2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing

Middleware for Customizable Multi-staged Dynamic Upgrades of Multi-tenant SaaS Applications

Fatih Gey, Dimitri Van Landuyt and Wouter Joosen
iMinds-DistriNet, KU Leuven
3001 Heverlee, Belgium
firstname.lastname(s)@kuleuven.be

***Abstract*—Multi-tenant Software as a Service (SaaS) is the cloud computing delivery model that maximizes resource sharing up to the level of a single application instance servicing many customer organizations (tenants) at once. Due to this scale of delivery, a SaaS offering, once successful, becomes difficult to upgrade and evolve without affecting service continuity and tenant businesses profoundly.**

However, not all tenants are equal, and to some organizations such disruptions are more costly than to others. To account for such tenant-specific requirements, middleware for upgrading SaaS applications should support tenant-specific enactment of upgrades that allow for a customizable schedule and type of enactment in accordance to the tenant SLA.

In this paper, we present our design and implementation of a SaaS middleware that enables run-time adaptation by means of a gradual tenant-by-tenant activation of upgrades. The adaptation mechanism is multi-staged, i.e. supports configuration based on the inputs of the tenant administrator and other stakeholders, and is maximally automated.

We have validated the middleware in an OSGi-based prototype implementation and evaluated this prototype, showing negligible performance overhead of the middleware and yet clearly showcasing service continuity improvements in realistic upgrade scenarios.

I. Introduction

In the cloud computing paradigm, Software-as-a-Service (SaaS) applications are long-living Internet applications that are offered at low cost to organisations (called tenants) in order to compete directly with business applications that are operated locally. A SaaS provider's key requirements for cost-efficient service provisioning are high utilization of resources and operation at large-scale in order to attain significant economies-of-scale effects. At run time, the highest utilization level can be attained with *application-level multi tenancy* [12], i.e. sharing application instances among different tenants. For ensuring application integrity of shared instances, however, side effects of one tenant's execution must be *isolated* from those of the others [12], [51].

Such a multi-tenant SaaS application –once successful such that it provides reliably to many tenants' businesses– becomes difficult to change without affecting service continuity, consequently profoundly impacting tenant businesses. In the course of evolution, a SaaS application eventually will, however, have to tolerate service disruptions. As not all tenants are equal, to some tenant organisations such disruptions are more costly than to others, but manually managing individual exceptions is complex, error-prone and costly [43], [13]. To account for such tenant-specific quality requirements, support for dynamic upgrades of a SaaS application is necessary that allows for *tenant-specific* and *customizable* enactment.

Our proposed middleware provides support for multiple enactment strategies, enabling the SaaS developer to provide alternative impact variants on service continuity for *unforeseen* upgrades out of which one can be chosen and configured by involved stakeholders of the application's operation in a multi-staged [41] fashion; much in line with recent trends such as DevOps [16] that advocate tight integration between development and operation. Moreover, the actual dynamic enactment is carried out by means of a gradual, maximally automated, tenant-by-tenant activation, allowing tenant-specific constraints to be respected on the one hand, and improving service continuity throughout the upgrade enactment process on the other hand. More technically, our middleware enhances SaaS applications modelled as dynamic service-oriented architectures with explicit software variability support [42], [35] in that it provides a *dynamic service lookup component* for service composition based on tenant configurations which is the key component we rely on for activating upgrades.

Focusing on the impact of the run-time transition between two application versions, our middleware support for evolution renders the upgrade enactment less harmful for the stakeholders involved and in consequence more cost efficient. Specifically, it allows for (i) scalable tenant-by-tenant activation of upgrades which timely decouples its impact on different tenants, and lowers the duration of service disruption per tenant. Moreover, (ii) it supports different activation strategies, effectively allowing for different compromises in service continuity, which are (iii) configurable by different stakeholders.

We have built a prototype of our middleware as an extension of the Apache Felix [1] OSGi framework, and evaluate it in the context of a real-world multi-tenant SaaS application.

The remainder of this paper is structured as follows: First a brief background on multi-tenant SaaS applications is given in Section II and their need for and the challenges of run-time evolution is motivated, putting forward a set of key requirements for an enhanced middleware support that addresses those challenges (Section III). Sections IV, V and VI present our middleware concept, its prototype implementation and evaluation. Thereafter, Section VII discusses our design decision to use a service component as smallest unit of evolution, and extends the application scope of our middleware. Section VIII discusses related work before a final conclusion and pointers to future work are given in Section IX.

II. Background

Multi-tenant SaaS applications are commonly built as service-oriented architectures: Loosely-coupled services are stored in a repository and listed in a registry when becoming available [29], [21], [52]. For cost-efficiency, a known tactic is application-level multi-tenancy [12] where *service component* [10] instances are shared among tenants [35] in a way that application state and user data are strictly separated [12], and their composition is configured according to specific tenant requirements [29], [17], [52], [7], [42], as illustrated in Figure 1. To the artifacts that describe tenant-specific service compositions, we refer as *tenant configurations*. They reside in a tenant configuration repository (Fig. 1) that is accessible for service components, and can be maintained directly by the tenant organisation [43].

As typical for service-oriented architectures, several service components of an application collaborate in order to provide the requested end-to-end functionality to end-users (cf. service composition illustrated in Fig. 1). Here, it is important to note that service requests may be *dynamically interdependent* [32]: a service request may trigger a service component to request service from another (serial dependency), multiple service requests between two parties may be required to complete a collaboration (parallel dependency), or a combination of both. In this paper, we use the notion of *transactions* [27], [49], [32] to group the set of service requests that are dynamically interdependent.

The following common stakeholders for multi-tenant SaaS applications [52] are of significance to this paper: The *SaaS developer* and *SaaS operator* are employed by the SaaS provider. While the former is aware of implementation internals of current and upcoming versions, the latter is concerned with the provisioning and operation of the SaaS application. The *tenant administrator* and *end users* are associated with the tenant organisation. The administrator configures and manages the SaaS application on the tenant's behalf, and end-users are either its employees or customers and represent the consumers of the SaaS application.

III. Motivation

From the experience that we gained from different research collaborations with actual SaaS providers [3], [5], [4], we learned that over time, a successful multi-tenant SaaS application not only serves a numerous amount of tenants simultaneously –and therefore must provide a continuously available service– but also faces evolution that requires different kinds of upgrades [30] of the application at run time. In this setting, a dynamic enactment of upgrades is supposed to introduce the upgraded software (for example, an upgraded service

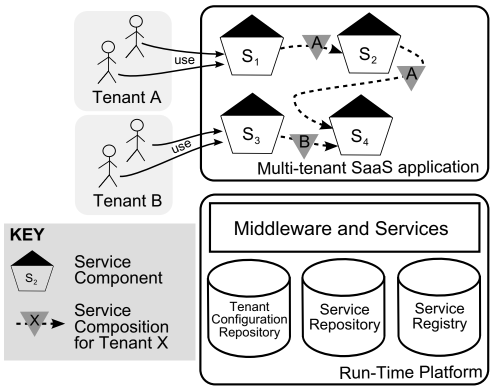

Fig. 1: Common multi-tenant SaaS applications built as dynamic service-oriented architectures and employing dynamic service composition for tenant-specific customization needs.

component), while imposing minimal (ideally, no) impact on the on-going operations.

Whenever the enactment of an upgrade unavoidably causes a disruption on service continuity (e.g. in the case of an incompatible upgrade [9]), it affects either its availability in favour of maintaining behavioural consistency or vice versa [27], [6]. To account for the interdependent relation between those qualities, we defined in earlier work *yield* and *harvest* as metrics for service continuity during dynamic enactment of an upgrade [19]: yield describes the observable continuous availability to accept new service requests, and harvest the processing completeness.

An enactment mechanism that maintains one metric of service continuity at the cost of the other provides a specific *quality compromise* that may be easily compensable by some tenants while proving harmful for others. Yet, a multi-tenant SaaS application that traditionally evolves in one shot [9], [13] has no room for considerations on a per-tenant basis. Moreover, due to the large-scale operation of multi-tenant SaaS applications, a service disruption is typically of significant duration, potentially causing a profound impact on the businesses of many tenants. For example, waiting for application-wide quiescence [27] in preparation for enacting an upgrade will cause huge loss of yield simultaneously for all tenants.

Our motivation is based on the following key observations: (i) Software upgrades of different compatibility have the potential to vary the degree of impact in terms of yield and harvest [9] (ii) activating an upgrade to a single tenant of the systems is less disruptive than enacting the upgrade system-wide (cf. time to reach quiescence), due to less application state and user data involved, (iii) related work on dynamic software upgrades and dynamic adaptations provides several alternative strategies, each favouring either yield (e.g. [6], [31]) or harvest (e.g. [27], [49], [32]) over the other, effectively enabling *alternative* quality compromises throughout an upgrade.

From these observations we derive four requirements for enhanced middleware support to evolve multi-tenant SaaS applications, while controlling the duration and business impact of a service disruption:

(R1) Tenant Isolation of Upgrade Enactment: The isolated dynamic enactment of upgrades –that is, an upgrade becoming effective (or *active*) for one tenant, while for others the SaaS application remains unchanged– is a key enabler for fine-grained per-tenant control, as it facilitates the activation for one tenant to be of a different nature than for another tenant [19]. This independent upgrade activation for tenants inherently requires support for co-existing service component and tenant configuration versions [19]: Tenants that did not (yet) upgrade must be served by the previous component version which typically shares the operational environment with the upgraded components. At the same time, the high availability context requires a phased cut-off [33], i.e. to overlap the phase-out of the current and the phase-in of the upgraded version of service components, which renders co-existing tenant-configuration versions necessary.

(R2) Support for Compatibility Nature of Upgrade: Given the different compatibility natures [9] of upgrades (i.e. each with a different potential impact on service continuity), alternative activations should be supported. In earlier work [19], we presented a collection of alternative *activation strategies* as a catalogue to address the compatibility nature of upgrades closely by customizing the activation:

(S1) Passivate and Queue. This strategy implements the quiescence [27] type of dynamic upgrade enactment in two-phases: In the preparation phase, requests that would initiate new transactions are blocked from the set of services that are affected by the upgrade, called: the *passivation set*. Requests that continue on on-going transactions, however, *are* processed, while the system is waiting for those transactions to complete. In the upgrade phase, the components-under-upgrade are first replaced, before resuming normal operation.

(S2) Don't Passivate and Process in Parallel. Current and new service component versions are deployed in parallel. Transactions initiated after the beginning of an activation are dispatched to the new service component version.

(S3) Don't Passivate and Upgrade In-progress. All service requests (regardless whether they are part of an on-going transaction or will initiate a new one) are dispatched to the newest available service component version.

(R3) Control over Service Continuity Degradation: In cases of unavoidable service disruption, alternative implementations of one or more activation strategies (hereafter referred to as *activation mechanisms*) must be provided to support alternative trade-offs between yield and harvest. As technical aspects as well as the business context play a role in identifying the most-suitable trade-off for a given upgrade, we suggest a multi-staged configuration [41] in which multiple stakeholders, e.g. the tenant administrator and SaaS operator, must be able to pick and configure an alternative.

(R4) Limited performance and scalability repercussions: The ability to address upgrade enactment on a per-tenant basis should not cause significant scalability repercussions, as scalability is a crucial driver for multi-tenant SaaS applications. Specifically, a middleware for evolution support must ensure (i) minimal overhead during normal operation, and (ii) scalable management of upgrades. For example of the latter, the enactment of upgrades should to be highly automated [19] and if manual labour is required, it should involve not single but clusters of tenants.

IV. SaaS Middleware

This section presents our middleware for the basic case of upgrading individual components that do not involve changing service interfaces[1].

In Section IV-A, we present a brief overview of our middleware architecture, while sections IV-B, IV-C and IV-D elaborate in further detail on respectively the dynamic service composition mechanism, the packaging of upgrades, and the provided evolution support that is built upon the service composition mechanism.

A. Overview

We propose a middleware to support configurable activation of upgrades at run time that internally applies a sequence of manipulations to specific service composition instances in order to converge those that refer to service components of the current version to those that refer to *activated* service component versions in a gradual, controlled manner, and limiting the impact on service continuity. A key component in our middleware is the dynamic service lookup component `DSlookup` that resolves only the single service binding of a composition that is necessary for the service invocation at hand, incrementally providing a tenant-specific, pre-configured dynamic service composition. More specifically, when a service component attempts to access a required service interface, `DSlookup` determines a concrete service that provides that interface by querying the tenant configuration (see Figure 2). In this way, the end-to-end service composition is resolved lazily and along the collaboration of service components.

[1]Section VII discusses how upgrades that do involve breaking service interfaces can be supported.

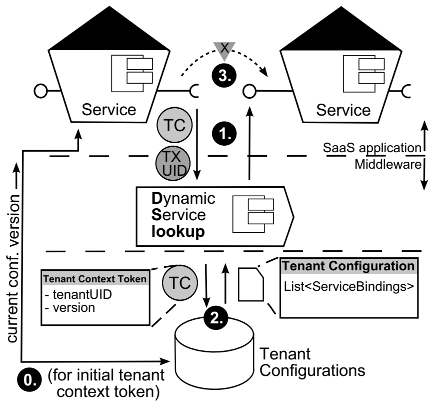

Fig. 2: Service Composition using **D**ynamic **S**ervice Lookup: DSlookup queries tenant configurations to return a matching service binding.

`DSlookup` allows, in addition, its service lookup to be manipulated based on the transaction context of the triggering service request. Our concept employs those manipulations to co-host upgraded and current service composition simultaneously, while facilitating *version consistency* [32] for transactions.

B. Dynamic Service Composition

A set of shared service components that serves tenant-specific requirements must be aware of a tenant context and be composed in a tenant-specific way, as introduced in Section II. Both can be accomplished by the use of tenant context tokens that are attached to all service requests of end-users [24]: A tenant context must be attached across all service requests of a transaction, i.e. a service component that creates (outbound) service requests r_O in the course of processing an (inbound) request r_I must copy the tenant context of r_I to r_O [24]. Service components that receive service requests without a tenant context (e.g. situated at the authoritative boundaries of the SaaS application) must generate an initial tenant context.

Our `DSlookup` component relies on a tenant context as shown in Figure 2: A service component addresses `DSlookup` to lookup a service that provides a specific interface, attaching the tenant context token and a transaction UID ((1) in this figure). To service the request, ((2) in this figure) `DSlookup` consults the corresponding tenant configuration from the repository, looking for a matching *service binding*. If successful, the reference to an instance of the specified target service is returned to the lookup requestor ((1) in Figure 2) who now is able to invoke that service ((3) in Figure 2).

Tenant configurations contain a set of *service bindings*, each specifying a mapping between a source and target (as shown in Figure 3): the source entails a required service interface as well as the identity of the calling service, and the target describes the identity of a service instance that provides the required interface. It is worth noting that on both sides of the mapping, a service is identified by name *and* version. This facilitates a version-aware configurability over

Required interface	Source (ServiceId)		⇔	Target (ServiceId)	
	Name	Version		Name	Version

Fig. 3: Format of a *Service Binding* stored in a *Tenant Configuration*.

TABLE I: Use of manipulation primitives for implementing an upgrade strategy.

Strategy	Supported Manipulation Primitives		
	Current Tenant Configuration Version	Configuration Version	Service Ref. Cache
S1	change at beginning of activation	Fail	generic
S2	change at beginning of activation	-	-
S3	change at beginning of activation	Yes	generic and TX-specific

service compositions in an application with co-existing service component versions.

The *tenant context token* must uniquely refer to a tenant configuration: not taking the evolution aspect into account, a simple unique identifier of the tenant would be sufficient. However, an evolving multi-tenant SaaS application has multiple co-existing tenant configurations in order to support the overlap of operations in two adjacent application versions for affected tenants (aka parallel cutover [33]). To that end, a tenant configuration can only be referred to uniquely from a tenant context token by tenant UID and version. An initial tenant context token can be obtained as follows: the tenant UID is derived from the service request (e.g. from authentication data [51]), and the current configuration version can be looked up from the tenant configuration repository (step 0 in Figure 2). The goal of using a version-containing tenant context token in the presence of co-existing service component versions is to ensure *version consistency* [32], i.e. service requests in a transaction that are being processed by one application behaviour (described by one specific tenant configuration version), are consistently processed with that behaviour oblivious from any co-existing service component versions.

For performance and scalability reasons, each service component caches service references in a hierarchical cache: a service reference is indexed as a generic reference to that service *and* as a transaction-specific service reference. The cache is queried in the reverse hierarchy order: only if transaction-specific service references cannot be found, generic references are searched. It is interesting to note that such a cache supports "sticky sessions": Within a session (here: *transaction*), always the same service component instance will be used, similar to the work of Rellermeyer et al. [40] in the context of OSGi.

C. Upgrade Bundle

An *upgrade bundle* is a deployment unit that is prepared by the SaaS developer and contains updated service implementation(s), activation mechanism(s) and a description of the upgrade (for example, using annotations [38]) that indicate expected service impact(s) of the enclosed activation mechanism(s).

An *activation mechanism* specifies a sequence of service composition *manipulation primitives* –implementing an activation strategy for a concrete upgrade at hand– which are supported by our middleware and available for instantiation and configuration within an activation mechanism. Moreover, we require the SaaS developer to implement and provide activation mechanisms along with an upgrade bundle. To that end, this role must be aware of those primitives and requires know-how on developing an algorithm for dynamic enactment that activates the upgrade at hand in a minimally disruptive way. Specifically, the following primitives are available:

Change Current Tenant Configuration Version. The current tenant configuration version (stored in the tenant configuration repository) can be changed. This causes new transactions to be assigned a tenant context with a changed tenant configuration version.

Change Configuration Version of Tenant Context Token. At `DSlookup`, the configuration version entry of a tenant context token can be manipulated for specific service lookup queries *before* the actual lookup. Moreover, this component can be set to fail a lookup deliberately, i.e. return that no services are available. For communicating such manipulations, we introduce a data structure called *Service Lookup Manipulation Directive* that is similar to a pattern-matching rule: a matching part describes a service lookup query in terms of the source information of a service binding, i.e. required service interface and a source `ServiceId`, while the action part can either specify a specific tenant configuration version to be set in the tenant context token, or to fail the lookup in question.

Flush Service Reference Cache. This cache (maintained by every service component), can be cleared for transaction-specific or generic service references for a given service.

Table I shows our expected mapping between activation strategies and manipulation primitives. For example, S1 (aka Passivate and Queue) requires the upgrade of the current tenant configuration version at the beginning of the activation and will disable parts of the service composition by deliberately failing the service lookup targeting the service components that are being upgraded. Finally, it may flush the cached generic service references to those service components.

D. Evolution Support

Our middleware enables configuration of upgrade activations and carries out the configured activation autonomously. In this section, first the *configuration options* for an activation are explained. Then, the *activation protocol*, a sequence of activities that result in the activation, is presented. Finally, the *activation controller*, our key component to coordinate manipulation primitives (as described in Sec. IV-C) is introduced.

Activation Configuration Options: The activation can be configured by either the tenant administrator or the SaaS operator who is able to set the following options.

Time of Activation. The stakeholder may provide a specific time of activation, typically aligned with the business context of the tenant or the (expected future) load of the application.

Activation Mechanism. The stakeholder may pick one of the alternative activation mechanisms that may have been provided by the SaaS developer, each imposing a different trade-off between yield and harvest.

Activation Protocol: After the deployment of an upgrade bundle, the following protocol describes its activation for a tenant:

Creation of Tenant Configuration. A new tenant configuration must be provided that refers to the upgraded service component(s), typically created by the tenant administrator using a self-service.

Creation of Activation Configuration. An activation configuration must be provided.

Upgrade Activation. This step runs in full automation, passing through multiple *activation phases*: Given the content of

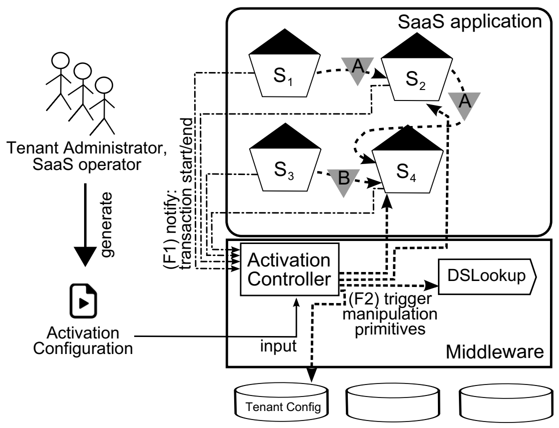

Fig. 4: Features of the Activation Controller: (F1) Monitoring on-going transactions (F2) Remotely invoking manipulation primitives on specific service component instances.

the upgrade bundle and the two configurations of the previous steps as input, the activation controller upgrades a service component from a *current* to a *new version* in the following phases:

(A1) Preparation. In this phase, only the current version of the service component is available and potentially in use.

(A2) Transition. This phase starts when the new service component version becomes available: Service component instances of current and new version are active in this phase. This phase must actively converge towards the next phase A3. Moreover, we propose to set time boundaries for this phase to ensure termination of the activation.

(A3) Postparation. At the point in time at which the current service component version is no longer used, the postparation phase starts. Its purpose is to clean up settings and manipulations that were only necessary for the transition phase, such that after completion, the application returns to a normal operation (or to phase A1 of the next activation for that tenant).

Activation Controller: The activation controller provides two features, as shown in Figure 4: (F1) monitoring the presence of active transactions, and (F2) coordinating a set of service composition manipulation primitive invocations to activate an upgrade as described by the activation configuration and activation mechanism. For the former, each service component is required to broadcast a notification at the beginning and the end of a transaction. The activation controller observes those notifications to maintain an accurate, up-to-date view of on-going transactions. For the latter, the controller can infer the current phases of an activation (A1, A2 or A3), either based on an event (e.g. "no transaction active anymore") or on time (e.g. n seconds after previous phase), and invokes the primitives of the given mechanism that are applicable to the current phase. Several instances of an activation controller can co-exist and balance the load of activations, which contributes to the scalability of our approach. Yet, it is important to bear in mind that a single activation controller is responsible for a single activation, and that for one tenant, only a single activation can be performed at a time.

V. Implementation

We have built[2] our middleware for enhanced evolution support on top of the OSGi framework Apache Felix [1]. In this section, we first introduce the concepts of the OSGi framework that we employed. Then we present the implementation of our two key components `DSlookup` and the activation controller along other supportive components.

A. Background on the OSGi framework

The OSGi framework is a highly modular middleware for Java to build service-oriented architecture (SOA) applications and provides explicit and fine-grained control over component dependencies. Particularly, provided module versions can be loaded into only the context of the required module which enables the co-existence of otherwise incompatible service component versions while imposing only little constraints. Such capabilities successfully hide complexity of co-existing service component versions in a SOA application.

On start-up (at service registration) a service component has to specify its provided service interfaces and may expose additional properties in the form of key-value pairs. Service discovery, consequently, may match on both: provided service interface and properties. For the sake of modularity, OSGi does not encourage pro-active service discovery but instead promotes a reactive discovery based on dependency injection for which a service consumer must signal required interfaces and may choose thereafter either to become passive and wait for a service discovery or to continue and be called back on a parallel thread. Under the hood, OSGi uses an event-based infrastructure to propagate service (dis-)appearances.

B. Dynamic Service Lookup and Composition

While our concept requires to find specific (by UID and version) service component instances, the OSGi framework mandates for discovery by service interface and untyped properties only. Therefore, we introduce the `ServiceId` data type (as seen in the service binding in Sec. IV-C) that stores a UID and a version and that can be easily used to label and discover service components. Moreover, we provide the Java class `MTServiceContext` that implements the template method pattern to abstract from the concrete use of the `ServiceId` object, and that must be specialized by any service component. It provides methods to instantiate and register local classes as a service component on the one hand, and to deal with service discovery on the other hand. Internally, the Java class name of the service component implementation is used as the UID, and the enclosing OSGi bundle's version as service version.

Our `Dynamic Composition Enablement Layer` (DCEL), as shown in Figure 5, implements two features: it provides the template class `MTServiceContext`, and contains the service reference cache. Both are implemented by encapsulating outbound service component calls: For each service invocation, the method `callService` of the DCEL must be called, specifying a required service interface, a transaction identifier and a call-back function which is called by the DCEL synchronously after a successful lookup, and to which the resulting service reference is passed as an argument. Internally, DCEL uses the UID and tenant context of the local service and the service reference cache to lookup the service.

`DSlookup` supports a service lookup directly or via the tenant configuration: If the lookup request (pre-processed by

[2]available at https://distrinet.cs.kuleuven.be/software/flesensevo

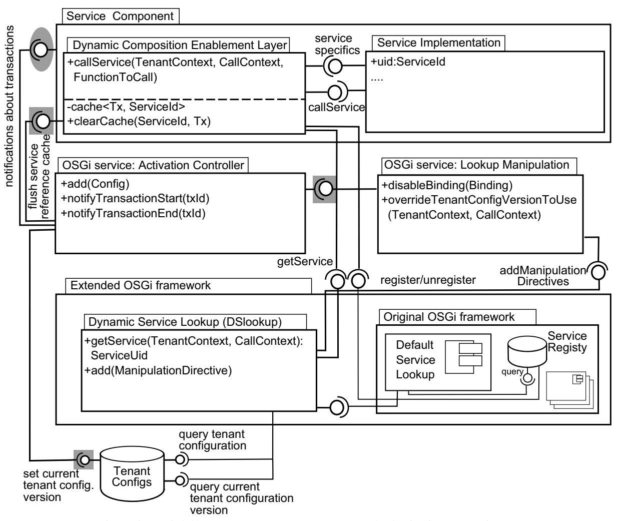
Service Component
notifications about transactions
flush service reference cache
Dynamic Composition Enablement Layer
+callService(TenantContext, CallContext, FunctionToCall)
-cache<Tx, ServiceId>
+clearCache(ServiceId, Tx)
service specifics
callService
Service Implementation
+uid:ServiceId
....
OSGi service: Activation Controller
+add(Config)
+notifyTransactionStart(txId)
+notifyTransactionEnd(txId)
OSGi service: Lookup Manipulation
+disableBinding(Binding)
+overrideTenantConfigVersionToUse (TenantContext, CallContext)
getService
register/unregister
addManipulation Directives
Extended OSGi framework
Dynamic Service Lookup (DSlookup)
+getService(TenantContext, CallContext): ServiceUid
+add(ManipulationDirective)
Original OSGi framework
Default Service Lookup
Service Registy
query
query tenant configuration
set current tenant config. version
Tenant Configs
query current tenant configuration version

Fig. 5: Middleware components and their interactions.

the DCEL) contains a `ServiceId` of the desired service, then this service is looked up directly (this is a natural extension of OSGi's default behaviour using specific `ServiceIds`). Otherwise, if the *source side* of a service binding is enclosed, then an indirect lookup consulting the tenant configuration as described in Section IV-B is performed.

C. Evolution Support

In our prototype, we employ the publish-subscribe infrastructure provided by the OSGi EventAdmin [2] to (i) notify the activation controller(s) about on-going transactions in the SaaS application (connectors marked with an oval in Figure 5) and (ii) to remotely invoke manipulation primitives (square-marked connectors in Figure 5). For example of the latter, the component-local service reference cache can only be flushed at the component, and changing the configuration version of a context token requires a manipulation at the `DSlookup` component.

Support for Manipulation Primitives

We implemented the tenant configuration repository as a simple, centrally-deployed RESTful web service (as depicted at the bottom of Figure 5). It provides interfaces for CRUD operations on both tenant configurations and tenants' current configuration versions.

As the `DSlookup` component is located within the OSGi framework (run-time environment), each distributed instance of a service component will access its own instance. For allowing injection of manipulation directives into a distributed OSGi framework instance, an event sink (OSGi service `Lookup Manipulation`, see Figure 5) is deployed at each OSGi framework instance that forwards those directives to `DSlookup`.

For the manipulation of the service reference caches, we provided specific event sinks in the Dynamic Composition Enablement Layer which are used by every service component.

Activation Mechanisms and Controller

We modelled *activation mechanisms* as Java objects that are provided by the deployed upgrade package and executed by the activation controller. The objects programmatically use the capabilities provided by the middleware at the phases indicated by the activation controller. To ensure termination in our prototype, we set the time-out for the transition phase A2 to 1800 seconds.

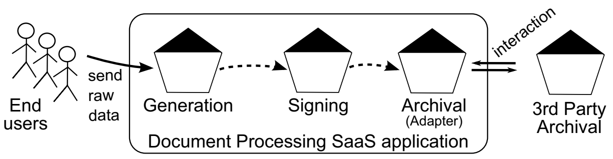

Fig. 6: Interactions between service components, external services, and end-users of the document processing muli-tenant SaaS application that we use for evaluation.

The *activation controller* is implemented as an OSGi service providing two event sinks, as shown in Figure 5: one for the notifications on transactions, and another one to accept activation configurations. Receiving the latter triggers an action for the therein specified upgrade and tenant.

VI. Evaluation

We evaluate our enhanced middleware support for evolution of multi-tenant SaaS application in the context of a real-world SaaS provider that provides a document processing service [21]: We enact three different upgrades to a multi-tenant SaaS application that is constantly busy processing requests. To showcase the flexibility of our middleware, we define up-front a set of application-specific invariants that must hold true in order to process transactions of service requests correctly.

A. Document Processing SaaS application

This multi-tenant Software-as-a-Service application is a composite service that generates, digitally signs, and archives different kinds of documents (e.g. pay checks, invoices) on behalf of its tenant organizations. Herein, the archival service is offered by a third party service and connected to the SaaS application by an adapter service, as shown in Figure 6. The external archiving service is paid at the basis of the up-time per tenant.

Once set up for a specific tenant, the SaaS application accepts a list of raw data (*batch*) in which each entry represents the inputs for generating one document. Each document in this batch is generated, signed, and archived sequentially. For correctly processing a batch, the following invariants must be met:

Invariant 1: All documents of one batch must be signed by the same cryptographic key material.

Invariant 2: All documents of one batch must be archived at the same storage provider.

Invariant 3: Due to service fees, only one kind of archiving service shall be used at a time.

Implementation

For evaluation purposes, we implemented a simplified version of this SaaS application on top of our middleware prototype[3]: each service component performs a wait operation of 300 milliseconds to simulate a corresponding workload. Between the processing of two documents of a batch a delay of 350 milliseconds in used to account for I/O operations.

For each service component the following information is recorded in an event log for post-processing: the UID of the service, service results (i.e. successful or exceptions thrown),

[3]available at https://distrinet.cs.kuleuven.be/software/flesensevo

the time it completed, whether a lookup was necessary for invoking the next service (i.e. at a cache miss), and the delay of said lookup. The external archiving service is modelled to shut down after an idle time of 2 seconds. That is, as long as the adapter to that service keeps sending requests in shorter intervals, the external service will be held back from shutting down and, in consequence, continue billing fees for its up-time.

B. Evolution

We evaluate the enactment of three different yet related upgrade scenarios, each one representing fundamentally different types of upgrades.

Upgrade Scenario 1: Switching Archiving Service. In this scenario, the SaaS application is required to switch to another external archiving service and must therefore upgrade its internal adapter service. The current and the future external service can be used in parallel, however, attention must be given to service fees: accessing both services during the enactment will result in doubling the service fees. Ideally, the SaaS application should therefore complete all requests that require the current external service first. When invariants 2 and 3 are respected, this implies completing the archiving step of *all* on-going batches (and thus, all on-going service requests) before starting to use the new archiving service.
This upgrade scenario is representative of a type in which requests of on-going and new transactions *share significant side-effects* in an incompatible way (here: When the use of both archival services are frequently alternated, the fees are doubled). Typically, the dynamic enactment of such an upgrade requires quiescence or tranquility.

Upgrade Scenario 2: Changing Cryptographic Keys. In this upgrade, the signing service is replaced, together with its cryptographic key material. In order to respect invariant 1, an upgrade should only affect transactions that have been started with the new service component version.
This upgrade scenario is representative of a type in which requests *within* a transaction share side-effects but either *do not share a significant side-effect with other requests or do it in a compatible way*.

Upgrade Scenario 3: Adding Meta-data to Signature. The digitally signing service shall add a report (i.e. naming the signing device and a time stamp) to its output.
This upgrade scenario is representative of a type in which a request does either *not share any significant side-effects with another or does it a compatible way*.

C. Service Continuity Metrics

As mentioned in Section IV, degraded service continuity, for example during the enactment of an incompatible [9] upgrade, can be measured in terms of yield and harvest: Yield is a relative metric that involves comparing to the performance of the application during normal (i.e. non-degraded) operation. Harvest expresses relative completeness of requests, thereby comparing against the yield of the application. In this section, the concrete determination of these metrics in the context of the three different upgrades is laid out.

Yield: Yield is determined by the amount of document processing requests accepted. Such a request is considered accepted only if all service components accept it.

Harvest for Upgrade 1: If archive requests are received more frequently than each every 2 seconds, the external service does not shut down. The processing times of the adapter service (read from the event log) are used to trace the time periods in which two external archiving services are up simultanously. Documents that have been archived during this period are considered invalid (due to invariant 3), and are not counted into the harvest.

Harvest for Upgrade 2: If two documents of a batch are signed with a different key, the batch is considered invalid (due to invariant 1). All documents of that batch are not counted into the harvest.

Harvest for Upgrade 3: Whether or not additional meta-data is present along with the signature has no relevance to the validity of a document.

D. Setup

We have deployed our middleware prototype together with Apache Karaf 4.0M2, the document processing SaaS application (Section VI-A), and the tenant configuration repository on a standard laptop computer with an Intel i3 2.3 GHz CPU, 4 GB memory, and a Samsung 256GB solid state disk. We have run multiple evaluations: One round for each combination of upgrade and activation strategies (3x3=9 variants in total), and one round of an evolution-free operation (used as a baseline).

Workload: For each round, the system is simultaneously servicing 3 tenants (hereafter referred to as tenant A, B and C), each with three batches before and three batches after the upgrade activation, as indicated in Table II. In order to account for different workloads and different times at which those arrive at the SaaS application, we chose batches that contain random amount of documents, and different boundary intervals for the tenants. For example, tenant B may have batches containing at most 50 documents, thereby simulating an early start of a batch, following the assumption that 100 documents is a typical size for a batch.

Evaluation Round: Each round is repeated 5 times to limit the effect of accidental outliers, and consists of the following three phases: In a *warm-up phase*, the SaaS application is booted, the upgrades are deployed (but are not activated yet), and the emulated payload of tenants and upgrade activation configurations is loaded into memory. A schedule, as shown in Table II, describes events for the *operation phase* by their logical time (expressed in seconds). This phase starts with a trigger that depicts the logical time of zero of the application. As the time progresses, scheduled events are fired via the publish-subscribe subsystem of OSGi. The *cool-down phase* is triggered, when the application is no longer processing any payload. Here, caches are cleared and current tenant configuration versions are reset to their initial value.

E. Results

Table III shows a summary of the evaluation results: each column represents an evaluation round (mean over all 5 iterations, including standard derivation), and the rows describe harvest and yield measurements as well as the amounts in which a dynamic lookup (using `DSlookup`) and the service reference cache have been used to lookup a service.

Column (1) shows one representative of the measurement for tenants A and B of an evaluation round *with* evolution in

TABLE II: Evaluation round schedule.

Time	Tenant	Event
0	A	start of 3x batch, each 20-130 docs
	B	start of 3x batch, each 20-50 docs
	C	start of 3x batch, each 20-100 docs
10	C | all	upgrade activation
16	all	start of 3x batch, each 50-150 docs

TABLE III: Evaluation measurements.

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)
Strategy	*	∅	S1	S2	S3	S1	S2	S3	S1	S2	S3	S1	S1	S1
Upgrade	*	∅	U1	U1	U1	U2	U2	U2	U3	U3	U3	U1@all	U2@all	U3@all
Yield (#docs)	100% (664)	100% (655)	68% (444±11)	100% (655±0)	100% (655±0)	68% (445±4)	100% (655±0)	100% (655±0)	69% (450±4)	100% (655±0)	100% (655±0)	55% (361±6)	55% (357±4)	54% (356±6)
Harvest (#docs)	100% (664)	100% (655)	100% (444±11)	44% (286±14)	83% (546±149)	100% (445±4)	100% (655±0)	92% (601±122)	100% (450±4)	100% (655±0)	100% (655±0)	100% (361±6)	100% (357±4)	100% (356±6)
DSlookup [#]	8±5	7±4	222±8	12±4	10±2	220±4	16±10	11±3	220±7	9±2	8±3	306±6	309±6	310±7
Cache [#]	1320±5	1303±4	1087±9	1298±4	1300±2	880±8	1294±10	1299±3	885±10	1301±2	1302±3	1004±6	704±9	701±12
Tenant	A,B	C												

which tenant C but not tenants A or B have been activating an upgrade (as indicated in Table II). In the next column (2), the measurements for tenant C for an evolution-free round are given; its yield specifies the reference value for all subsequent columns. For tenant C, columns (3)-(11) present results of evaluation rounds for the 9 variations as described above, and columns (12)-(14) print the results of an upgrade enactment without isolated per-tenant activation.

These results mainly confirm that our middleware meets the requirements put forward in Section III:

R1: Comparing yield in (3), (6), and (9) with those of columns (12)-(14) show an improvement due to isolated activation.

R2: Different strategies score best in columns (5), (7), (10) and (11), depending on the type of the upgrade, indicating the value of supporting multiple enactment strategies.

R3: Columns (3) and (5) represent service degradations, two alternatives, each of which may be preferable in a specific business context (availability over completeness or vice versa) – this has been our premise in R3 and is only feasible in satisfaction of R1.

R4: To assess the performance impact, we measured all uses of `DSlookup` during the evaluation rounds, including cases in which `DSlookup` deliberately fails (and therefore does not query the tenant configuration repository), and calculated a mean service lookup time of 318,46±1707,16 milliseconds (ms). Around 60% of this overhead can be accounted to querying the tenant configuration repository (1001,28±34,13 ms), and by consequence, the `DSlookup` component introduces around 700 ms overhead[4]. It is however important to note that, due to hierarchical caching, the `DSlookup` component is used fairly infrequently[5], i.e. only once per service component when processing the first (parallel) request(s) of a tenant using a specific application version, and only for the first (parallel) request(s) of a tenant after each upgrade enactment. For enabling individual per-tenant enactment, the SaaS provider's necessary manual efforts are limited to providing activation mechanisms and configurations, both on a per-upgrade basis and, thereby, addressing many tenants at once. We conclude that performance and scalability impact of our middleware is acceptable.

VII. Discussion

We chose a service component to be the smallest *unit of evolution* for several reasons: Firstly, service-oriented architectures provide mature support for connecting service components – both in terms of technology and eco-system. By our choice, these connection mechanisms can be leveraged when rewiring the application to include the upgraded piece of software. Secondly, as others [28], we consider the evolution at the architectural level (i.e. replacing service component modules) to be the appropriate level of granularity: its enactment can be generic [27] and does not require technology-specific measures (cf. memory invasive operations) which are not widely applicable in our context, as a cloud provider may dictate the technology of the operational environment. Simultaneously, it provides sufficient granularity to replace only necessary fragments of the application, which leads to high amount of service components that can be shared across application versions.

Another motivation to activate an upgrade by adapting the application's control flow is run-time and management efficiency of the enactment: components that consume the service-component-under-upgrade are only made to switch their reference at the right point in time and only for in-progress transactions. As thereby most of the consuming component's architecture can operate in an unchanged manner, only a little computational overhead is required from the activation process that provides support for requests of the "old" generation while the application is committed to serve the "new". Moreover, in presence of many tenants activating different upgrades, maximally sharing service components leads to less necessary service component deployments, which increases manageability.

An obvious limitation of our approach as presented in Section IV is the interface dependency between the consumer and (two versions of) the provider. An upgrade that introduces a change in a service component's interfaces would require its consumers to support this new interface *additionally*. That is at the same time our proposition: When an upgrade U_{prime} imposes a (syntactically or semantically) changed service interface, then its consuming services must be upgraded in advance (U_{pre}) to support the change of U_{prime}. Note that U_{pre} is a compatible upgrade, as it does not expose any changed behaviour to the in-progress transactions. Thereby not causing service disruption, the additional activation U_{pre} is cost-neutral to the stakeholders involved and is therefore negligible as far as service continuity is concerned.

Our middleware naturally supports upgrades that affect multiple, interdependent service components, providing a bare-bone infrastructure for activation mechanisms: While our middleware is concerned with the indication of the presence of activities started before the activation and provides a set of building blocks for the manipulation of service composition instances, it is the obligation of the upgrade-specific activation mechanism (provided by the SaaS developer) to coordinate in-

[4]Removing the static service lookup time of around 23,38±242,46 ms from the equation.

[5]In our evaluation, only 1% of service requests triggered a dynamic lookup. Exceptions observed in columns 3, 6, 9, and 12-14 represent the cases in which `DSlookup` deliberately fails and can therefore be neglected, as performance is only relevant for successful requests.

dividual manipulations to account for interdependencies among the service components that are upgraded.

VIII. Related Work

We discuss related work in four domains:

a) Dynamic Software Updates and Adaptation: Updating an application at run time has been studied for decades [8], [22], [27], increasingly reducing the impact on its normal operation. This body of work leads to two dominant and essentially different approaches: *Dynamic software updates* [23] score well in service continuity and focus on update safety, but are limited to specific types of upgrades. Moreover, they usually depend on memory-invasive operations which are not applicable in a cloud context. *Dynamic Adaptation* concepts [27], [49], [32] are appliable for any type of upgrade, are technology independent but require a safe state of the application before upgrade, thereby imposing significant impact on availability. *Self-adaptive Systems* [7], [34] represent a subset of dynamic adaptation approaches that address the complexity of a large and interconnected system in an automated way [28]: Monitoring, Analysis, Planning, and Execution form the closed feedback loop in which a system is able to react to an environment based on a computational model. Our middleware supports any type of upgrade and different types of dynamic upgrade concepts to account for the different nature of upgrades in terms of service continuity. Moreover, in our context, the best of alternatives cannot be determined by computation, as it requires assessment of the business context and human input.

b) Adaptation through Control Flow Manipulation: Jørgensen et al. [24] manipulate control flow within component collaborations to customize client-specific behaviour. Unlike us, they assume a "closed-world", i.e. choosing between upfront anticipated components. In the community of Aspect-oriented Programming, the manipulations supported by our middleware are known as regulative ("delaying operation" or "preventing continuation") and invasive aspects (adding and removing operations) [26]. While Aspects are technology-dependent (and therefore not suitable for the cloud context), they are complementary to our work, providing a well-explored formal foundation for manipulating control flows.

c) Business Process- and Workflow Management: Another area is that of workflow- and business process management [39], [45]: A *workflow definition* (WFD) specifies a sequence of activities and supports conditional branches, similar to a control flow graph of an application. As workflows can incorporate interactive activities, their execution is often a long-running process [25]. A *workflow execution instance* has an *execution state* and *instance variables*. Run-time adaptation is a popular topic in this research area. In its simplest case, abstract activities are defined in the WFD and lazily bound to service instances [35], based on a policy [18] or on the argument (object type) passed to the external service [11] – an approach that applies to a closed world but not to our context. Alternatively, the definition a workflow execution is using is *dynamically changed* [14], [44]; this approach is limited to certain changes, thereby similar to Dynamic Software Updates. A fundamentally different approach proposes at the end of one activity to determine the next one, thereby abandoning predefined work- or control flows: Case handling [47] mandates (manual or computation-based) roles to decide on the basis of available activities and current instance variables of an workflow execution, while constraint-based approaches [36], [46] evaluate decisions computationally. As such an approach anticipates coarse-grained request-aware service compositions, it would add significant overhead in our context of a fine-grained but pre-configured service composition during normal operation. Moreover, we cannot demand knowledge about composability from the end-users.

d) Management and Middleware Support for Evolution of Cloud Applications: Dumitras et al. [13] propose a middleware that moves an entire application to a "parallel universe" to avoid inconsistencies of otherwise incremental upgrades of enterprise-sized cloud applications. Opposed to theirs, our approach promotes a service component as the smallest unit for evolution. Others [29], [17] support adaptation and evolution of a SaaS application for anticipated upgrades. Ertel et al. [15] present a framework to support dynamic evolution of dataflow programs. While their support is based on types of applications that differs from multi-tenant SaaS application, their work is complementary to ours as it focuses on an algorithm for automated enactment (corresponds to our activation mechanism), accounting for state-transfer, referential integrity and timeliness of dependent upgrades. In the context of the Internet of Things, a middleware has been proposed for dynamic adaptation in the sense that collaboration with unanticipated services is supported seamlessly [50]. While we focus on service continuity improvements, their focus is on service discovery and ad-hoc interaction.

IX. Conclusion and Future Work

We presented our enhanced middleware support for evolution of multi-tenant SaaS applications that incorporates control of multiple stakeholders (developer, operator and tenant administrator) using a multi-staged configuration, and allows for customized service continuity compromises (from a technical- and cost-perspective) when enacting different types of software upgrades. A key enabler is our dynamic service lookup component that allows fine-grained (yet efficient) manipulation of service compositions, and is used for a phased cut-off: Operations of the current application are gradually phased out while the application provides the functionality of its upgraded version. This approach enables the SaaS developer to implement an upgrade activation strategy using primitive manipulation operations provided by the middleware. Our prototype showed a significant improvement in service continuity (expressing in yield and harvest) during upgrade enactment while imposing negligible performance overhead at normal operation.

In future work, we will strengthen the evaluation of our prototype by involving additional SaaS applications and upgrade scenarios. Furthermore, we will focus on the support for different stakeholders in our middleware. Increasing automation for the SaaS developer is one example: a set of predefined activation algorithms or automated reasoning based on the application's observed behaviour could be supported to define an activation mechanism. Moreover, integrating change impact analysis techniques [37] would enrich the activation configuration in that it would project a technical expectation on service disruption to the abstraction of the application. Finally, we will explore the dimension of service continuity further, addressing key questions, such as: What are common cases for low harvest? Is an application-specific recovery of those requests possible? If so, primitive building for recovery could be another addition to the middleware.

Our middleware architecture and in particular the mechanism for tenant-level fine-grained manipulation of service compositions are key contributions to the challenges of continuously evolving multi-tenant SaaS applications for the sake of time-to-market improvements [48], [20]. Moreover, the flexibility we provide for service compositions has strong potential in other challenging and highly dynamic customization scenarios [41], [48], such as in the federation of SaaS applications.

Acknowledgments

This research is partially funded by the Research Fund KU Leuven, the ADDIS research program funded by KU Leuven GOA and D-BASE. The D-BASE project is co-funded by iMinds (Interdisciplinary Institute for Technology).

References

[1] Apache felix framework v4.6.1. http://felix.apache.org.

[2] OSGi compendium release 5. http://www.osgi.org/Download/Release5.

[3] iMinds-CUSTOMSS Project, http://distrinet.cs.kuleuven.be/research/projects/showProject.do?projectID=CUSTOMSS, Jan. 2013.

[4] iMinds-D-BASE Project: Optimization of Business Process Outsourcing Services, http://www.iminds.be/en/projects/2014/05/06/d-base, 2014.

[5] iMinds-DMS2 Project: Decentralized Data Management and Migration of SaaS, http://www.iminds.be/en/projects/2014/03/06/dms2, 2014.

[6] S. Ajmani, B. Liskov, and L. Shrira. Modular software upgrades for distributed systems. In *ECOOP*. Springer, 2006.

[7] L. Baresi, S. Guinea, and L. Pasquale. Service-oriented dynamic software product lines. *Computer*, 45(10):42–48, 2012.

[8] T. Bloom and M. Day. Reconfiguration and module replacement in argus: theory and practice. *Software Engineering Journal*, 1993.

[9] E. Brewer. Lessons from giant-scale services. *Internet Computing, IEEE*, 5(4):46–55, Jul 2001.

[10] H. Cervantes and R. S. Hall. Autonomous adaptation to dynamic availability using a service-oriented component model. In *ICSE*, 2004.

[11] C. Chiao, V. Kuenzle, K. Andrews, and M. Reichert. A tool for supporting object-aware processes. In *EDOCW*, 2014.

[12] F. Chong and G. Carraro. Architectural strategies for catching the long tail., http://msdn.microsoft.com/en-us/library/aa479069.aspx, 2006.

[13] T. Dumitraş and P. Narasimhan. Why do upgrades fail and what can we do about it?: Toward dependable, online upgrades in enterprise system. In *Middleware*, 2009.

[14] C. Ellis, K. Keddara, and G. Rozenberg. Dynamic change within workflow systems. In *COCS '95*, 1995.

[15] S. Ertel and P. Felber. A framework for the dynamic evolution of highly-available dataflow programs. In *Middleware*, 2014.

[16] B. Fitzgerald and K.-J. Stol. Continuous software engineering and beyond: Trends and challenges. In *RCoSE*, RCoSE. ACM, 2014.

[17] J. García-Galán, L. Pasquale, P. Trinidad, and A. Ruiz-Cortés. User-centric adaptation of multi-tenant services: Preference-based analysis for service reconfiguration. In *SEAMS*, 2014.

[18] K. Geebelen, S. Michiels, and W. Joosen. Dynamic reconfiguration using template based web service composition. In *MW4SOC*, 2008.

[19] F. Gey, D. Van Landuyt, W. Joosen, and V. Jonckers. Continuous evolution of multi-tenant saas applications: A customizable dynamic adaptation approach. In *Pesos*, May 2015.

[20] F. Gey, D. Van Landuyt, S. Walraven, and W. Joosen. Feature models at run time: Feature middleware for multi-tenant saas applications. In *Models@run.time*, 2014.

[21] F. Gey, S. Walraven, D. Van Landuyt, and W. Joosen. Building a customizable Business-Process-as-a-Service application with current state-of-practice. In *Software Composition*, 2013.

[22] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *Software Engineering*, 1996.

[23] C. M. Hayden, S. Magill, M. Hicks, N. Foster, and J. S. Foster. Specifying and verifying the correctness of dynamic software updates. In *Verified Software: Theories, Tools, Experiments*. 2012.

[24] B. N. Jørgensen and E. Truyen. Evolution of collective object behavior in presence of simultaneous client-specific views. In *OOIS*. 2003.

[25] M. B. Juric, A. Sasa, and I. Rozman. Ws-bpel extensions for versioning. *Information and Software Technology*, 51(8):1261 – 1274, 2009.

[26] S. Katz. Aspect categories and classes of temporal properties. In *Transactions on Aspect-Oriented Software Development I*. 2006.

[27] J. Kramer and J. Magee. The evolving philosophers problem: dynamic change management. *Software Engineering*, 1990.

[28] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *FOSE*, 2007.

[29] I. Kumara, J. Han, A. Colman, and M. Kapuruge. Runtime evolution of service-based multi-tenant saas applications. In *ICSOC*. 2013.

[30] M. Lehman, J. Ramil, P. D. Wernick, D. Perry, and W. Turski. Metrics and laws of software evolution-the nineties view. 1997.

[31] W. Li. Evaluating the impacts of dynamic reconfiguration on the qos of running systems. *Journal of Systems and Software*, 84(12), 2011.

[32] X. Ma, L. Baresi, C. Ghezzi, V. Panzica La Manna, and J. Lu. Version-consistent dynamic reconfiguration of component-based distributed systems. In *FOSE*. ACM, 2011.

[33] R. McLeod and E. Jordan. *Systems development: a project management approach*. Wiley, 2001.

[34] B. Morin, O. Barais, J. Jezequel, F. Fleurey, and A. Solberg. Models@run.time to support dynamic adaptation. *Computer*, 42, 2009.

[35] T. Nguyen, A. Colman, M. A. Talib, and J. Han. Managing service variability: state of the art and open issues. In *VaMoS'11*. ACM, 2011.

[36] M. Pesic, M. Schonenberg, N. Sidorova, and W. van der Aalst. Constraint-based workflow models: Change made easy. 2007.

[37] S. L. Pfleeger and S. Bohner. A framework for software maintenance metrics. In *Software Maintenance*, 1990.

[38] S. C. Previtali and T. R. Gross. Annotations for seamless aspect-based software evolution. In *RAM-SE08-ECOOP08 Workshop*, 2008.

[39] D. Redlich, G. Blair, A. Rashid, T. Molka, and W. Gilani. Research challenges for business process models at run-time. In *Models@run.time*. Springer, 2014.

[40] J. Rellermeyer and S. Bagchi. Dependability as a cloud service - a modular approach. In *DSN Workshops*, 2012.

[41] J. Schroeter, P. Mucha, M. Muth, K. Jugel, and M. Lochau. Dynamic configuration management of cloud-based applications. In *SPLC*, 2012.

[42] J. Siljee, I. Bosloper, J. Nijhuis, and D. Hammer. Dysoa: Making service systems self-adaptive. In *ICSOC*. Springer, 2005.

[43] W. Sun, X. Zhang, C. J. Guo, P. Sun, and H. Su. Software as a service: Configuration and customization perspectives. In *Services*, 2008.

[44] W. van der Aalst. Exterminating the dynamic change bug: A concrete approach to support workflow change. *IFS*, (3), 2001.

[45] W. van der Aalst, A. Hofstede, and M. Weske. Business process management: A survey. In *Business Process Management*. 2003.

[46] W. van der Aalst, M. Pesic, and H. Schonenberg. Declarative workflows: Balancing between flexibility and support. (2), 2009.

[47] W. M. van der Aalst, M. Weske, and D. Grünbauer. Case handling: a new paradigm for business process support. *DKE*, (2), 2005.

[48] D. Van Landuyt, S. Walraven, and W. Joosen. Variability middleware for multi-tenant saas applications. In *SPLC*, 2015.

[49] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *Software Engineering*, Dec. 2007.

[50] R. O. Vasconcelos, I. Vasconcelos, and M. Endler. A middleware for managing dynamic software adaptation. In *ARM*, 2014.

[51] S. Walraven, E. Truyen, and W. Joosen. A Middleware Layer for Flexible and Cost-Efficient Multi-tenant Applications. In *Middleware*. Springer, 2011.

[52] 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. *Journal of Systems and Software*, 2014.