadata. Since Tenant A has a large amount of employees, it has strict demands in terms of the guaranteed throughput of the document processing application, and it is prepared to pay premium fees to obtain a specific SLA in terms of throughput and deadlines.

Tenant B is active on the financial services market and uses the SaaS application for processing its invoices and distributing these to corporate customers. Tenant B only provides raw data (i.e. document data and metadata) as input, for example in a spreadsheet or in CSV format, and thus requires the generation of the invoices (based on a custom layout). The generated documents should be delivered to the

[1]http://www.unifiedpost.com/

[2]For non-disclosure reasons, we anonymized the names of the tenant companies.

customers (i.e. end users) of Tenant B, depending on their preference: via email or on paper (printed mail). In the case of email delivery, a link to the document is provided, which enables the customer, after authenticating, to view and download the document. However, if after 48 hours the document has not been retrieved, the particular document should still be printed and sent via printed mail. In addition, all generated documents should be signed and archived securely for a legally defined period (e.g. 24 months). Tenant B does not want to pay an extra charge for a guaranteed throughput level.

6.4.2 Domain analysis

Fig. 6.3 presents a subset of the feature model that is the main result of the domain analysis activity. This is a simplified feature diagram that covers the subset of the UnifiedPost application that matches the running example.

At the top level, the service line offers features related to `Processing`, `Distribution` and `Archival`. `Processing` groups the following sub-features: (i) different alternatives to acquire metadata from the raw input data (under `Metadata_Acquirement`), (ii) an optional feature for document generation based on different templates (under `Document_Generation`), (iii) an optional feature named `Signing` for signing documents using the SaaS provider's or tenant's certificate, and (iv) performance SLAs (`Throughput`) that offer different performance levels.

In terms of `Distribution`, the tenant can select from several (single) delivery channels (i.e. email or printed mail), or he can opt for a cascaded delivery. In the latter case, first an email is sent out, and if the document is not retrieved after a certain period, then the document is printed and sent on paper.

Finally, the optional `Archival` feature enables tenants to archive documents conform legislation. The logical expression at the bottom of the figure ($\neg Archival \vee Signing$) represents a dependency between features: documents can only be archived if they are signed. So, selecting the `Archival` feature implies that the `Signing` sub-feature must be selected as well.

We used FeatureIDE [101] as modeling tool to create this feature diagram, which does not support all required model elements for our method (e.g. attributes and versioning). However, these elements do exist in the underlying model specification in our prototype.

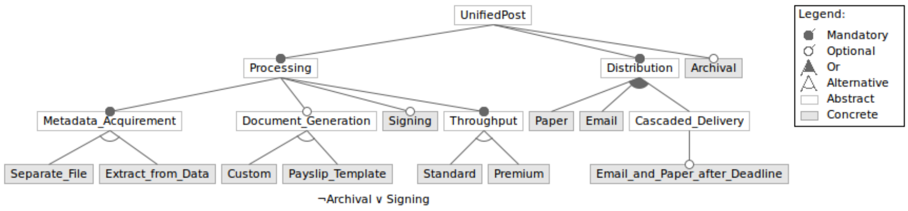


Figure 6.3: Feature diagram showing a simplified subset of the variations in the document processing case study. Specifically, only the relevant features for the two tenants presented in Section 6.4.1 are depicted. The diagram is created using FeatureIDE [101].

6.4.3 Service line architecture design & implementation

Below, we discuss how we designed and implemented the document processing service line. More specifically, we focus on illustrating how we realized variability in the architecture (modeling specific variations) on the one hand, and the composability and traceability of features in the design and implementation of the service line on the other hand.

Variability meta-model. We designed the document processing application as a customizable workflow (i.e. a service orchestration) expressed in the Business Process Modeling Notation (BPMN). The workflow is triggered when raw data is uploaded by the tenant. In the first step of this workflow, metadata is acquired from this raw input data. Subsequently, the output documents are (optionally) generated based on this metadata and according to the template selected by the tenant. Then, the generated documents are delivered to the appropriate recipients, using the selected delivery mechanism.

Variability is required in the application at two different levels of abstraction. At the workflow level, the abstract service types (e.g. delivery channel for documents) represent the variation points, and each service implementation (e.g. email delivery service) represents a specific variant. Thus, the composition type is a regular service composition. Variability is also required at a more fine-grained level, i.e. at the level of individual service implementations. For example, the document generation service offers several strategies for document generation, and in this case, we use dependency injection as composition type.

In its architectural design, the document processing workflow thus comprises a basic flow (the mandatory features) and a set of extensions, which are developed as separate, modular entities. To realize this separation, the workflow complies to the generic variability meta-model presented in Fig. 6.4. This meta-model stipulates that (i) the service line is described as a set of compositions (`Composition`), and (ii) there is no fixed level of granularity at which variability can be supported (represented by the composite structure around `Component` and `Composition` which allows the service line to be structured hierarchically). By specifying explicit `Variation Points` for a `Composition`, the architect can indicate whether that particular composition is customizable.

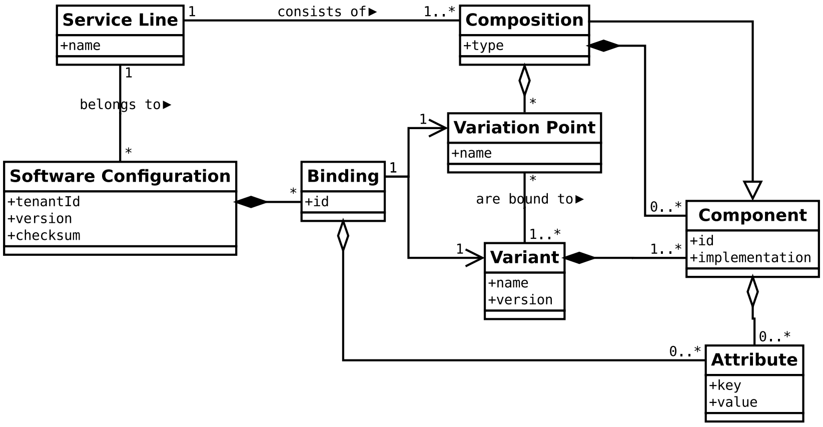


Figure 6.4: Meta-model for variability in service line architectures.

A `Service Line` is a set of compositions, some of which are customizable. The meta-model therefore allows variations to be introduced at different levels of abstraction: from the top level of coarse-grained system-level components, down to the level of fine-grained compositions. Furthermore, the meta-model makes abstraction of the specific composition mechanisms and technologies (cf. the `type` attribute) that will be used to compose these variations at run time.

This degree of openness is an essential requirement to support customization of multi-tenant SaaS applications. Variability in multi-tenant SaaS applications is typically not limited to one specific level of granularity, and SaaS middleware commonly provides a range of complementary run-time composition and adaptation mechanisms.

Furthermore, the support for co-existing configurations across the different tenants is an important requirement to enable the customization of multi-tenant SaaS applications. Therefore, the meta-model supports multiple co-existing `Software Configurations`, which define for a particular tenant (cf. the `tenantId` attribute) which specific variants should be dynamically bound to the different variation points. Such a software configuration represents the result of the automatic transformation of tenant-specific feature configurations (cf. Section 6.3.5). Since a tenant can have multiple versions of his feature configuration, software configurations also have to be versioned to enable the tenant to easily compare with an earlier version.

The openness of the proposed variability meta-model with respect to (i) the level of granularity of software variants and (ii) the technology that is used to realize the corresponding compositions, as well as (iii) the support for co-existing configurations, reaches beyond the state of the art in architecture-level variability views (cf. Section 6.6.3).

Mapping features to variants. In order to enable the automated and dynamic composition of specific variants, machine-interpretable mappings between the features and the software variants have to be defined. In addition to enforcing a hierarchical design structure, the meta-model of Fig. 6.4 allows the SaaS architect to define feature-to-software-composition mappings. These mappings bind a `Variation Point` to a specific `Variant`. To realize a `Binding`, in some cases specific attributes (`Attribute`) have to be filled in.

We have specified a mapping for each feature in the document processing application (i.e. those defined in Fig. 6.3) to components and compositions defined in the service line architecture, using a custom grammer as presented in [216]. For example, Listing 6.1 shows a feature-to-software-configuration mapping that specifies which variants to use in case the `Email_and_Paper_after_Deadline` feature (line 2) has been selected by the tenant. The workflow for the document processing application is defined as the `DocumentProcessing` composition. A relevant variation point for the running example is the delivery channel (line 5), and the `CascadedDeliveryWorkflow` (using multiple delivery channels) is the corresponding variant for this feature (line 6). This variant, however, consists of another (customizable) composition, called `CascadedDelivery`. Therefore a dependency is added to this composition via its ID (line 8). The latter composition has two variation points for the two possible delivery channels (lines 11-18), and a variation point for the algorithm that decides when to use the second delivery channel. For this feature, the timeout-based decision algorithm is selected (lines 19-23). In addition, the timeout attribute in the `Email_and_Paper_after_Deadline` feature is passed through as input for the decision algorithm via the ‘timeout’ attribute in the mapping (line 22).

The proposed approach to define reusable feature-to-software-composition mappings that can be applied to automatically (re)configure the multi-tenant SaaS application at run time is an important new element in our solution (cf. Section 6.6.3).

Listing 6.1: Example of a feature mapping for the Cascaded Delivery feature.

```
featuremapping CascadedDelivery—Email—Paper—Deadline {
    feature Email_and_Paper_after_Deadline—v1;
    composition DocumentProcessing {
        binding {
            variationPoint: DeliveryChannel;
            variant: CascadedDeliveryWorkflow—v1;
        }
        composition: CascadedDelivery;
    }
    composition CascadedDelivery {
        binding {
            variationPoint: FirstChannel;
            variant: EmailService—v1;
        }
        binding {
            variationPoint: SecondChannel;
            variant: PostalService—v1;
        }
        binding {
            variationPoint: DecisionAlgorithm;
            variant: TimeoutBasedDecisionAlgorithm—v1;
            attribute: timeout;
        }
    }
}
```

6.4.4 Service line deployment & operation

Our prototype implementation is a layered solution. In a first step, we developed a reusable middleware layer to support the management of service lines, on top of which we implemented the application-specific document processing services.

Service line management support layer. In addition to the enabling middleware services that address the core requirements of (i) versioning support, (ii) application-level multi-tenancy and (iii) dynamic composition (cf. Section 6.3.3), we have developed a generic support layer to facilitate the development and management of service lines [216].

This layer offers several service interfaces to the different stakeholders and to the application layer. The `IFeatureManagement` interface provides the SaaS architect/developer with a service to manage the feature model and the feature mapping specifications. Tenant administrators, which are (in the context of service lines) employees of the tenant organization (cf. self-service in Section 6.3), have access to the `ITenantManagement` interface. This interface enables the registration of new tenants to the service line as well as the tenant-specific customization by selecting features based on the tenant's preferences and by parameterizing these features. The internal `ConfigurationMapping` service ensures that feature configurations are automatically transformed into tenant-specific software configurations by means of the feature mapping specifications. Finally, the `ITenantConfigurationRetrieval` interface is offered to the multi-tenant application, more specifically to allow the look-up of tenant-specific software configurations and to enable the run-time composition of the SaaS application (cf. Section 6.3.6).

We have implemented these services as a reusable middleware layer using Java EE. The front-ends are realized with Java Servlets, JSPs, and RESTful services; the business logic is implemented using session beans; for persistence we used entities that are stored into a MySQL database. Furthermore, the custom grammar to express the feature-to-software-composition mappings is developed using ANTLR. This middleware layer is designed for reuse across different service lines.

Prototype implementation. The document processing prototype is built on top of JBoss Application Server 7, which is Java EE 6-compliant. It uses jBPM and Drools for customizable document processing workflows, and MySQL for persistence. To achieve application-level multi-tenancy and tenant-aware dynamic composition within services, we rely on the modular middleware layer that we presented in previous work [214]. To realize the throughput SLAs in the running example, we have provided alternative service implementations that each guarantee a certain throughput (premium versus normal).

6.4.5 Tenant provisioning

As indicated in Section 6.3, the actual provisioning of (new) tenants becomes relatively straightforward by leveraging on the investment of service line development and deployment. During the Tenant Requirements Analysis activity, tenant administrators use the `ITenantManagement` configuration interfaces presented above, to select and parameterize features based on their requirements. For *Tenant A* and *Tenant B* of the running example, this configuration activity has resulted in the following two feature configurations:

```
Tenant A:          Tenant B:
  Separate_File      Extract_from_Data
  Premium            Custom
  Paper              Signing
                     Standard
                     Email_and_Paper_after_Deadline timeout:48
                     Archival duration:24
```

These feature configurations specify the set of selected features, and optionally the associated attributes and their values. The validity of these configurations are checked by looking at the constraints (mandatory features, dependencies, etc.) defined in the feature model (cf. Section 6.4.2). This way, errors and invalid configurations can be avoided before activation.

Subsequently, these feature configurations are automatically transformed into tenant-specific software configurations (cf. Configuration Mapping activity). Because of the automation, the effort associated with this activity is minimal and the updated configurations are thus immediately effective for the end users of the tenants.

Finally, the composition of particular services into the document processing workflow occurs at run time based on the tenant-specific software configuration that is applicable to the incoming request (cf. Configuration Activation activity). For example, when a document processed for *Tenant A* is ready to be delivered to the appropriate recipient, the service line will activate the `PostalService` variant for the `DeliveryChannel` variation point (corresponding to the `Paper` feature). The document is first sent to the printing service and next the printed document is delivered via the postal service to the appropriate recipient's address. In comparison, a delivery workflow that is started for *Tenant B* (cf. Listing 6.1) sends the output documents via email to the recipients. When a particular document has not been fetched after a timeout period of 48 hours, it is also printed and delivered via the postal service.

These two different tenant-specific configurations of the document processing service line regarding document delivery are activated at run time and can execute simultaneously. Further details about the run-time customization of the document processing service line can be found in [74].

6.5 Evaluation

This section evaluates our service line engineering (SLE) method as follows. We perform a comparative cost analysis of the required effort in each of the scenarios of Section 6.2.1 and substantiate the claimed efficiency benefits in Section 6.5.1. This section ends with a discussion of the strengths and limitations of the presented method in Section 6.5.2.

6.5.1 Service line efficiency

In Section 6.2.1, we presented five key development and management scenarios that occur in the life-cycle of a multi-tenant SaaS application. In this subsection, we re-evaluate these scenarios in the context of our SLE method. Specifically, we compare the current state of practice (discussed in Section 6.2.1) with our SLE method *by analysing the cost for the SaaS provider (in terms of required effort)* to perform these scenarios. We leverage on this cost analysis to substantiate the claimed efficiency benefits. The main cost variables that affect the required effort are t (the number of tenants), f (the number of features), and v (the average number of variation points per feature). The different costs are functions of these variables and they grow with an increasing value of these variables.

First, we look at the cost to provision (new) tenants with a configuration (cf. Scenarios #1 to #3). Next, we discuss the cost related to evolving and updating the service line itself (cf. Scenarios #4 and #5).

Provisioning new tenants

The cost to provision (new) tenants with a configuration has a major impact on SaaS scalability: if the provisioning cost grows significantly with an increasing number of tenants, then this hinders scalability of the SaaS offering. We discuss for each relevant scenario the associated cost for the SaaS provider.

Scenario #1: Initial development of SaaS application. Our SLE method clearly requires more effort in the initial development phases: variability analysis has to be performed, leading to the creation of an initial feature model. In addition, the feature-to-software-composition mappings between features and software artifacts have to be specified explicitly. This extra cost compared to the current state of practice thus mainly depends on the number of features f and variation points per feature v.

Scenario #2: Provision a new tenant. In the current state of practice (see Section 6.2.1), the SaaS provider has to (manually) specify and test a tenant-specific software configuration for f features and an average of v variation points per feature that have to be bound to a particular variant.

With the SLE method however, once the service line is up and running, we can rely on self-service and automation and so provisioning new tenants does not require any intervention from the SaaS provider. Specifically, the tenant administrator (i.e. employee of the tenant) can now customize the SaaS application by selecting and configuring the desired features via a management interface. Subsequently, a tenant-specific software configuration is generated in a consistent way and —given that the feature mapping specifications themselves are well-tested and correct— free of errors. Thus, with respect to provisioning new tenants, the SLE method results in a constant cost for each tenant and no extra cost in terms of required effort for the SaaS provider.

Scenario #3: Update tenant-specific configuration. A service line allows the tenant administrator to update the tenant's feature configuration via the management interface. Based on this updated feature configuration, a new tenant-specific software configuration is generated, which is immediately effective for the service line, without any (human) intervention. Essentially, the cost of updating a tenant-specific configuration is equal to the cost of provisioning a new tenant (see Scenario #2).

Conclusion. This cost analysis indicates the trade-off between spending effort on early development of the service line versus the costs and effort to configure and compose the running service line after deployment. The efficiency benefits are realized by leveraging service line design elements (feature mappings, composability of features, feature traceability, modularity of feature implementations, etc.) that have been created (and thus incurred an extra cost) during the initial development of the service line.

In the current state of practice, the effort to manage the running SaaS application will grow drastically with an increasing number of tenants t, which hinders the scalability of the SaaS offering. In comparison, because of the automation of most activities in the configuration and composition processes, the increase of effort in the service line engineering method will largely depend on the number of features f and not the number of tenants. Since the number of tenants is typically a magnitude larger than the number of features and the cost to specify a feature mapping is generally smaller than the cost to create an entire software configuration[3], the higher initial effort required by our method to develop the service line is rapidly reversed and more importantly will enable the SaaS provider to realize the benefits of scale required to run a profitable SaaS offering.

Especially the required effort to realize Scenario #2 (i.e. provisioning new tenants) is an important metric, as it expresses the time to service and provision new tenants. A faster time-to-provision will provide the SaaS provider with a real business advantage over its competitors.

Maintenance and evolution

Evolution entails introducing new requirements as well as updating and maintaining multi-tenant SaaS applications. Moreover, in SaaS applications updates can be performed more frequently, requiring the SaaS provider to adopt a smoother, more gradual strategy of continuous evolution. Because of this continuous flow of updates,

[3]This is confirmed in our prototype: the average size of a mapping specification is around 12 lines, while the average size of the generated software configurations is around 44 lines.

the cost of evolution also has an important impact on SaaS scalability and profitability. We discuss the initial analysis of the cost of evolution based on Scenarios #4 and #5.

Scenario #4: Support new requirements. In the current state of practice, new requirements incur a heavy cost for verifying and (manually) updating *each* tenant-specific software configuration with f features and an average of v variation points per feature (see Section 6.2.1).

In a service line, new and unforeseen requirements will affect the feature model (cf. the feedback loop in Fig. 6.1). For example, a new top-level feature could be introduced or an existing feature could be further decomposed into one or more sub-features. Adding a feature requires extending the feature model and the service line architecture, as well as specifying a feature mapping for the new feature. This will not (or hardly) affect existing features, mapping specifications and feature implementations thanks to the composability of the software variants and the reusability of the feature mappings. However, when adding new features, the existing tenant-specific configurations must be verified and potentially updated to fix compatibility issues.

In summary, the cost to add a new feature includes the effort to create a new feature mapping that defines v variation points, and a constant cost per tenant to verify his feature configuration and to handle issues. We consider it to be a constant because a service line offers automated support for verifying whether existing feature configurations comply to the constraints of the updated feature model and for localizing issues. This enables the SaaS provider to quickly detect any issues and handle them (e.g. by setting a default feature for a new variation point, or by supporting multiple co-existing versions of the service line), independently of the size or complexity of the feature configuration.

In general however, the effort to support new requirements can be significantly higher, especially when these involve major changes in the feature model and the service line architecture. However, the traceability links between features, architectural components and software artifacts enable the SaaS provider to quickly identify and manage these interactions, and the modularity of the architectural components will limit the extent of ripple effects throughout the service line architecture and implementation (cf. Section 6.3.4).

Scenario #5: Update and maintain SaaS application. Using our SLE method, this scenario only involves updating the feature mapping specifications instead of the entire configuration of *all* tenants (cf. Section 6.2.1). Because of the traceability support in the service line architecture, the SaaS provider can easily identify which specific feature mappings have to be updated, and a smaller amount of mappings have to be updated instead of all of them (i.e. f).

In case the updates conflict with existing tenant-specific requirements, this problem will be detected during the update process of the mapping specifications. The tenants to which such conflicts apply can easily be identified, i.e. these are the tenants who selected this particular feature in their configuration. In this case, the SaaS operator can decide to perform a gradual roll-out of the updates, temporarily hosting the old as well as the new versions of the features and software variants. Based on the version information in the tenant-specific configurations and the feature mapping specifications, the appropriate version will be activated in the service line at run time.

Conclusion. We have shown that the use of a feature model and feature mappings is also beneficial to evolving and updating the service line itself. Many changes can be encapsulated within the feature model and the feature mappings, while the current state of practice requires the SaaS provider to verify and update *all* configurations (because of the ripple effects). Similar to the proportion between the effort required for creating a feature mapping and for creating a tenant-specific software configuration, the effort for updating a feature mapping is in general also smaller than the effort for updating an entire software configuration. As a consequence, the higher initial effort during development also enables the SaaS provider to realize benefits of scale while evolving and maintaining the running service line.

Moreover, the SaaS paradigm enables SaaS providers to update their application continuously and rapidly. Consequently, the initial development effort is a one-time (or infrequent) investment cost compared to the continuous updates ("develop once, adapt/evolve forever"), which results in major cost benefits in the long term w.r.t. the management and customization of multi-tenant SaaS applications.

6.5.2 Discussion

This section provides an open-ended discussion on the strengths and limitations of the service line engineering method presented in this paper. In parallel, we discuss future work.

Non-functional tenant requirements. As indicated in Section 6.3.1, the context of multi-tenant SaaS introduces additional variability, especially driven by non-functional requirements (or quality attributes) such as availability, performance and security. These requirements are represented as SLAs in the feature model. In the other activities of our method however, these non-functional requirements also have to be taken into account: quality attributes are often important architectural drivers and put constraints on the deployment environment. For example, a monitoring service is necessary to monitor the resource usage on a per-tenant basis, while a policy enforcement engine has to ensure that the delivered performance is in compliance with the tenant-specific SLAs. These enabling middleware services are identified during the design of the architecture and should be provided by the underpinning platform.

Furthermore, a SaaS application can depend on third-party services. For example, in the document processing application, the SaaS provider could decide to outsource the distribution of printed documents. Even in such a cross-organizational context, the different SLAs still have to be ensured. In the state of the art, several solutions exist for service-oriented product lines to negotiate with third-party services to provide quality-of-service (QoS) guarantees, e.g. [105] and [114]. It is worth to investigate the applicablity of these approaches in a multi-tenant SaaS context.

Our mapping specification (see Section 6.4.3) focuses on features (and SLAs) that can be mapped to a set of architectural components (and their corresponding software artifacts). With availability and performance SLAs this is often not the case. These features (and their attributes) provide input for the underpinning middleware services or a broker. In future work, we will extend our middleware support to include these SLAs in the automated configuration mapping activity.

Support for evolution and maintenance. Although our method covers the evolution and maintenance aspects of service lines, we did not provide specific service management functionalities to support the SaaS provider with the evolution and maintenance of service lines. For example, additional tool support is required to enable the SaaS provider to analyse the impact of an update on the service line architecture and the different tenant-specific configurations. In addition, since multiple versions of the features, configurations and software artifacts can co-exist, the SaaS provider requires explicit support to monitor and manage (i.e. upgrade, disable, make obsolete, etc.) these versions within the running service line. This middleware support is currently lacking in our implementation. In fact, most of these issues are open challenges for SaaS applications. In future work, we will further investigate these issues to support effective operation of service lines.

Besides the impact analysis of an update based on the service line architecture and the different configurations, updates have to be tested and validated thoroughly before they are rolled out. This is especially relevant in a multi-tenant SaaS context, because conflicts or errors (e.g. caused by updates) potentially affect all tenants and thus the availability of the SaaS application. To realize continuous and seamless updates to service lines (part of service line operation), there is a need for integration with frameworks for automated testing and gradual roll-out of updates. These requirements are out of scope for this paper, but are relevant for a service line engineering method to be investigated in future work.

Finally, it is possible that during the tenant requirements analysis new requirements are discovered that cannot be addressed. For example, the new variations that have to be introduced cannot be implemented without run-time conficts. Note that this issue is not limited to service lines, as it can also occur in traditional software product lines. However, multi-tenant SaaS applications have the additional constraint that these different requirements should be addressed by a single application instance and customization should be achieved by means of dynamic composition. When the requirements cannot be met because of this multi-tenant context, the SaaS provider could decide to deploy and operate a second (different) instance of the service line. However, this requires the SaaS architect to make a distinction between deploy-time and run-time variability (e.g. by using variability types [67]), and multiple deployments evidently result in higher resource usage. Therefore, the SaaS provider has to make a comparative assessment (ability to address more requirements versus increasing management complexity and resource usage).

6.6 Related work

This section discusses three categories of related work: (a) dynamic and service-oriented product lines, (b) customization of multi-tenant SaaS applications, and (c) variability management in software architecture.

6.6.1 Dynamic and service-oriented product lines

The work on service-oriented product line engineering (SOPL) [50, 83, 128] tries to combine the benefits of the open-ended model with late-bound variability in service engineering (SE) with the closed world of managed variability and planned reuse in SPLE. The fundamental building block of a service-oriented architecture (SOA) is a service; the application (i.e. the product instance of a SOPL) is thus an orchestration of services. However, in the case of SaaS, the application itself is a service. The key difference however remains that a SOPL creates different product instances per customer that have to be deployed separately by the customer (possibly in an automated way). Although a (multi-tenant) SaaS application can be developed based on a SOA, a service line results in a single, multi-tenant product instance (deployed and managed by the SaaS provider) that is dynamically customized based on the tenant-specific configurations.

[149] tries to increase the cost efficiency of customizing web services: their work keeps track of which service variants are already deployed to ensure only one instance is deployed per variant. When compared to application-level multi-tenancy, this approach is less efficient in terms of operational costs (resource usage and

maintenance effort), especially when many variants exist (i.e. high amount of features). Therefore it is not suitable for the development and customization of multi-tenant SaaS applications.

Furthermore, the concept of SOPL is often combined with dynamic software product lines (DSPL) [21, 86] because of the open-ended model of services: services can be discovered and consumed at run time [87, 105, 114]. In addition, a software system can use a DSPL to cope with changes in the current context or environment, or with unforeseen situations (e.g. [21, 164]). Although these approaches cope better with run-time variability, these dynamic adaptations are applicable to all users of the product instance (instead of on a per-tenant basis). There is no support for co-existing (tenant-specific) configurations. Moreover, a DSPL is still limited to the features that were included during feature selection.

Clearly, the current SOPL approaches are based on traditional, static SPLE approaches, which are not suited for multi-tenant applications. Our service line engineering method, however, focuses on the development and customization of multi-tenant SaaS applications to efficiently manage and reuse the different software variations and configurations, even at run time.

6.6.2 Customization of multi-tenant SaaS

Mietzner et al. [138, 141] have extensively studied the customization of multi-tenant SaaS applications. They apply variability modeling techniques from the SPLE domain to support the management of variability in service-oriented SaaS applications, more specifically BPEL processes in the cloud. The variability is realized by defining variability descriptors that create application templates. Tenants fill in these application templates to create tenant-specific BPEL processes, which have to be deployed separately. Our work, however, introduces an integrated method for the development and customization of multi-tenant SaaS applications, independent from the underpinning technologies. In addition, we provide support for the management of the different software variations and tenant-specific configurations. The solutions presented by Mietzner et al. can be integrated in our work as a possible tenant-aware customization technique for workflows (instead of the current solution in our prototype using jBPM and Drools).

In previous work [214], we have developed a middleware layer for PaaS platforms to enable tenant-specific customizations, while limiting the performance overhead and preserving the operational cost benefits. It focuses on SaaS applications consisting of a single service and dynamic composition is supported via dependency injection. As it provides a tenant-aware customization technique, this middleware layer could be one way to realize the service line composition activity of our method (as illustrated by our prototype).

[186] identifies requirements for a variable architecture for multi-tenant SaaS applications and describes how their existing architecture for self-adaptive systems can be extended to support multi-tenancy. In contrast, the focus of our work is on providing an integrated method for efficient customization of multi-tenant SaaS applications and is to a large degree independent from any specific system or platform.

In [187], the authors propose a concept for dynamic configuration management, with different stages [55] and stakeholder views in the configuration phase. Their work is limited to the problem space and covers only one activity of our method, i.e. the tenant requirements analysis. In this paper, we focus on the tenant and the SaaS provider, but our approach does not exclude more stakeholders and stages and is thus complementary.

Furthermore, work has been performed in the context of architectural patterns for developing and deploying customizable multi-tenant applications. [140] presents several multi-tenancy patterns and describes how services in a service-oriented SaaS application can be composed and deployed using these different multi-tenancy patterns. In this paper we focus on customization in application-level multi-tenancy, which maps to their single configurable instance pattern. Kabbedijk et al. [97, 98] present architectural patterns to realize run-time variability in multi-tenant SaaS applications. These patterns are relevant during the service line development process (Activity 2) to implement composable variants. For example, our dynamic composition technology [214] corresponds to the Component Interceptor Pattern.

6.6.3 Variability management in software architecture

Several solutions exist for the representation and management of variability in service compositions, mainly limited to providing extensions to the BPEL language [150]. [1] introduces a multiple-view modeling approach for variability in service-oriented architectures (SOA) in a platform-independent way. However, their approach focuses on a SOA context with views for business processes and service interfaces, and it does not support dynamic composition. Therefore, it is not directly applicable in a multi-tenant SaaS context.

[67] suggests a more generic approach for representing variability in SOA. However, their meta-model inherently assumes multiple instantiations of a SOPL, and thus not supports co-existing configurations. In addition, it does not associate variation points and variants with software artifacts. A more extended meta-model of an architectural viewpoint for variability is proposed by [68]. Such a variability viewpoint facilitates the representation and analysis of variability in software architectures. The proposed variability viewpoint is complementary to our method and could be integrated in the service line architecture, but this requires adapting our feature mapping specification.

In addition, the meta-model of [68] would have to be extended with versioning support and with the concept of (tenant-specific) configurations.

[82] integrates variability management support into an existing approach for describing and analysing software architectures. This results in one consistent model which links architectural artifacts (e.g. requirements, features, components) with the variability models to support full traceability. Our work is complementary, as we defined an integrated method for service line engineering that depends on variability management and traceability to support efficient reuse of software variations and configurations. Therefore, provided some modifications, the solution by [82] could be a suitable enabler to implement our method. Again, their solution would have to be modified to support versioning, and thus to support co-existing configurations and evolution of service lines.

Furthermore, [18] and [38] present meta-models to support variability and traceability throughout the process of software product line engineering (SPLE). As above, this work is complementary to our method, more specifically the service line development process, provided some modifications regarding the multi-tenant SaaS context.

Finally, [93] supports multiple perspectives in feature-based configuration. Configuration views are tailored to the profiles of the various stakeholders. In this paper, we have considered a limited number of stakeholders (employees of the tenant and the SaaS provider). However, other stakeholders could be involved in the tenant requirements analysis, for example a service reseller. Therefore, this approach of multi-view feature-based configuration is complementary and can be plugged into our service line method, more specifically as part of the tenant requirements analysis activity (see Section 6.3.4).

6.7 Conclusion

This paper presents an integrated service line engineering method for multi-tenant SaaS applications. This feature-oriented method supports co-existing tenant-specific configurations, and facilitates the development, deployment, run-time configuration and composition, operation and maintenance of SaaS applications, without compromising the scalability. The method is generic in the sense that each activity is open for existing work to be leveraged upon, yet it imposes some specific constraints (e.g. composability and traceability of features) and some enablers (e.g. multi-tenancy, versioning support, dynamic composition) to obtain the desired variability in the service line.

We have validated this method in the development of a service line for a real-world industrial SaaS application in the domain of document processing, and performed a comparative cost assessment with respect to the current state of practice. These results clearly show that this work presents a better, more efficient trade-off between the design-time effort required to develop the initial service line, and the run-time effort required to operate, evolve and maintain the service line as a whole.

A body of knowledge about a computing paradigm typically establishes itself after years of practical application and experience (e.g. the documentation of the GoF design patterns after the wide-spread adoption of object orientation). Cloud computing, and more specifically multi-tenancy at the application level (SaaS), is an increasingly prominent and important software delivery model, which also has a profound impact on how the software is developed and maintained. However, these trends are currently underinvestigated from a software engineering perspective.

The work presented in this paper represents an initial exploration and consolidation of how to develop and build qualitative SaaS applications. More specifically, the presented method focuses on realizing efficiency improvements in terms of reuse and modularity of service variations, as well as on maintenance and evolution aspects of the entire service line.

Acknowledgements

We would like to thank Awais Rashid for the fruitful discussions on the concept of service lines. We also thank the reviewers for their helpful comments to improve this paper. This research is partially funded by the Research Fund KU Leuven (project GOA/14/003 - ADDIS) and by the CUSTOMSS project, which is co-funded by iMinds (Interdisciplinary institute for Technology), a research institute founded by the Flemish Government. Companies and organizations involved in the project are Agfa Healthcare, Televic Healthcare and UnifiedPost, with project support of IWT.

Chapter 7

Conclusion

This chapter concludes the dissertation. Section 7.1 summarizes the contributions and the evaluation efforts. Furthermore, Section 7.2 discusses its limitations and directions to future research, while Section 7.3 discusses the relevance and applicability of this work. Finally, Section 7.4 presents a long-term vision and identifies major open research challenges.

7.1 Contributions and evaluation

The main goal put forward in this dissertation is to introduce the necessary flexibility into multi-tenant SaaS applications without compromising the key benefits of cloud computing, such as cost efficiency and scalability. More specifically, this dissertation addresses the following challenges: (i) the actual realization of the different tenant-specific customizations in a concurrent and isolated way, and (ii) the management of the many application and configuration variations in a scalable way, while (iii) limiting the performance overhead and the additional engineering complexity.

Contributions. As the first contribution of this dissertation, we have investigated three different and representative PaaS offerings and have performed a *gap analysis* with respect to the requirements stated above (see Chapter 2). This analysis clearly identified a lack of support for application-level multi-tenancy in all its aspects as well as for the migration of SaaS applications (i.e. limited portability) [215]. In order to improve the applicability of this work, the following contributions leverage these existing PaaS offerings as base platform to build further upon.

The feature-based approach presented in this dissertation to address these challenges is twofold and maps directly to the two core contributions of this work: (i) a *middleware framework* to support co-existing variants in multi-tenant SaaS applications (see Chapters 3, 4 and 5), and (ii) a software engineering method, called *service line engineering (SLE)*, to introduce and manage variability throughout the entire lifecycle of multi-tenant SaaS applications (see Chapter 6).

The middleware framework enables the customization of multi-tenant SaaS applications driven by the tenant-specific configurations, while preserving the operational cost benefits and limiting the unavoidable increase in application engineering complexity [214]. More specifically, it offers support for the run-time composition of software variants as well as the run-time enforcement of performance isolation in compliance with tenant-specific SLAs. We have achieved this flexibility without compromising the performance and scalability of the applications.

Further building on our experience with this middleware framework, we have developed a generic feature-based method to develop and manage a customizable, multi-tenant SaaS application. Just as the software product line engineering (SPLE) method aims to exploit commonalities and to improve variability management in traditional software products, the SLE method focuses on managing variability in multi-tenant SaaS applications in a cost-efficient and scalable manner [217]. It provides a better, more efficient trade-off between the design-time effort required to build and construct a customizable SaaS application, and the run-time effort required to provision tenants and to update and maintain the application [216, 217]. This has been achieved by means of the end-to-end feature-based approach, self-service and automation.

Evaluation. Based on the goals of this dissertation, the main evaluation criteria are (i) high cost efficiency, in particular low operational costs and low application engineering cost, (ii) higher flexibility, (iii) low performance overhead and high scalability, and (iv) versatility of the middleware framework and the SLE method. Several of these criteria cannot be translated into specific metrics, and are thus hard to evaluate. Therefore, the evaluation approach in this dissertation consists of *validation via several prototypes and two application cases*, the development of *cost models*, and *measurements of performance and resource usage*.

As part of the validation strategy, the different middleware platforms, concepts, methods and techniques have been implemented in the realization of different prototypes (see `https://distrinet.cs.kuleuven.be/projects/CUSTOMSS/`). These prototypes are based on industry-relevant application cases, namely the component-based, multi-tier hotel booking application and the batch-driven document processing application consisting of several services that cooperate in a workflow.

With respect to the middleware framework for co-existing variants, this dissertation focused on run-time composition of software variants and on performance isolation. Both middleware contributions have been validated using the two application cases and different customization techniques, on top of several existing cloud platforms (see Chapter 3). Chapters 4 and 5 each focus on one specific instance of the middleware framework. These validations demonstrated that our approach introduced the necessary flexibility into multi-tenant applications. In addition, the implementation and use of these prototypes illustrate the versatility of the middleware framework.

The validation of the SLE method has been mainly conducted in the development of a service line for the document processing application, in the context of the CUSTOMSS project [53] and in close collaboration with an actual European SaaS provider (see Chapter 6 and [216]). Moreover, the SLE method and its principles have been conceptually validated in two other SaaS application cases in the healthcare domain (also in the context of the CUSTOMSS project [53]). In particular, we compared the state of practice with the proposed SLE method in terms of the required effort for the SaaS provider. This assessment indicated that the effort to provision tenants and to maintain service lines is independent of the (growing) number of tenants, and thus results in more scalable variability management, despite the higher (initial) development cost.

Furthermore, we have developed a cost model to compare the operational and re-engineering costs between a single-tenant and multi-tenant application (see Chapter 4). We also included the cost of flexibility into this model. Moreover, we have measured the resource usage of these approaches, focusing on two commonly used parameters for billing, in order to evaluate the cost model. The cost model as well as these measurements clearly indicated that application-level multi-tenancy leads to major operational cost savings, which is also confirmed by [35, 110]. The other chapters built further on these results. With respect to the re-engineering cost, we used lines of source code to analyse the impact of the use of our middleware on application development.

Finally, the impact of the middleware framework and the introduced flexibility on the performance and scalability has been evaluated. More specifically, we measured the performance overhead of our prototypes compared to the default, state-of-practice approach. The results show that we were able to minimize this performance overhead. The scalability of the middleware framework has been tested within the capabilities (and restrictions) of the underpinning platform. For example, on top of Google App Engine (GAE) [77], we were able to scale out to 40 application instances, while on our private cloud we used up to 22 worker threads spread over 11 virtual machines. In both cases our prototype was not the limiting factor.

7.2 Limitations and future work

This section discusses the limitations of the work presented in this dissertation as well as directions for future research. More specifically, we focus on (i) further evaluation of the middleware architecture and the SLE method, (ii) complementary research with respect to performance isolation algorithms, evolution, and integration with the underpinning cloud platform, and (iii) customizable SaaS applications in a multi-cloud and cross-organizational context.

7.2.1 Further evaluation

Although the middleware framework and SLE method have been validated and evaluated in different prototypes using two different types of SaaS applications, further evaluation is always useful to endorse and improve our contributions.

Cost. Cost efficiency is a major motivational factor for this dissertation. It is commonly accepted in the state of the art that application-level multi-tenancy maximizes resource sharing and thus leads to major operational cost benefits. This is also confirmed by our cost model and our experiments (cf. Chapter 4). However, it is definitely worth to further investigate these operational cost benefits as well as the impact of our middleware and flexibility by (i) extending and improving the cost model, (ii) measuring all relevant parameters in a more accessible and controllable environment (e.g. a private cloud platform), and (iii) effectively associating the cost model and the measurements with actual costs.

Furthermore, the comparative cost analysis to evaluate the SLE method can be further extended and formalised in a cost model, similar to Chapter 4. Currently, the cost analysis is limited to the required effort for the SaaS provider based on three main cost variables: the number of tenants, the number of features, and the average number of variation points per feature. Additionally, the effort for the tenants, the average update and evolution frequency, as well as a more detailed development cost can be taken into account to substantiate the efficiency benefits. This is currently work in progress.

Finally, it is relevant to quantify the effort of (re-)engineering a SaaS application in order to fit the feature-based application model and to use the presented middleware framework. However, such an evaluation of the applicability and (re)usability of the proposed solution is hard to perform. We already gained some knowledge from the use of the middleware framework by several MSc and PhD students, and from measuring the code changes (cf. Chapters 3 and 4). But a more systematic approach, for example empirical evaluation, is required to draw more evidence-based conclusions. This (re-)engineering effort is also related to the versatility of the middleware framework.

Versatility and flexibility. The versatility property of the middleware framework and the SLE method refers to their general applicability as well as to their openness and extensibility. In order to further validate and improve the applicability, the SLE method and the middleware framework have to be applied to more different (types of) applications and domains. For example, in our ongoing research, we are effectively applying these concepts, methods and platforms in the domain of healthcare [127] and payment [167]. Such an extensive validation also enables us to determine the required level of flexibility and variability in multi-tenant SaaS applications.

Furthermore, it is relevant to investigate how well different composition mechanisms and performance isolation algorithms can be integrated into the middleware, as well as other customization types can be supported. For example, dependency injection (DI) [66] is limited in terms of extensibility of variations points and amount of variants per variation point [204]. Therefore, a SaaS provider can opt to use a more powerful composition mechanism, such as dynamic aspect-oriented programming (AOP) [165, 169, 206], context-oriented programming (COP) [92], and reflection [123]. The same applies to the mechanism to customize workflows. Moreover, this allows us to compare and analyse these customization mechanisms, and to identify the limitations of the framework in terms of customizability and flexibility. Similarly, the openness of the SLE method with respect to existing work in the state of the art can be evaluated.

Furthermore, additional implementations of the middleware architecture can be developed on top of different PaaS offerings and cloud-enabling middleware platforms, for example Red Hat OpenShift [175] and Cloud Foundry [211].

Scalability. The scalability of the middleware framework has to be evaluated more extensively. As the architecture supports multiple implementation and deployment strategies, it is definitely worth to analyse which strategies are more suitable at a particular scale, and to evaluate and compare the different implementations. This is especially relevant for the workflow-based applications. For example, we expect that the centralized, in-memory task queue (cf. Chapter 5) will scale less with a high amount of workers. In this case, a distributed queue could be a more appropriate solution, despite the performance overhead to keep it consistent.

7.2.2 Complementary research and extension

Besides the need for a more extensive evaluation, there is a need for complementary research which was not in the scope of this dissertation. To ensure *performance* *isolation*, the middleware depends on an algorithm to assign priorities based on the monitoring data and the different tenant-specific service level agreements (SLAs). In the state of the art, several *algorithms* exist to predict potential violations of performance SLAs and to enforce performance isolation and fairness, for example [117, 193], but these algorithms are not necessarily applicable to multi-tenant SaaS applications. Moreover, the characteristics of an application have a major impact on the selection of an appropriate algorithm. The middleware framework can therefore provide an execution environment to compare alternative algorithms, instead of or in addition to simulation.

Furthermore, the SLE method covers the *evolution and maintenance* aspects of customizable SaaS applications, but the necessary middleware support for this was out of scope of this dissertation. For example, the SaaS provider requires supporting middleware to manage different versions of features, components and software artifacts, to perform an impact analysis of changes, as well as to automatically test and gradually roll-out updates. A future research challenge is extending the SLE method and the underpinning middleware platform to realize the main assumptions defined by the SLE method.

Finally, with respect to non-functional requirements (e.g. availability, performance and scalability), *integration* is required of the presented middleware framework with the underpinning platform as well as with the monitoring and cloud management services (cf. Fig. 1.1). As explained in Chapters 5 and 6, the non-functional requirements are specified in tenant-specific SLAs that are taken into account by the middleware. However, the enforcement of these SLAs cannot fully occur at the application layer. For example, the dynamic resource management system is responsible for scaling out and to respond to failures, while the end-to-end performance also depends on the cloud services provided by the underpinning platform. In future work, we will investigate the extension and integration of this work with cloud management systems, as well as the impact of the platform-provided cloud services on performance isolation.

7.2.3 Multi-cloud and cross-organizational context

With the advance of cloud computing, cloud consumers and providers gain more experience with the advantages and disadvantages of current public and private cloud solutions. This results in a growing interest in (i) multi-cloud environments in general to improve availability and to address vendor lock-in, (ii) hybrid clouds to combine high scalability with increased control, and (iii) cross-organizational applications and systems to compose and integrate cloud services from different providers. Our ongoing and future work focuses on several challenges that SaaS providers face when adopting these approaches.

Portability and interoperability of SaaS applications are major challenges, especially in a multi-cloud setting [166]. These challenges refer to the migration of SaaS applications from on-premise to the (public) cloud and across different cloud offerings, as well as the cooperation between these diverse and distributed applications and systems. Thus, the goal is to avoid vendor lock-in and to improve availability. For example, portability should be improved with respect to typical cloud services such as scalable storage and caching, multi-tenancy, application management and monitoring, as we identified in Chapter 2. As part of our ongoing research, we have developed the PaaSHopper middleware for multi-cloud SaaS applications. The core of this middleware consists of an initial abstraction layer that enables portability over multiple cloud services across various PaaS platforms [173]. A more long-term solution to improve portability is standardization, which by nature happens at a late stage of technology maturation.

Other important properties in the context of hybrid cloud applications are *flexibility and control.* Tenant organizations want to maintain control over the execution of their application and the processing of their data (public cloud versus private data center). The PaaSHopper middleware provides tenants the flexibility to control the execution in a fine-grained way via application-specific policies [59]. Driven by these policies, the middleware decides at run time in which part of the hybrid cloud a particular request or task should be executed. However, more expressive policies are required, for example to take into account the cost and monitoring information of different public clouds. In addition, to achieve run-time deployment of the PaaSHopper middleware and the application components, we have to integrate this work with dynamic resource management systems.

Furthermore, a SaaS application can in its turn consume other, external software services (i.e. across corporate boundaries). This is particularly relevant in the context of Business Process as a Service (BPaaS) [8]. However, tenant-specific customizations have to be activated and propagated across this distributed, *cross-organizational* and definitely heterogeneous context. This is strongly related to the interoperability challenge. In previous work, we have developed a coordination middleware to enable cross-organizational customization in distributed service systems [212]. To address the heterogeneity a feature ontology is introduced as additional abstraction layer, which specifies the different high-level features in a particular domain or service network. However, this approach has not been evaluated in a multi-tenant context. Since ontologies are also employed in other work to support interoperability in heterogeneous distributed systems [30, 174] as well as in combination with feature models [54], this is certainly a track for future research.

7.3 Relevance and applicability

Cloud providers inherently aim for growth. Therefore, economies of scale are key for a profitable cloud offering: the offering should scale with the increasing number of consumers without the operational costs skyrocketing. This results into the requirements of efficient utilization and management of resources, which will be reflected in both the profit margins and the pricing strategy. *This cost efficiency as well as differentiation (e.g. via customization) enable a cloud provider to achieve a competitive advantage relative to its rivals [170].* This dissertation aims to combine both, and is thus directly relevant for the competitiveness of SaaS offerings.

As discussed in Chapter 1, multi-tenancy (at any level of the stack) is a key characteristic of cloud computing to leverage economies of scale by lowering the operational costs and by improving scalability. Moreover, with the growing awareness of the high power consumption by the many data centers, multi-tenancy can also play a role in the trend of green computing [109] by multiplexing resources among multiple tenants and thus by limiting the total resource usage [20, 71].

The leading technology research and advisory companies, such as Gartner and IDC, predict worldwide spendings on cloud services in the tens of billions of dollars [72, 95]. The majority of these spendings go to SaaS offerings, as these are spread over many diverse application domains, which results into a much larger market. In comparison, IaaS and PaaS are more generic services that require large investments in infrastructure and extensive knowledge on complex system software (e.g. virtualization, application servers). Therefore, we expect further consolidation to only a few big players in the IaaS and PaaS market (both public and private offerings), such as Amazon, Google, Oracle and Microsoft.

Often SaaS providers will rely on these (external) cloud providers to deliver the necessary infrastructure and platform, allowing them to focus on application development without having to make huge infrastructure investments. In this context, SaaS providers have less or even no control over the underpinning platform and its resource usage. Application-level multi-tenancy is then an effective approach to employ the available resources in a cost-efficient way and thus to run a more profitable SaaS offering.

However, customizability is also relevant to achieve a competitive advantage [170], especially in an enterprise or B2B context (probably less for consumer-oriented web applications): each tenant organization is unique with its own business requirements that evolve over time. Therefore, a one-size-fits-all approach is not suitable and SaaS applications should satisfy these tenant-specific requirements, possibly at an additional price. With an increasing number of tenants, cost efficiency, and more specifically *efficient mass customization*, becomes even more important.

The work presented in this dissertation can be a useful asset for SaaS providers to support the tenant-specific customization of their SaaS offerings in an efficient way. More specifically, the middleware framework can be used as an extension to the underpinning cloud platform, for example in the form of a library. It was a key decision in our approach to build upon existing PaaS offerings and cloud-enabling middleware platforms in order to improve the applicability of our middleware solution, instead of developing a completely new platform. In addition, we decided to comply with the common programming model in contrast to the metadata-driven PaaS platforms (e.g. Force.com [182]), again to improve applicability and to limit portability issues [215]. Therefore, we also used a common composition mechanism, i.e. dependency injection, in our prototype.

Secondly, the SLE method provides the SaaS provider with methodical support in the different stages to develop, deploy and manage a customizable, multi-tenant SaaS application, with an inherent focus on efficiency and scalability. Moreover, this does not exclude other existing software engineering methods or frameworks to be used during or across these different stages.

A good indication of the relevance of this work for SaaS providers is the adoption of our approach and concepts by an industrial partner in the CUSTOMSS project[1]. However, to improve further adoption, a more extensive cost analysis (cf. Section 7.2) is key in order to determine the total cost of ownership (TCO) of using both the presented middleware framework and SLE method. Furthermore, the applicability highly depends on the availability of full-fledged implementations of the middleware framework, including support for evolution, maintenance, and interoperability with different cloud platforms.

Finally, this work is also relevant for PaaS providers. More specifically, PaaS providers can extend their offerings with middleware support to develop, run and manage customizable, multi-tenant SaaS applications. By providing support to facilitate the development of customizable, multi-tenant SaaS applications, a PaaS provider can become more attractive for SaaS providers in comparison to its competitors.

7.4 Utility computing and beyond

Cloud computing aims to enable the flexible and on-demand consumption of resources and applications. Despite the many promises, the previous sections have indicated that a number of open challenges remain to obtain the key characteristics and thus benefits of cloud computing. However, to realize the promise of utility computing [162], even

[1]`http://www.iminds.be/userfiles/files/persberichten/2013/eng/Press%20Announcement%20-%20Strategic%20Partnerships%20-%2004%2012%202013.pdf`

more flexibility and cost efficiency are required in order to really deliver computing resources and storage as a utility like electricity.

For example, the Internet of Things (IoT) [13, 14] will surround us as a ubiquitous computing infrastructure. As these surrounding devices will often be idling, smart buildings and cities should offer the opportunity to spontaneously consume these available resources offered by the environment. Evidently, each device has different properties (e.g. server versus sensor), and thus the selection of these resources depends on the targeted purpose as well as the current load. Moreover, billing will also be affected by these properties.

Similarly, the software components and services running on these devices (as part of IoT applications) can be consumed on demand and combined into a new composition or application, even for a limited period (e.g. as long as the user is in the environment). Furthermore, tenants and end users have different preferences, which results into different short-lived compositions and configurations of these components and services.

Flexibility and lifecycle management are key to realize such complex distributed systems: flexibility to adapt and evolve towards different and changing contexts and environments, and lifecycle management to keep an overview on the different applications and services, including the many co-existing variants. Similar to the work presented in this dissertation, both an appropriate software engineering method and an advanced adaptive middleware platform is required to tackle these challenges, without neglecting important non-functional concerns such as availability, cost, performance and security. In China, researchers have proposed the name "Internetware" for such a software paradigm that provides technologies to develop applications and services for such open, dynamic and constantly changing environments [129].

From this perspective, the work presented in this dissertation represents a step towards this long-term goal. Concurrent and context-sensitive adaptation and composition remain crucial to achieve the necessary flexibility. In comparison to this work, not only the current tenant and the state and behaviour of the application define the context that drives the run-time adaptations and compositions, but also the purpose and application type, the location, and the available services provided by the surrounding environment. In addition, the middleware platform should be able to cope with larger as well as more diverse and heterogeneous environments. Interoperability thus remains and even becomes a more critical factor in order to support this level of flexibility. New concepts such as emergent middleware [29] could offer an adequate solution to overcome this heterogeneity.

Furthermore, the SLE method already provides variability management across the different stages of the development and exploitation lifecyle of long-running SaaS applications. However, in future open and dynamic environments, all variability cannot be determined in advance: applications and services are used in varying and unpredictable compositions during their entire lifecyle. Therefore, future research should focus on the extension of the SLE method with lifecycle management as well as on the development of middleware support for run-time evolution. The feature-based approach remains relevant, as it provides an appropriate abstraction level to communicate about variability towards the different stakeholders exposed to it (in our case, the SaaS architects, developers and operators, as well as the tenant administrators). However, the feature model should also be able to adapt and evolve dynamically as the context changes, thus representing a high-level, run-time model of the SaaS application.

In summary, we observe an increasing trend towards large, distributed and heterogeneous systems, consisting of long-running, shared services that will be used in different and varying compositions and contexts during their lifecycle. In this evolution, the importance of cost efficiency and flexibility will only grow. Indeed, future systems will have a longer (operational) lifecyle and will be deployed on a combination of different heterogeneous environments across different organizations. This puts the emphasis on requirements such as portability and interoperability, adaptability and evolution, and quality of service (QoS) concerning availability, cost, performance and security. Currently, this trend is already visible going from single cloud environments to federated architectures and multi-cloud applications, but becomes even more important in systems of systems (SoS) [37, 106, 124, 181], the IoT [13, 14], and utility computing [12, 162].

This dissertation fits nicely in this overall trend as it addresses the trade-off between flexibility and cost efficiency. Although this work stays within the scope of customizing multi-tenant SaaS applications within a single (cloud) environment, its twofold approach is generally applicable: the creation of (i) adaptive middleware to realize concurrent and context-sensitive adaptation and composition in an efficient way, even throughout heterogeneous environments and across organization boundaries, and (ii) a software engineering approach for shared applications and services to deal with evolution and lifecycle management, as well as to manage heterogeneity and variability not as an afterthought.

Bibliography

[1] M. Abu-Matar and H. Gomaa. “Variability modeling for service oriented product line architectures”. In: *SPLC ’11: 15th International Software Product Line Conference*. 2011, pp. 110–119. DOI: `10.1109/SPLC.2011.26` (p. 174).

[2] Amazon Web Services LLC. *Amazon Elastic Compute Cloud (Amazon EC2)*. `http://aws.amazon.com/ec2/`. [Last visited on April 1, 2014] (pp. 4, 15, 72, 73).

[3] Amazon Web Services LLC. *Amazon Simple Workflow Service (SWF)*. `http://aws.amazon.com/swf/`. [Last visited on April 1, 2014] (p. 117).

[4] Amazon Web Services LLC. *Amazon Web Services (AWS)*. `http://aws.amazon.com/`. [Last visited on April 1, 2014] (pp. 36, 73).

[5] Amazon Web Services LLC. *AWS Elastic Beanstalk*. `http://aws.amazon.com/elasticbeanstalk/`. [Last visited on April 1, 2014] (pp. 27, 66).

[6] Amazon Web Services LLC. *How quickly can I scale my capacity both up and down?* `http://aws.amazon.com/ec2/faqs/#How_quickly_can_I_scale_my_capacity_both_up_and_down`. [Last visited on April 1, 2014] (p. 120).

[7] V. Andrikopoulos, T. Binz, F. Leymann, and S. Strauch. “How to adapt applications for the cloud environment”. In: *Computing* 95.6 (2013), pp. 493–535. DOI: `10.1007/s00607-012-0248-2` (p. 74).

[8] T. Anstett, F. Leymann, R. Mietzner, and S. Strauch. “Towards BPEL in the cloud: Exploiting different delivery models for the execution of business processes”. In: *Services ’09: World Conference on Services - I*. IEEE, July 2009, pp. 670–677. DOI: `10.1109/SERVICES-I.2009.32` (p. 183).

[9] S. Apel, T. Leich, and G. Saake. “Aspectual Feature Modules”. In: *IEEE Transactions on Software Engineering* 34.2 (Mar. 2008), pp. 162–180. DOI: `10.1109/TSE.2007.70770` (p. 10).

[10] Apprenda, Inc. *Apprenda - Enterprise PaaS leader*. `http://apprenda.com/platform/`. [Last visited on April 1, 2014] (pp. 66, 109).

[11] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. *Above the clouds: A berkeley view of cloud computing.* Tech. Rep. UCB/EECS 28. EECS Department, University of California, Berkeley, 2009 (pp. 26, 28, 72).

[12] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. "A view of cloud computing". In: *Commun. ACM* 53.4 (2010), pp. 50–58. DOI: `10.1145/1721654.1721672` (pp. 1–5, 187).

[13] K. Ashton. "That 'Internet of Things' thing". In: *RFiD Journal* (2009) (pp. 186, 187).

[14] L. Atzori, A. Iera, and G. Morabito. "The Internet of Things: A survey". In: *Computer Networks* 54.15 (2010), pp. 2787–2805. DOI: `10.1016/j.comnet.2010.05.010` (pp. 186, 187).

[15] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. "Multi-tenant databases for Software as a Service: Schema-mapping techniques". In: *SIGMOD '08: Proceedings of the ACM SIGMOD international conference on Management of Data.* ACM, 2008, pp. 1195–1206. DOI: `10.1145/1376616.1376736` (p. 73).

[16] S. Aulbach, D. Jacobs, A. Kemper, and M. Seibold. "A comparison of flexible schemas for Software as a Service". In: *SIGMOD '09: Proceedings of the ACM SIGMOD International Conference on Management of Data.* ACM, 2009, pp. 881–888. DOI: `10.1145/1559845.1559941` (p. 73).

[17] A. Azeez, S. Perera, D. Gamage, R. Linton, P. Siriwardana, D. Leelaratne, S. Weerawarana, and P. Fremantle. "Multi-tenant SOA middleware for cloud computing". In: *CLOUD '10: IEEE International Conference on Cloud Computing.* IEEE Computer Society, 2010, pp. 458–465. DOI: `10.1109/CLOUD.2010.50` (pp. 27, 93).

[18] 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". In: *PFE '04: Software Product-Family Engineering.* Springer Berlin / Heidelberg, 2004, pp. 66–80. DOI: `10.1007/978-3-540-24667-1_6` (pp. 153, 175).

[19] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. M. Léon, Y. Li, A. Lloyd, and V. Yushprakh. "Megastore: Providing scalable, highly available storage for interactive services". In: *CIDR '11: Proceedings on Conference on Innovative Data Systems Research.* 2011 (pp. 29, 38, 39).

[20] J. Baliga, R. W. A. Ayre, K. Hinton, and R. Tucker. "Green cloud computing: Balancing energy in processing, storage, and transport". In: *Proceedings of the IEEE* 99.1 (Jan. 2011), pp. 149–167. DOI: `10.1109/JPROC.2010.2060451` (p. 184).

[21] L. Baresi, S. Guinea, and L. Pasquale. "Service-oriented dynamic software product lines". In: *Computer* 45.10 (2012), pp. 42–48. DOI: 10.1109/MC.2012.289 (p. 173).

[22] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. 2nd. Addison-Wesley Professional, 2003 (p. 148).

[23] F. Bergmans. "Middleware support for product line development of multi-tenant SaaS applications". MA thesis. Department of Computer Science, KU Leuven, 2012 (pp. 85, 87).

[24] C.-P. Bezemer and A. Zaidman. *Challenges of reengineering into multi-tenant SaaS applications*. Software Engineering Research Group (SERG) 12. TU Delft, 2010 (p. 74).

[25] C.-P. Bezemer, A. Zaidman, B. Platzbeecker, T. Hurkmans, and A. Hart. "Enabling multi-tenancy: An industrial experience report". In: *ICSM '10: Proceedings of the 26th International Conference on Software Maintenance*. 2010, pp. 1–8. DOI: 10.1109/ICSM.2010.5609735 (pp. 20, 74, 109, 154).

[26] L. N. Bhuyan and D. P. Agrawal. "Generalized hypercube and hyperbus structures for a computer network". In: *IEEE Transactions on Computers* C-33.4 (1984), pp. 323–333. DOI: 10.1109/TC.1984.1676437 (p. 16).

[27] L. F. Bittencourt and E. R. M. Madeira. "HCOC: A cost optimization algorithm for workflow scheduling in hybrid clouds". In: *Journal of Internet Services and Applications* 2.3 (2011), pp. 207–227. DOI: 10.1007/s13174-011-0032-0 (p. 139).

[28] M. Bjorkqvist, L. Y. Chen, and W. Binder. "Cost-driven service provisioning in hybrid clouds". In: *SOCA '12: 5th IEEE International Conference on service-oriented computing and applications*. 2012, pp. 1–8. DOI: 10.1109/SOCA.2012.6449447 (p. 139).

[29] G. Blair and P. Grace. "Emergent middleware: Tackling the interoperability problem". In: *IEEE Internet Computing* 16.1 (Jan. 2012), pp. 78–82. DOI: 10.1109/MIC.2012.7 (p. 186).

[30] G. S. Blair, A. Bennaceur, N. Georgantas, P. Grace, V. Issarny, V. Nundloll, and M. Paolucci. "The role of ontologies in emergent middleware: Supporting interoperability in complex distributed systems". In: *Middleware '11: Proceedings of the 12th International Middleware Conference*. Springer Berlin / Heidelberg, Dec. 2011, pp. 410–430. DOI: 10.1007/978-3-642-25821-3_21 (p. 183).

[31] G. S. Blair, G. Coulson, P. Robin, and M. Papathomas. "An architecture for next generation middleware". In: *Middleware '98: Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer-Verlag, 1998, pp. 191–206 (pp. 9, 109).

[32] J. Bozman and G. Chen. "Cloud computing: The need for portability and interoperability". In: *IDC Analyze the Future* (Aug. 2010). Sponsored by Red Hat, Inc. (pp. 17, 19).

[33] R. Buyya, S. K. Garg, and R. N. Calheiros. "SLA-oriented resource provisioning for cloud computing: Challenges, architecture, and solutions". In: *CSC '11: International Conference on Cloud and Service Computing.* 2011, pp. 1–10. DOI: `10.1109/CSC.2011.6138522` (p. 139).

[34] H. Cai, N. Wang, and M. J. Zhou. "A transparent approach of enabling SaaS multi-tenancy in the cloud". In: *SERVICES-1 '10: 6th World Congress on Services.* July 2010, pp. 40–47. DOI: `10.1109/SERVICES.2010.48` (pp. 93, 108, 120).

[35] H. Cai, K. Zhang, M. J. Zhou, W. Gong, J. J. Cai, and X. S. Mao. "An End-to-End Methodology and Toolkit for Fine Granularity SaaS-ization". In: *CLOUD '09: IEEE International Conference on Cloud Computing.* 2009, pp. 101–108. DOI: `10.1109/CLOUD.2009.63` (p. 179).

[36] R. N. Calheiros, C. Vecchiola, D. Karunamoorthy, and R. Buyya. "The Aneka platform and QoS-driven resource provisioning for elastic applications on hybrid clouds". In: *Future Generation Computer Systems* 28.6 (2012), pp. 861–870. DOI: `10.1016/j.future.2011.07.005` (p. 139).

[37] P. G. Carlock and R. E. Fenton. "System of Systems (SoS) enterprise systems engineering for information-intensive organizations". In: *Systems Engineering* 4.4 (2001), pp. 242–261. DOI: `10.1002/sys.1021` (p. 187).

[38] Y. C. Cavalcanti, I. do Carmo Machado, P. A. da Mota, S. Neto, L. L. Lobato, E. S. de Almeida, and S. R. de Lemos Meira. "Towards metamodel support for variability and traceability in software product lines". In: *VaMoS '11: Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems.* ACM, 2011, pp. 49–57. DOI: `10.1145/1944892.1944898` (pp. 153, 175).

[39] M. Chalkiadaki and K. Magoutis. "Managing service performance in NoSQL distributed storage systems". In: *MW4NG '12: Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing.* ACM, 2012, pp. 5–1. DOI: `10.1145/2405178.2405183` (p. 138).

[40] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. "Bigtable: A distributed storage system for structured data". In: *ACM Trans. Comput. Syst.* 26.2 (2008), pp. 4–1. DOI: `10.1145/1365815.1365816` (pp. 29, 38, 39).

[41] D. Chappell. *Introducing the Windows Azure platform.* `http://www.davidchappell.com/writing/white_papers/Introducing_the_Windows_Azure_Platform,_v1.4--Chappell.pdf`. Sponsored by Microsoft Corporation. Oct. 2010 (pp. 29, 30).

[42] P. P.-S. Chen. "The entity-relationship model - Toward a unified view of data". In: *ACM Trans. Database Syst.* 1.1 (Mar. 1976), pp. 9–36. DOI: 10.1145/320434.320440 (p. 33).

[43] N. Chohan, C. Bunch, S. Pang, C. Krintz, N. Mostafa, S. Soman, and R. Wolski. "AppScale: Scalable and open AppEngine application development and deployment". In: *CloudComp '09: First International Conference on Cloud Computing.* Springer Berlin / Heidelberg, 2010, pp. 57–70. DOI: 10.1007/978-3-642-12636-9_4 (pp. 27, 109).

[44] F. Chong and G. Carraro. *Architecture strategies for catching the long tail.* http://msdn.microsoft.com/en-us/library/aa479069.aspx. Microsoft Corporation. Apr. 2006 (pp. 1, 3, 7, 15, 17, 20, 45, 52, 74, 91, 103, 109, 115, 138, 143).

[45] F. Chong, G. Carraro, and R. Wolter. *Multi-tenant data architecture.* http://msdn.microsoft.com/en-us/library/aa479086.aspx. Microsoft Corporation. June 2006 (pp. 55, 73).

[46] Citrix Systems, Inc. *CloudStack - Open source cloud computing project.* http://cloudstack.org. [Last visited on April 1, 2014] (p. 72).

[47] A. Classen, Q. Boucher, and P. Heymans. "A text-based approach to feature modelling: Syntax and semantics of TVL". In: *Science of Computer Programming* 76.12 (2011). Special Issue on Software Evolution, Adaptability and Variability, pp. 1130–1143. DOI: 10.1016/j.scico.2010.10.005 (p. 151).

[48] P. Clements and L. M. Northrop. *Software product lines: Practices and patterns.* 3rd. Addison-Wesley Professional, 2001 (pp. 9, 144).

[49] CloudBees, Inc. *CloudBees: The Java PaaS company.* http://www.cloudbees.com. [Last visited on April 1, 2014] (p. 27).

[50] S. Cohen and R. Krut, eds. *Proceedings of the 1st Workshop on Service-oriented Architectures and Software Product Lines (SOAPL).* Carnegie Mellon University - Software Engineering Institute, May 2008 (p. 172).

[51] P. Costanza, R. Hirschfeld, and W. Meuter. "Efficient layer activation for switching context-dependent behavior". In: *JMLC '06: Proceedings of the 7th Joint Modular Languages Conference.* Springer Berlin / Heidelberg, 2006, pp. 84–103. DOI: 10.1007/11860990_7 (p. 10).

[52] G. Coulson, G. S. Blair, M. Clarke, and N. Parlavantzas. "The design of a configurable and reconfigurable middleware platform". In: *Distributed Computing* 15.2 (2002), pp. 109–126. DOI: 10.1007/s004460100064 (pp. 9, 109, 110).

[53] CUSTOMSS. *CUSTOMization of Software Services in the cloud (iMinds ICON project).* http://www.iminds.be/en/research/overview-projects/p/detail/customss. [Last visited on April 1, 2014]. 2011 (pp. 87, 144, 145, 158, 179).

[54] K. Czarnecki, C. Hwan, P. Kim, and K. T. Kalleberg. “Feature models are views on ontologies”. In: *SPLC ’06: 10th International Software Product Line Conference.* 2006, pp. 41–51 (p. 183).

[55] K. Czarnecki, S. Helsen, and U. Eisenecker. “Staged configuration through specialization and multilevel configuration of feature models”. In: *Software Process: Improvement and Practice* 10.2 (2005), pp. 143–169. DOI: `10.1002/spip.225` (p. 174).

[56] S. M. Davis. “From “future perfect”: Mass customizing”. In: *Strategy & Leadership* 17.2 (1989), pp. 16–21. DOI: `10.1108/eb054249` (p. 7).

[57] L. DeMichiel. *JSR 317: Java^{TM} Persistence 2.0.* `http://www.jcp.org/en/jsr/detail?id=317`. [Last visited on April 1, 2014]. Dec. 2009 (p. 24).

[58] L. DeMichiel and B. Shannon. *JSR 342: Java^{TM} Platform, Enterprise Edition 7 (Java EE 7) Specification.* `http://www.jcp.org/en/jsr/detail?id=342`. [Last visited on April 1, 2014]. May 2013 (pp. 71, 109).

[59] T. Desair, W. Joosen, B. Lagaisse, A. Rafique, and S. Walraven. “Policy-driven middleware for heterogeneous, hybrid cloud platforms”. In: *ARM ’13: Proceedings of the 12th International Workshop on Adaptive and Reflective Middleware.* ACM, Dec. 2013, pp. 7–12. DOI: `10.1145/2541583.2541585` (p. 183).

[60] A. van Deursen and P. Klint. “Domain-specific language design requires feature descriptions”. In: *Journal of Computing and Information Technology* 10.1 (2002), pp. 1–17 (p. 151).

[61] T. Dumitraş and P. Narasimhan. “Why do upgrades fail and what can we do about it?” In: *Middleware ’09: 10th ACM/IFIP/USENIX International Conference on Middleware.* Springer Berlin / Heidelberg, 2009, pp. 349–372. DOI: `10.1007/978-3-642-10445-9_18` (p. 154).

[62] L. Ellison. *What the hell Is cloud computing?* `https://www.youtube.com/watch?v=0FacYAI6DY0`. [Last visited on April 1, 2014]. Sept. 2008 (p. 2).

[63] N. Engelsen. *Multi-tenant architecture via dependency injection.* `http://blog.tallan.com/2010/07/11/multi-tenant-architecture-via-dependency-injection-part-1/`. [Last visited on April 1, 2014]. July 2010 (p. 56).

[64] A. Etien and C. Salinesi. “Managing requirements in a co-evolution context”. In: *Proceedings of the 13th IEEE International Conference on Requirements Engineering.* IEEE, 2005, pp. 125–134. DOI: `10.1109/RE.2005.37` (p. 150).

[65] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit. *Aspect-oriented software development.* 1st. Addison-Wesley Professional, 2004 (p. 110).

[66] M. Fowler. *Inversion of control containers and the dependency injection pattern.* http://martinfowler.com/articles/injection.html. Jan. 2004 (pp. 50, 85, 96, 99, 155, 181).

[67] M. Galster. "Describing variability in service-oriented software product lines". In: *ECSA '10: Proceedings of the 4th European Conference on Software Architecture: Companion Volume.* ACM, 2010, pp. 344–350. DOI: 10.1145/1842752.1842815 (pp. 172, 174).

[68] M. Galster and P. Avgeriou. "A variability viewpoint for enterprise software systems". In: *WICSA/ECSA '12: Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture.* IEEE, 2012, pp. 267–271. DOI: 10.1109/WICSA-ECSA.212.43 (pp. 152, 174, 175).

[69] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented software.* 1st. Addison-Wesley, 1994 (p. 127).

[70] K. Garen. "Software portability: Weighing options, making choices". In: *The CPA Journal* 77.11 (2007), pp. 10–12 (p. 19).

[71] S. K. Garg and R. Buyya. "Harnessing green IT: Principles and practices". In: ed. by S. Murugesan and G. R. Gangadharan. 1st ed. Wiley - IEEE Press, 2012. Chap. Green cloud computing and environmental sustainability, pp. 315–339. DOI: 10.1002/9781118305393.ch16 (p. 184).

[72] Gartner, Inc. *Gartner says worldwide public cloud services market to total $131 billion.* http://www.gartner.com/newsroom/id/2352816. Feb. 2013 (p. 184).

[73] K. Geebelen, S. Walraven, E. Truyen, S. Michiels, H. Moens, F. De Turck, B. Dhoedt, and W. Joosen. "An open middleware for proactive QoS-aware service composition in a multi-tenant SaaS environment". In: *ICOMP '12: Proceedings of the 2012 International Conference on Internet Computing.* CSREA Press, July 2012 (p. 89).

[74] F. Gey, S. Walraven, D. Landuyt, and W. Joosen. "Building a customizable Business-Process-as-a-Service application with current state-of-practice". In: *SC '13: 12th International Conference on Software Composition.* Springer Berlin / Heidelberg, June 2013, pp. 113–127. DOI: 10.1007/978-3-642-39614-4_8 (pp. 77, 85, 87, 89, 166).

[75] A. Giessmann and K. Stanoevska-Slabeva. "Platform as a Service : A conjoint study on consumers' preferences". In: *ICIS '12: 33rd International Conference on Information Systems.* Association for Information Systems (AIS), 2012 (pp. 17, 21).

[76] GigaSpaces Technologies, Inc. *XAP - In-memory computing platform.* http://www.gigaspaces.com/xap/. [Last visited on April 1, 2014] (pp. 17, 27, 66, 109).

[77] Google, Inc. *Google App Engine*. http://developers.google.com/appengine/. [Last visited on April 1, 2014] (pp. 5, 15, 17, 26–28, 36, 37, 49, 72, 73, 81, 92, 96, 100, 108, 121, 179).

[78] Google, Inc. *Guice*. http://code.google.com/p/google-guice/. [Last visited on April 1, 2014] (pp. 29, 50, 92, 100).

[79] Google, Inc. *The JRE class white list*. http://developers.google.com/appengine/docs/java/jrewhitelist.html. [Last visited on April 1, 2014] (p. 37).

[80] Google, Inc. *Will it play in Java*. http://code.google.com/p/googleappengine/wiki/WillItPlayInJava. [Last visited on April 1, 2014] (p. 38).

[81] T. B. Gooley. "Mass customization: How logistics makes it happen". In: *Logistics Management & Distribution Report* 37.4 (1998), pp. 49–52 (p. 7).

[82] I. Groher and R. Weinreich. "Integrating variability management and software architecture". In: *WICSA/ECSA '12: Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*. IEEE, 2012, pp. 262–266. DOI: 10.1109/WICSA-ECSA.212.42 (pp. 152, 153, 175).

[83] S. Günther and T. Berger. "Service-oriented product lines: Towards a development process and feature management model for web services". In: *SOAPL '08: Workhop on Service-oriented Architectures and Software Product Lines*. 2008, pp. 131–136 (p. 172).

[84] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, and B. Gao. "A framework for native multi-tenancy application development and management". In: *CEC/EEE '07: 9th IEEE International Conference on E-Commerce Technology and 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services*. July 2007, pp. 551–558. DOI: 10.1109/CEC-EEE.2007.4 (pp. 3, 7, 8, 15, 17, 20, 45, 70, 74, 91, 108, 109, 115, 138, 143, 154).

[85] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. "Enforcing performance isolation across virtual machines in Xen". In: *Middleware '06: Proceedings of the ACM/IFIP/USENIX International Conference on Middleware*. Springer Berlin / Heidelberg, 2006, pp. 342–362. DOI: 10.1007/11925071_18 (p. 138).

[86] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. "Dynamic software product lines". In: *Computer* 41.4 (Apr. 2008), pp. 93–95. DOI: 10.1109/MC.2008.123 (p. 173).

[87] S. Hallsteinsen, S. Jiang, and R. Sanders. "Dynamic software product lines in service oriented computing". In: *DSPL '09: Proceedings of the 3rd International Workshop on Dynamic Software Product Lines*. 2009, pp. 28–34 (p. 173).

[88] Ø. Haugen and et al. *Common Variability Language (CVL)*. http://www.omgwiki.org/variability/. OMG Revised Submission. Aug. 2012 (p. 151).

[89] Heroku, Inc. *Heroku cloud application platform.* `http://www.heroku.com/`. [Last visited on April 1, 2014] (p. 27).

[90] S. Hillyer. *An introduction to environments.* `http://wiki.developerforce.com/index.php/An_Introduction_to_Environments`. [Last visited on April 1, 2014] (p. 45).

[91] S. Hiranandani, K. Kennedy, and C.-W. Tseng. "Compiling Fortran D for MIMD distributed-memory machines". In: *Commun. ACM* 35.8 (1992), pp. 66–80. DOI: `10.1145/135226.135230` (p. 16).

[92] R. Hirschfeld, P. Costanza, and O. Nierstrasz. "Context-oriented programming". In: *Journal of Object Technology* 7.3 (Mar. 2008), pp. 125–151. DOI: `10.5381/jot.2008.7.3.a4` (pp. 85, 181).

[93] A. Hubaux, P. Heymans, P.-Y. Schobbens, D. Deridder, and E. Abbasi. "Supporting multiple perspectives in feature-based configuration". In: *Software and Systems Modeling* (2011), pp. 1–23. DOI: `10.1007/s10270-011-0220-1` (p. 175).

[94] IBM Corporation. *IBM SmartCloud application services.* `http://www.ibm.com/cloud-computing/us/en/paas.html`. [Last visited on April 1, 2014] (p. 27).

[95] IDC. *IDC forecasts worldwide public IT cloud services spending to reach nearly $108 billion by 2017 as focus shifts from savings to innovation.* `http://www.idc.com/getdoc.jsp?containerId=prUS24298013`. Sept. 2013 (p. 184).

[96] JBoss Community. *JBoss AOP.* `http://www.jboss.org/jbossaop/`. [Last visited on April 1, 2014] (p. 110).

[97] J. Kabbedijk and S. Jansen. "Variability in multi-tenant environments: Architectural design patterns from industry". In: *Advances in Conceptual Modeling. Recent Developments and New Directions.* Springer Berlin / Heidelberg, 2011, pp. 151–160. DOI: `10.1007/978-3-642-24574-9_20` (p. 174).

[98] J. Kabbedijk, T. Salfischberger, and S. Jansen. "Comparing two architectural patterns for dynamically adapting functionality in online software products". In: *PATTERNS '13: 5th International Conferences on Pervasive Patterns and Applications.* 2013, pp. 20–25 (p. 174).

[99] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. "FORM: A feature-oriented reuse method with domain-specific reference architectures". In: *Annals of Software Engineering* 5.1 (1998), pp. 143–168 (p. 151).

[100] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. *Feature-oriented domain analysis (FODA) feasibility study.* Tech. rep. 21. Software Engineering Institute, Carnegie Mellon University, 1990 (pp. 10, 96, 151).

[101] C. Kastner, T. Thum, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. "FeatureIDE: A tool framework for feature-oriented software development". In: *ICSE '09: 31st IEEE International Conference on Software Engineering.* May 2009, pp. 611–614. DOI: 10.1109/ICSE.2009.5070568 (pp. 151, 155, 160, 161).

[102] N. Kohari. *Ninject! - Open source dependency injector for .NET.* http://www.ninject.org/. [Last visited on April 1, 2014] (pp. 31, 55).

[103] F. Kon, F. Costa, G. Blair, and R. H. Campbell. "The case for reflective middleware". In: *Commun. ACM* 45.6 (2002), pp. 33–38. DOI: 10.1145/508448.508470 (pp. 9, 109).

[104] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, C. Magalhães, and R. H. Campbell. "Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB". In: *Middleware '00: IFIP/ACM International Conference on Distributed systems platforms.* Springer-Verlag New York, Inc., 2000, pp. 121–143 (pp. 9, 110).

[105] G. Kotonya, J. Lee, and D. Robinson. "A consumer-centred approach for service-oriented product line development". In: *WICSA/ECSA '09: Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture.* IEEE, 2009, pp. 211–220. DOI: 10.1109/WICSA.2009.5290807 (pp. 171, 173).

[106] V. Kotov. *Systems of Systems as communicating structures.* HPL 124. Hewlett Packard Computer Systems Laboratory, Oct. 1997 (p. 187).

[107] R. Krebs, C. Momm, and S. Kounev. "Metrics and techniques for quantifying performance isolation in cloud environments". In: *QoSA '12: Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures.* ACM, 2012, pp. 91–100. DOI: 10.1145/2304696.2304713 (pp. 115, 136).

[108] P. B. Kruchten. "The 4+1 view model of architecture". In: *IEEE Software* 12.6 (1995), pp. 42–50. DOI: 10.1109/52.469759 (p. 152).

[109] P. Kurp. "Green computing". In: *Commun. ACM* 51.10 (Oct. 2008), pp. 11–13. DOI: 10.1145/1400181.1400186 (p. 184).

[110] T. Kwok and A. Mohindra. "Resource calculations with constraints, and placement of tenants and instances for multi-tenant SaaS applications". In: *ICSOC '08: 6th International Conference on Service-oriented Computing.* Springer Berlin / Heidelberg, 2008, pp. 633–648. DOI: 10.1007/978-3-540-89652-4_57 (pp. 139, 179).

[111] B. Lagaisse and W. Joosen. "True and transparent distributed composition of aspect-components". In: *Middleware '06: Proceedings of the 7th ACM/IFIP/USENIX International Conference on Middleware.* Springer Berlin / Heidelberg, Nov. 2006, pp. 41–62. DOI: 10.1007/11925071_3 (pp. 109, 110).

[112] A. Lakshman and P. Malik. "Cassandra: A decentralized structured storage system". In: *SIGOPS Oper. Syst. Rev.* 44.2 (2010), pp. 35–40. DOI: 10.1145/1773912.1773922 (p. 129).

[113] N. Leavitt. "Hybrid clouds move to the forefront". In: *Computer* 46.5 (2013), pp. 15–18. DOI: 10.1109/MC.2013.168 (p. 6).

[114] J. Lee and G. Kotonya. "Combining service-orientation with product line engineering". In: *IEEE Software* 27.3 (2010), pp. 35–41. DOI: 10.1109/MS.2010.30 (pp. 171, 173).

[115] P. Leitner, B. Satzger, W. Hummer, C. Inzinger, and S. Dustdar. "CloudScale: A novel middleware for building transparently scaling cloud applications". In: *SAC '12: Proceedings of the 27th Annual ACM Symposium on Applied Computing*. ACM, 2012, pp. 434–440. DOI: 10.1145/2245276.2245360 (pp. 117, 139).

[116] P. Leitner, B. Wetzstein, F. Rosenberg, A. Michlmayr, S. Dustdar, and F. Leymann. "Runtime prediction of service level agreement violations for composite services". In: *ICSOC/ServiceWave '09: Service-oriented Computing Workshops*. Springer Berlin / Heidelberg, 2010, pp. 176–186. DOI: 10.1007/978-3-642-16132-2_17 (p. 139).

[117] X. Li, T. Liu, Y. Li, and Y. Chen. "SPIN: Service performance isolation infrastructure in multi-tenancy environment". In: *ICSOC '08: Service-oriented Computing*. Springer Berlin / Heidelberg, 2008, pp. 649–663. DOI: 10.1007/978-3-540-89652-4_58 (pp. 138, 182).

[118] R. E. Lopez-Herrejon, D. Batory, and W. Cook. "Evaluating support for features in advanced modularization technologies". In: *ECOOP '05: Proceedings of the 19th European Conference on Object-Oriented Programming*. Springer Berlin / Heidelberg, 2005, pp. 169–194. DOI: 10.1007/11531142_8 (p. 10).

[119] D. Lorenzoli and G. Spanoudakis. "EVEREST+: Run-time SLA violations prediction". In: *MW4SOC '10: Proceedings of the 5th International Workshop on Middleware for Service Oriented Computing*. ACM, 2010, pp. 13–18. DOI: 10.1145/1890912.1890915 (p. 139).

[120] W. Lu, J. Jackson, and R. Barga. "AzureBlast: A case study of developing science applications on the cloud". In: *HPDC '10: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, 2010, pp. 413–420. DOI: 10.1145/1851476.1851537 (p. 117).

[121] X. Ma, L. Baresi, C. Ghezzi, V. Panzica La Manna, and J. Lu. "Version-consistent Dynamic Reconfiguration of Component-based Distributed Systems". In: *ESEC/FSE '11: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ACM, 2011, pp. 245–255. DOI: 10.1145/2025113.2025148 (p. 83).

[122] N. B. Mabrouk, S. Beauche, E. Kuznetsova, N. Georgantas, and V. Issarny. "QoS-aware service composition in dynamic service oriented environments". In: *Middleware '09: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*. Springer-Verlag New York, Inc., 2009, pp. 123–142 (p. 139).

[123] P. Maes. "Concepts and experiments in computational reflection". In: *OOPSLA '87: Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*. ACM, 1987, pp. 147–155. DOI: `10.1145/38765.38821` (p. 181).

[124] M. W. Maier. "Architecting principles for systems-of-systems". In: *Systems Engineering* 1.4 (1998), pp. 267–284. DOI: `10.1002/(SICI)1520-6858(1998)1:4<267::AID-SYS3>3.0.CO;2-D` (p. 187).

[125] M. Mao and M. Humphrey. "A performance study on the VM startup time in the cloud". In: *CLOUD '12: IEEE 5th International Conference on Cloud Computing*. 2012, pp. 423–430. DOI: `10.1109/CLOUD.2012.103` (p. 120).

[126] M. Mao and M. Humphrey. "Auto-scaling to minimize cost and meet application deadlines in cloud workflows". In: *SC '11: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2011, pp. 1–12 (pp. 117, 139).

[127] MAPC. *A next generation multi-party solution for Medical APplications in the Cloud (IWT project)*. `https://distrinet.cs.kuleuven.be/research/projects/showProject.do?projectID=MAPC`. [Last visited on April 1, 2014]. 2013 (p. 181).

[128] F. M. Medeiros, E. S. de Almeida, and S. R. de Lemos Meira. "Towards an approach for service-oriented product line architectures". In: *SOAPL '09: Proceedings of the 3rd Workshop on Service-Oriented Architectures and Software Product Lines*. 2009, pp. 151–164 (p. 172).

[129] H. Mei, G. Huang, and T. Xie. "Internetware: A software paradigm for Internet computing". In: *Computer* 45.6 (2012), pp. 26–31. DOI: `10.1109/MC.2012.189` (p. 186).

[130] P. Mell and T. Grance. *The NIST definition of cloud computing*. Special Publication 800-145. `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`. National Institute of Standards and Technology (NIST), Sept. 2011 (pp. 2, 4, 5, 15, 16).

[131] M. Mendonca, M. Branco, and D. Cowan. "S.P.L.O.T.: Software product lines online tools". In: *OOPSLA '09: Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. ACM, 2009, pp. 761–762. DOI: `10.1145/1639950.1640002` (pp. 151, 155).

[132] Microsoft Corp. *ADO.NET Entity Framework*. `http://msdn.microsoft.com/en-us/data/aa937723`. [Last visited on April 1, 2014] (p. 24).

[133] Microsoft Corp. *ASP.NET MVC*. `http://www.asp.net/mvc`. [Last visited on April 1, 2014] (pp. 24, 40).

[134] Microsoft Corp. *Windows Azure*. `http://www.windowsazure.com/`. [Last visited on April 1, 2014] (pp. 5, 17, 26, 27, 29, 36, 72, 73, 117).

[135] Microsoft Corp. *Windows Azure AppFabric*. `http://www.microsoft.com/windowsazure/AppFabric`. [Last visited on April 1, 2014] (pp. 30, 42, 44).

[136] Microsoft Corp. *Windows Azure Architecture Guide*. `http://wag.codeplex.com/`. [Last visited on April 1, 2014] (pp. 43, 52, 54, 63, 68).

[137] Microsoft Corp. *Windows Communication Foundation (WCF)*. `http://msdn.microsoft.com/en-us/library/dd456779%28v=vs.110%29.aspx`. [Last visited on April 1, 2014] (pp. 40, 42, 69).

[138] R. Mietzner and F. Leymann. “Generation of BPEL customization processes for SaaS applications from variability descriptors”. In: *SCC ’08: IEEE International Conference on Services Computing*. Vol. 2. 2008, pp. 359–366. DOI: `10.1109/SCC.2008.85` (pp. 144, 155, 173).

[139] R. Mietzner, F. Leymann, and M. P. Papazoglou. “Defining composite configurable SaaS application packages using SCA, variability descriptors and multi-tenancy patterns”. In: *ICIW ’08: 3rd International Conference on Internet and Web Applications and Services*. June 2008, pp. 156–161. DOI: `10.1109/ICIW.2008.68` (p. 75).

[140] R. Mietzner, F. Leymann, and T. Unger. “Horizontal and vertical combination of multi-tenancy patterns in service-oriented applications”. In: *Enterprise Information Systems* 5.1 (2011), pp. 59–77. DOI: `10.1080/17517575.2010.492950` (p. 174).

[141] 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: *PESOS ’09: ICSE Workshop on Principles of Engineering Service Oriented Systems*. IEEE Computer Society, 2009, pp. 18–25. DOI: `10.1109/PESOS.2009.5068815` (pp. 109, 144, 173).

[142] C. Momm and W. Theilmann. “A combined workload planning approach for multi-tenant business applications”. In: *COMPSACW ’11: IEEE 35th Annual Computer Software and Applications Conference Workshops*. IEEE, July 2011, pp. 255–260. DOI: `10.1109/COMPSACW.2011.96` (p. 136).

[143] T. Monheim. “Middleware support for performance isolation in multi-tenant SaaS applications”. MA thesis. Departement of Computer Science, KU Leuven, 2012 (pp. 85, 87).

[144] J. D. Mooney. “Strategies for supporting application portability”. In: *Computer* 23.11 (Nov. 1990), pp. 59–70. DOI: `10.1109/2.60881` (p. 19).

[145] R. Mordani. *JSR 315: Java^{TM} Servlet 3.0 Specification.* `http://www.jcp.org/en/jsr/detail?id=315`. [Last visited on April 1, 2014]. 2011 (p. 24).

[146] D. Mueller. *It's 2014 and PaaS is eating the world.* `https://www.openshift.com/blogs/its-2014-and-paas-is-eating-the-world`. Feb. 2014 (p. 13).

[147] J. Müller, J. Krüger, S. Enderlein, M. Helmich, and A. Zeier. "Customizing enterprise Software as a Service applications: Back-end extension in a multi-tenancy environment". In: *ICEIS '09: 11th International Conference on Enterprise Information Systems.* Springer Berlin / Heidelberg, May 2009, pp. 66–77. DOI: `10.1007/978-3-642-01347-8_6` (p. 109).

[148] Y. V. Natis, D. M. Smith, and D. W. Cearley. *Windows Azure AppFabric: A strategic core of Microsoft's cloud platform.* `http://www.gartner.com/id=1469531`. Nov. 2010 (pp. 30, 44).

[149] T. Nguyen, A. Colman, and J. Han. "Enabling the delivery of customizable web services". In: *ICWS '12: 19th IEEE International Conference on Web Services.* 2012, pp. 138–145. DOI: `10.1109/ICWS.2012.23` (p. 172).

[150] T. Nguyen, A. Colman, M. A. Talib, and J. Han. "Managing service variability: State of the art and open issues". In: *VaMoS '11: Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems.* ACM, 2011, pp. 165–173. DOI: `10.1145/1944892.1944913` (p. 174).

[151] A. Nicoară and G. Alonso. "Dynamic AOP with PROSE". In: *ASMEA '05: Proceedings of 1st International Workshop on Adaptive and Self-Managing Enterprise Applications.* 2005, pp. 125–138 (p. 110).

[152] Noelios Technologies. *Restlet framework.* `http://www.restlet.org/`. [Last visited on April 1, 2014] (pp. 37, 40).

[153] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. "The Eucalyptus open-source cloud-computing system". In: *CCGRID '09: Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid.* IEEE Computer Society, 2009, pp. 124–131. DOI: `10.1109/CCGRID.2009.93` (pp. 16, 72).

[154] B. Nuseibeh. "Weaving together requirements and architectures". In: *Computer* 34.3 (2001), pp. 115–119. DOI: `10.1109/2.910904` (p. 150).

[155] Open Grid Forum (OGF). *Occi: Open cloud computing interface.* `http://occi-wg.org/`. [Last visited on April 1, 2014] (p. 72).

[156] OpenStack, LLC. *OpenStack - Open source software for building private and public clouds.* `http://www.openstack.org/`. [Last visited on April 1, 2014] (pp. 7, 72).

[157] OpenText Corp. *OpenText Cordys.* `http://www.cordys.com/platform-as-a-service`. [Last visited on April 1, 2014] (p. 27).

[158] Oracle Corporation. *Jersey - RESTful web services in Java*. `http://jersey.java.net/`. [Last visited on April 1, 2014] (p. 40).

[159] Oracle Corporation. *Oracle Cloud Platform as a Service (PaaS)*. `https://cloud.oracle.com/build_apps`. [Last visited on April 1, 2014] (pp. 17, 27).

[160] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. "Adaptive control of virtualized resources in utility computing environments". In: *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*. ACM, 2007, pp. 289–302. DOI: `10.1145/1272996.1273026` (p. 139).

[161] F. Paraiso, N. Haderer, P. Merle, R. Rouvoy, and L. Seinturier. "A federated multi-cloud PaaS infrastructure". In: *CLOUD '12: IEEE 5th International Conference on Cloud Computing*. June 2012, pp. 392–399. DOI: `10.1109/CLOUD.2012.79` (p. 75).

[162] D. F. Parkhill. *The challenge of the computer utility*. Vol. 2. Addison-Wesley Publishing Company Reading, 1966 (pp. 1, 185, 187).

[163] D. L. Parnas. "On the criteria to be used in decomposing systems into modules". In: *Communications of the ACM* 15.12 (Dec. 1972), pp. 1053–1058. DOI: `10.1145/361598.361623` (p. 10).

[164] C. Parra, X. Blanc, and L. Duchien. "Context awareness for dynamic service-oriented product lines". In: *SPLC '09: 13th International Software Product Line Conference*. 2009, pp. 131–140 (p. 173).

[165] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. "JAC: A flexible solution for aspect-oriented programming in Java". In: *REFLECTION '01: Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*. Springer-Verlag, 2001, pp. 1–24 (pp. 110, 155, 181).

[166] D. Petcu. "Portability and interoperability between clouds: Challenges and case study". In: *ServiceWave '11: Towards a Service-Based Internet*. Springer Berlin / Heidelberg, 2011, pp. 62–74. DOI: `10.1007/978-3-642-24755-2_6` (pp. 17, 19, 75, 183).

[167] PLOPSA. *Product Line Oriented Payment Software Architecture (IWT project)*. `https://distrinet.cs.kuleuven.be/research/projects/showProject.do?projectID=PLOPSA`. [Last visited on April 1, 2014]. 2012 (p. 181).

[168] K. Pohl, G. Böckle, and F. Van Der Linden. *Software product line engineering: Foundations, principles, and techniques*. Springer-Verlag New York Inc., 2005 (pp. 9, 109, 144, 150).

[169] A. Popovici, G. Alonso, and T. Gross. "Just-in-time aspects: Efficient dynamic weaving for Java". In: *AOSD '03: Proceedings of the 2nd International Conference on Aspect-oriented Software Development*. ACM, 2003, pp. 100–109. DOI: `10.1145/643603.643614` (pp. 155, 181).

[170] M. E. Porter. *Competitive advantage: Creating and sustaining superior performance*. New edition. Free Press, July 2004 (p. 184).

[171] C. Prehofer. "Feature-oriented programming: A fresh look at objects". In: *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming*. Springer Berlin / Heidelberg, 1997, pp. 419–443. DOI: `10.1007/BFb0053389` (p. 10).

[172] R. Prodan and S. Ostermann. "A survey and taxonomy of Infrastructure as a Service and web hosting cloud providers". In: *GRID '09: 10th IEEE/ACM International Conference on Grid Computing*. Oct. 2009, pp. 17–25. DOI: `10.1109/GRID.2009.5353074` (p. 72).

[173] A. Rafique, S. Walraven, B. Lagaisse, T. Desair, and W. Joosen. "Towards portability and interoperability support in middleware for hybrid clouds". In: *CrossCloud '14: Proceedings of the 1st IEEE INFOCOM CrossCloud Workshop*. IEEE, 2014 (p. 183).

[174] A. Ranganathan and R. H. Campbell. "A middleware for context-aware agents in ubiquitous computing environments". In: *Middleware '03: Proceedings of the 4th ACM/IFIP/USENIX International Middleware Conference*. Springer Berlin / Heidelberg, 2003, pp. 143–161. DOI: `10.1007/3-540-44892-6_8` (p. 183).

[175] Red Hat, Inc. *Red Hat OpenShift*. `https://www.openshift.com/`. [Last visited on April 1, 2014] (pp. 5, 7, 17, 27, 65, 69, 181).

[176] G. Reese. *Cloud application architectures*. Theory in Practice. O'Reilly, 2009 (p. 73).

[177] B. P. Rimal, E. Choi, and I. Lumb. "A taxonomy and survey of cloud computing systems". In: *NCM '09: 5th International Joint Conference on INC, IMS and IDC*. Aug. 2009, pp. 44–51. DOI: `10.1109/NCM.2009.218` (p. 73).

[178] J. Rosenberg and A. Mateos. *The cloud at your service*. Manning Pubs Co Series. Manning Publications, 2010 (p. 73).

[179] R. Rouvoy, F. Eliassen, and M. Beauvois. "Dynamic planning and weaving of dependability concerns for self-adaptive ubiquitous services". In: *SAC '09: Proceedings of the ACM symposium on Applied Computing*. ACM, 2009, pp. 1021–1028. DOI: `10.1145/1529282.1529507` (p. 110).

[180] R. Rouvoy, F. Eliassen, J. Floch, S. Hallsteinsen, and E. Stav. "Composing components and services using a planning-based adaptation middleware". In: *SC '08: Proceedings of the 7th International Symposium on Software Composition*. Springer, 2008, pp. 52–67. DOI: `10.1007/978-3-540-78789-1_4` (p. 109).

[181] A. P. Sage and C. D. Cuppan. “On the systems engineering and management of Systems of Systems and Federations of Systems”. In: *Information, Knowledge, Systems Management* 2.4 (2001), pp. 325–345 (p. 187).

[182] Salesforce.com, Inc. *Force.com.* `http://www.salesforce.com/platform/`. [Last visited on April 1, 2014] (pp. 15, 17, 26, 27, 73, 185).

[183] Salesforce.com, Inc. *Salesforce CRM.* `http://www.salesforce.com/`. [Last visited on April 1, 2014] (pp. 5, 15, 26, 91, 109).

[184] SAP AG. *SAP cloud platform solutions.* `http://www.sap.com/pc/tech/cloud/software/cloud-platform/index.html`. [Last visited on April 1, 2014] (p. 27).

[185] D. C. Schmidt and F. Buschmann. “Patterns, frameworks, and middleware: Their synergistic relationships”. In: *ICSE ’03: 25th International Conference on Software Engineering.* IEEE, May 2003, pp. 694–704. DOI: `10.1109/ICSE.2003.1201256` (p. 77).

[186] J. Schroeter, S. Cech, S. Götz, C. Wilke, and U. Aßmann. “Towards modeling a variable architecture for multi-tenant SaaS-applications”. In: *VaMoS ’12: Proceedings of the 6th International Workshop on Variability Modeling of Software-Intensive Systems.* ACM, 2012, pp. 111–120. DOI: `10.1145/2110147.2110160` (pp. 144, 174).

[187] J. Schroeter, P. Mucha, M. Muth, K. Jugel, and M. Lochau. “Dynamic configuration management of cloud-based applications”. In: *SPLC ’12: Proceedings of the 16th International Software Product Line Conference - Volume 2.* ACM, 2012, pp. 171–178. DOI: `10.1145/2364412.2364441` (p. 174).

[188] L. Schubert, K. Jeffery, and B. Neidecker-Lutz. *The Future of cloud computing: Opportunities for European cloud computing beyond 2010.* Expert Group Report, Public Version 1.0. European Commission, Information Society and Media, Jan. 2010 (p. 16).

[189] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. “CloudScale: Elastic resource scaling for multi-tenant cloud systems”. In: *SOCC ’11: Proceedings of the 2nd ACM Symposium on Cloud Computing.* ACM, 2011, pp. 1–14. DOI: `10.1145/2038916.2038921` (p. 139).

[190] G. Shroff. “Dev 2.0: Model driven development in the cloud”. In: *SIGSOFT ’08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering.* ACM, 2008, pp. 283–283. DOI: `10.1145/1453101.1453139` (p. 73).

[191] G. Shroff. *Enterprise cloud computing: Technology, architecture, applications.* Cambridge University Press, 2010 (p. 73).

[192] G. Shroff, P. Agarwal, and P. Devanbu. “InstantApps: A WYSIWYG model driven interpreter for web applications”. In: *ICSE-Companion ’09: 31st International Conference on Software Engineering - Companion Volume.* May 2009, pp. 417–418. DOI: `10.1109/ICSE-COMPANION.2009.5071040` (pp. 17, 27, 73).

[193] D. Shue, M. J. Freedman, and A. Shaikh. “Performance isolation and fairness for multi-tenant cloud storage”. In: *OSDI ’12: 10th USENIX Symposium on Operating Systems Design and Implementation.* 2012 (pp. 137, 138, 182).

[194] H. J. Siegel. “Interconnection networks for SIMD machines”. In: *Computer* 12.6 (1979), pp. 57–65. DOI: `10.1109/MC.1979.1658780` (p. 16).

[195] J. P. Singh, W.-D. Weber, and A. Gupta. “SPLASH: Stanford parallel applications for shared-memory”. In: *SIGARCH Comput. Archit. News* 20.1 (1992), pp. 5–44. DOI: `10.1145/130823.130824` (p. 16).

[196] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman. *MPI: The complete reference.* MIT Press, 1995 (p. 16).

[197] G. Somani and S. Chaudhary. “Application performance isolation in virtualization”. In: *CLOUD ’09: IEEE International Conference on Cloud Computing.* IEEE, 2009, pp. 41–48. DOI: `10.1109/CLOUD.2009.78` (p. 138).

[198] B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster. “Virtual infrastructure management in private and hybrid clouds”. In: *IEEE Internet Computing* 13.5 (Sept. 2009), pp. 14–22. DOI: `10.1109/MIC.2009.119` (pp. 16, 72).

[199] SpringSource. *Aspect oriented programming with Spring.* `http://static.springsource.org/spring/docs/4.0.x/spring-framework-reference/html/aop.html`. [Last visited on April 1, 2014]. 2013 (pp. 110, 155).

[200] T. van der Storm. “Generic feature-based software composition”. In: *SC ’07: International Conference on Software Composition.* Springer Berlin / Heidelberg, 2007, pp. 66–80. DOI: `10.1007/978-3-540-77351-1_6` (p. 156).

[201] W. Sun, X. Zhang, C. J. Guo, P. Sun, and H. Su. “Software as a Service: Configuration and customization perspectives”. In: *SERVICES-2 ’08: IEEE Congress on Services Part II.* Sept. 2008, pp. 18–25. DOI: `10.1109/SERVICES-2.2008.29` (pp. 7, 8, 16, 20, 91, 109, 148).

[202] L. Tao. “Shifting paradigms with the application service provider model”. In: *Computer* 34.10 (Oct. 2001), pp. 32–39. DOI: `10.1109/2.955095` (p. 90).

[203] The Apache Software Foundation. *Apache jclouds.* `http://jclouds.apache.org/`. [Last visited on April 1, 2014] (p. 72).

[204] E. Truyen, N. Cardozo, S. Walraven, J. Vallejos, E. Bainomugisha, S. Günther, T. D'Hondt, and W. Joosen. "Context-oriented programming for customizable SaaS applications". In: *SAC '12: Proceedings of the 27th Annual ACM Symposium on Applied Computing*. ACM, Mar. 2012, pp. 418–425. DOI: 10.1145/2245276.2245358 (pp. 77, 85, 87, 181).

[205] E. Truyen, N. Janssens, F. Sanen, and W. Joosen. "Support for distributed adaptations in aspect-oriented middleware". In: *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*. ACM, 2008, pp. 120–131. DOI: 10.1145/1353482.1353497 (p. 110).

[206] E. Truyen, B. Vanhaute, B. N. Jørgensen, W. Joosen, and P. Verbaeten. "Dynamic and selective combination of extensions in component-based applications". In: *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*. IEEE Computer Society, 2001, pp. 233–242 (pp. 9, 110, 155, 181).

[207] N. Tuts. "Support for implementing portable applications on top of PaaS platforms: A case study with Google App Engine". MA thesis. Department of Computer Science, KU Leuven, 2013 (pp. 85, 87).

[208] D. Van Landuyt, S. Op de beeck, E. Truyen, and W. Joosen. "Domain-driven discovery of stable abstractions for pointcut interfaces". In: *AOSD '09: Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development*. ACM, 2009, pp. 75–86. DOI: 10.1145/1509239.1509251 (p. 153).

[209] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. "A break in the clouds: Towards a cloud definition". In: *SIGCOMM Comput. Commun. Rev.* 39.1 (Jan. 2009), pp. 50–55. DOI: 10.1145/1496091.1496100 (p. 15).

[210] M. Vatkina. *JSR 318: Enterprise JavaBeans^{TM} 3.1*. http://jcp.org/en/jsr/detail?id=318. [Last visited on April 1, 2014]. June 2013 (pp. 24, 69).

[211] VMware, Inc. *Cloud Foundry*. http://www.cloudfoundry.org/. [Last visited on April 1, 2014] (pp. 17, 27, 65, 181).

[212] S. Walraven, B. Lagaisse, E. Truyen, and W. Joosen. "Policy-driven customization of cross-organizational features in distributed service systems". In: *Software: Practice & Experience* 43.10 (Oct. 2013 [Oct. 13, 2011]), pp. 1145–1163. DOI: 10.1002/spe.1128 (p. 183).

[213] S. Walraven, T. Monheim, E. Truyen, and W. Joosen. "Towards performance isolation in multi-tenant SaaS applications". In: *MW4NG '12: Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing*. ACM, Dec. 2012, pp. 1–6. DOI: 10.1145/2405178.2405184 (pp. 77, 85, 87, 113, 137, 138).

[214] S. Walraven, E. Truyen, and W. Joosen. "A middleware layer for flexible and cost-efficient multi-tenant applications". In: *Middleware '11: Proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware*. Springer Berlin / Heidelberg, Dec. 2011, pp. 370–389. DOI: 10.1007/978-3-642-25821-3_19 (pp. 12, 15, 20, 70, 77, 87, 89, 115, 120, 138, 143, 144, 155, 158, 165, 173, 174, 178).

[215] S. Walraven, E. Truyen, and W. Joosen. "Comparing PaaS offerings in light of SaaS development. A comparison of PaaS platforms based on a practical case study". In: *Computing* (Oct. 2013), pp. 1–56. DOI: 10.1007/s00607-013-0346-9. Pre-published (pp. 12, 13, 177, 185).

[216] S. Walraven, D. Van Landuyt, F. Gey, and W. Joosen. *Service line engineering in practice: Developing an integrated document processing SaaS application.* CW Reports 652 (2nd revised). https://lirias.kuleuven.be/handle/123456789/428855. Department of Computer Science, KU Leuven, Feb. 2014 (Nov. 2013) (pp. 141, 159, 163, 164, 178, 179).

[217] S. Walraven, D. Van Landuyt, E. Truyen, K. Handekyn, and W. Joosen. "Efficient customization of multi-tenant Software-as-a-Service applications with service lines". In: *Journal of Systems and Software* 91 (May 2014 [Jan. 22, 2014]), pp. 48–62. DOI: 10.1016/j.jss.2014.01.021 (pp. 12, 141, 178).

[218] W.-J. Wang, Y.-S. Chang, W.-T. Lo, and Y.-K. Lee. "Adaptive scheduling for parallel tasks with QoS satisfaction for hybrid cloud environments". In: *The Journal of Supercomputing* (2013), pp. 1–29. DOI: 10.1007/s11227-013-0890-2 (pp. 136, 139).

[219] C. D. Weissman and S. Bobrowski. "The design of the force.com multitenant internet application development platform". In: *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data.* ACM, 2009, pp. 889–896. DOI: 10.1145/1559845.1559942 (pp. 15, 17, 26, 27, 73).

[220] E. Wohlstadter, S. Tai, T. Mikalsen, I. Rouvellou, and P. Devanbu. "GlueQoS: Middleware to sweeten quality-of-service policy interactions". In: *ICSE '04: Proceedings of the 26th International Conference on Software Engineering.* IEEE Computer Society, 2004, pp. 189–199 (p. 139).

[221] Wolf Frameworks India Pvt. Ltd. *WOLF.* http://www.wolfframeworks.com/. [Last visited on April 1, 2014] (pp. 17, 27).

[222] WSO$_2$ Inc. *WSO2 Stratos.* http://wso2.com/cloud/stratos/. [Last visited on April 1, 2014] (p. 27).

[223] L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. "QoS-aware middleware for web services composition". In: *IEEE Transactions on Software Engineering* 30.5 (2004), pp. 311–327. DOI: 10.1109/TSE.2004.11 (p. 139).

[224] Q. Zhang, L. Cheng, and R. Boutaba. "Cloud computing: State-of-the-art and research challenges". In: *Journal of Internet Services and Applications* 1.1 (2010), pp. 7–18. DOI: `10.1007/s13174-010-0007-6` (pp. 2–5).

[225] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. "CPI2: CPU performance isolation for shared compute clusters". In: *EuroSys '13: Proceedings of the 8th ACM European Conference on Computer Systems.* ACM, 2013, pp. 379–391. DOI: `10.1145/2465351.2465388` (p. 138).

List of publications

Journal articles

- S. Walraven, D. Van Landuyt, E. Truyen, K. Handekyn, and W. Joosen. “Efficient customization of multi-tenant Software-as-a-Service applications with service lines”. In: *Journal of Systems and Software* 91 (May 2014 [Jan. 22, 2014]), pp. 48–62. DOI: `10.1016/j.jss.2014.01.021`.
- S. Walraven, B. Lagaisse, E. Truyen, and W. Joosen. “Policy-driven customization of cross-organizational features in distributed service systems”. In: *Software: Practice & Experience* 43.10 (Oct. 2013 [Oct. 13, 2011]), pp. 1145–1163. DOI: `10.1002/spe.1128`.
- S. Walraven, E. Truyen, and W. Joosen. “Comparing PaaS offerings in light of SaaS development. A comparison of PaaS platforms based on a practical case study”. In: *Computing* (Oct. 2013), pp. 1–56. DOI: `10.1007/s00607-013-0346-9`. Pre-published.
- H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck. “Cost-effective feature placement of customizable multi-tenant applications in the cloud”. In: *Journal of Network and Systems Management* (Feb. 2013), pp. 1–42. DOI: `10.1007/s10922-013-9265-5`. Pre-published.

International conference papers

- F. Gey, S. Walraven, D. Landuyt, and W. Joosen. “Building a customizable Business-Process-as-a-Service application with current state of practice”. In: *SC ’13: 12th International Conference on Software Composition.* Springer Berlin / Heidelberg, June 2013, pp. 113–127. DOI: `10.1007/978-3-642-39614-4_8`.

- P. J. Maenhaut, H. Moens, M. Verheye, P. Verhoeve, S. Walraven, E. Truyen, W. Joosen, V. Ongenae, and F. De Turck. “Migrating medical communications software to a multi-tenant cloud environment”. In: *IM ’13: IFIP/IEEE International Symposium on Integrated Network Management*. IEEE, May 2013, pp. 900–903.
- H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck. “Network-aware impact determination algorithms for service workflow deployment in hybrid clouds”. In: *CNSM ’12: Proceedings of the 8th International Conference on Network and Service Management*. IFIP, Oct. 2012, pp. 28–36.
- K. Geebelen, S. Walraven, E. Truyen, S. Michiels, H. Moens, F. De Turck, B. Dhoedt, and W. Joosen. “An open middleware for proactive QoS-aware service composition in a multi-tenant SaaS environment”. In: *ICOMP ’12: Proceedings of the 2012 International Conference on Internet Computing*. CSREA Press, July 2012.
- H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck. “Feature placement algorithms for high-variability applications in cloud environments”. In: *NOMS ’12: Network Operations and Management Symposium*. IEEE, Apr. 2012, pp. 17–24. DOI: 10.1109/NOMS.2012.6211878.
- E. Truyen, N. Cardozo, S. Walraven, J. Vallejos, E. Bainomugisha, S. Günther, T. D’Hondt, and W. Joosen. “Context-oriented programming for customizable SaaS applications”. In: *SAC ’12: Proceedings of the 27th Annual ACM Symposium on Applied Computing*. ACM, Mar. 2012, pp. 418–425. DOI: 10.1145/2245276.2245358.
- S. Walraven, E. Truyen, and W. Joosen. “A middleware layer for flexible and cost-efficient multi-tenant applications”. In: *Middleware ’11: Proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware*. Springer Berlin / Heidelberg, Dec. 2011, pp. 370–389. DOI: 10.1007/978-3-642-25821-3_19.
- S. Walraven, B. Lagaisse, E. Truyen, and W. Joosen. “Dynamic composition of cross-organizational features in distributed software systems”. In: *DAIS ’10: Distributed Applications and Interoperable Systems*. Springer Berlin / Heidelberg, June 2010, pp. 183–197.

International workshop papers

- A. Rafique, S. Walraven, B. Lagaisse, T. Desair, and W. Joosen. “Towards portability and interoperability support in middleware for hybrid clouds”. In: *CrossCloud ’14: Proceedings of the 1st IEEE INFOCOM CrossCloud Workshop*. IEEE, 2014.
- T. Desair, W. Joosen, B. Lagaisse, A. Rafique, and S. Walraven. “Policy-driven middleware for heterogeneous, hybrid cloud platforms”. In: *ARM ’13: Proceedings of the 12th International Workshop on Adaptive and Reflective Middleware*. ACM, Dec. 2013, pp. 7–12. DOI: 10.1145/2541583.2541585.

- S. Walraven, T. Monheim, E. Truyen, and W. Joosen. “Towards performance isolation in multi-tenant SaaS applications”. In: *MW4NG ’12: Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing.* ACM, Dec. 2012, pp. 1–6. DOI: `10.1145/2405178.2405184`.
- H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck. “Developing and managing customizable Software as a Service using feature model conversion”. In: *CloudMan ’12: 3rd International Workshop on Cloud Management.* IEEE, Apr. 2012, pp. 1295–1302. DOI: `10.1109/NOMS.2012.6212066`.
- S. Walraven, B. Lagaisse, E. Truyen, and W. Joosen. “Aspect-based variability model for cross-organizational features in service networks”. In: *Composition & Variability ’10: Proceedings of the 1st International Workshop on Composition: Objects, Aspects, Components, Services and Product Lines.* CEUR-WS.org, Mar. 2010, pp. 57–63. URL: `https://lirias.kuleuven.be/handle/123456789/261950`.
- S. Walraven and P. Verbaeten. “AO middleware supporting variability and dynamic customization of security extensions in the ORB layer”. In: *Companion ’08: Proceedings of the ACM/IFIP/USENIX Middleware ’08 Conference Companion.* ACM, Dec. 2008, pp. 121–123. DOI: `10.1145/1462735.1462771`.

Technical reports

- S. Walraven, D. Van Landuyt, F. Gey, and W. Joosen. *Service line engineering in practice: Developing an integrated document processing SaaS application.* CW Reports 652 (2nd revised). `https://lirias.kuleuven.be/handle/123456789/428855`. Department of Computer Science, KU Leuven, Feb. 2014 (Nov. 2013).

FACULTY OF ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE
IMINDS-DISTRINET
Celestijnenlaan 200A box 2402
B-3001 Heverlee
stefan.walraven@cs.kuleuven.be
http://www.cs.kuleuven.be