2017 IEEE 5th International Conference on Future Internet of Things and Cloud

An Integrated Platform for Dynamic Adaptation of Multi-Tenant Single Instance SaaS Applications

Fatma Mohamed
Khalifa University
Abu Dhabi, UAE
fatma.mohamed@kustar.ac.ae

Rabeb Mizouni
Khalifa University
Abu Dhabi, UAE
rabeb.mizouni@kustar.ac.ae

Mohammad Abu-Matar
Eitsalat British Telecom Innovation Center
Abu Dhabi, UAE
mohammad.abu-matar@kustar.ac.ae

Mahmoud Al-Qutayri
Khalifa University
Abu Dhabi, UAE
mqutayri@kustar.ac.ae

Jon Whittle
Lancaster University
Lancaster, UK.
j.n.whittle@lancaster.ac.uk

***Abstract*— Software-as-a-Service (SaaS) has recently been adopted by many organizations to get their work done through subscription-based services. To leverage economies of scale, software and hardware resources are shared among multiple tenants who have different requirements that rapidly change with time. Responding to tenants' diverse needs requires SaaS providers to carefully manage software variability so that every tenant feels like having a distinct instance of the application. Tenants' evolvable needs require the SaaS instance to dynamically adapt. This paper presents an integrated platform that facilitates the dynamic adaptation of Multi-Tenant Single Instance SaaS applications and supports runtime tenants' configurations customization. The proposed platform is based on three concepts: Service-Orientation, Software Product Lines (SPLs), and Model Driven Architecture (MDA). The proposed solution spans over two dimensions: Feature-level variability management and runtime variability management. We propose raising the level of abstraction in which the whole adaptation process is addressed to better manage customization. The feasibility of the approach is illustrated via a functioning prototype. A realistic SaaS application was used to exercise the different adaptation scenarios and evaluate the platform prototype implementation.**

***Keywords*— Cloud; SaaS; Multi-tenant Applications; Dynamic Adaptation; SPL**

I. INTRODUCTION

Cloud computing is a widespread model for sharing computing resources. It promotes economies of scale by hosting software applications on a network of remote servers, shared across multiple customers. From a business perspective, cloud providers typically charge customers using a "pay as you go" model [1] [2]. This means that customers pay for as much or as little of a software application as they want at any given time– this is a principle known as *elasticity* [3]. While financially appealing, cloud computing can lead to major technical challenges in managing software/hardware infrastructure because multiple customers usually require different configurations of an application.

Generally speaking, there are two models of cloud computing: multi-instance and single-instance. In the multi-instance model, each customer uses a separate instance of an application, which runs on a logically isolated hardware environment. In the single-instance model, all customers use the same instance of the software and hardware infrastructure. In the latter case, the cloud provider must take care to reveal to a given tenant only part of the application; the part that corresponds to the configuration requested by that tenant.

Whilst the multi-instance model is simpler both technically and conceptually, the single-instance model is preferred financially because it allows cloud providers to minimize resource requirements as the number of customers' increases. With this regard, Multi-tenant Single-instance approach helps in leveraging economies of scale. Instead of maintaining a single instance of the application per tenant, the hardware and software resources are shared among multiple tenants at the same time. Maintaining a single instance per multiple tenants is simpler and cheaper from the cloud provider's point of view. However, our approach is still applicable for the Single-tenant Multi-instance approach for an extra cost of maintaining a single instance per tenant.

However, different tenants may have different functional and non-functional requirements for the same application. An example of this variability would be two tenants requesting the same feature/service but with different data schemes that satisfy the tenants' desired level of security, availability and performance. In this case, decoupling data schemes from functionalities in Multi-tenant Single Instance SaaS would be required. In addition, the single instance needs to adapt at runtime to the changing requirements of the subscribing tenants. Ideally, such adaptation should be automated and carefully managed. On the one hand, the adaptation should be isolated. It needs to avoid affecting the running tenants at the expense of evolving the configurations of other tenants. For example, adaptation should not affect the zero-down time requirement guaranteed per tenant [4] as downtime can be very costly depending on the business type according to [5]. On the other hand, integrity of the running instance should be guaranteed to avoid affecting the expected QoS and to preserve the integrity of the service. For instance, an adaptation cannot interrupt a running customer request, unless otherwise stated. Finally, the consistency of the changes should be verified and guaranteed throughout the whole process. At any given time, the tenant application should be a valid instance.

To address this problem, we argue for raising the level of abstraction at which this variability is managed; we propose a framework for multi-tenant variability adaptation for service-oriented SaaS using the concepts of software product lines (SPLs) [6] and model-driven engineering (MDE) [7].

Our Framework models a Multi-Tenant Single Instance SaaS application as a set of running services and tenants' service compositions. Each service composition adapts to satisfy the tenant's changing requirements at runtime. *We aim at allowing service providers to create and adapt a single instance SaaS application, automatically and dynamically, for multiple tenants and guide the adaptation process to preserve the consistency of the running instance.* Thus, any request that may cause inconsistencies in the tenant instance is rejected from the beginning. We provide runtime management facilities to ensure the consistency of requested adaptations and the integrity of both, the running tenants and services. Interruptible and non-interruptible services are distinguished to guarantee the integrity of the running services' instances.

The proposed solution is based on SPL's commonality concepts where software artifacts are reused to derive different configurations for different tenants. This reduces the effort, time and money required to develop and maintain new configurations during the adaptation process. Moreover, MDE automates the process of deriving valid SPL configurations which eliminates the manual effort needed to validate the derived applications. In addition, SOA enables service sharing which also helps in eliminating the additional cost of developing a different service per user.

Away from code, SOA enables us to generate tenants' orchestrations based on services' compositions. These services can be shared by multiple tenants based on their interface usage. SPLs enables reasoning about adaptation decisions at a higher level of abstraction, feature level, before deploying at runtime. Our approach could be applied to different runtime environments without the need to change the modeling and reasoning modules of the platform.

Our approach spans over feature-level and service-level management activities to derive and dynamically adapt tenant-specific configurations. At feature-level, the platform helps in creating the common and variable architecture for the entire SaaS SPL. Then, at service-level this architecture (both feature model and mapping to services) is used by tenants.

The contribution of this research can be summarized as:

- Proposing an end-to-end framework for automatic and dynamic adaptation of Multi-Tenant Single Instance SaaS Applications. The adaptation is driven by: the changing requirements of tenants, the degradation of the instance QoS, and the integrity of services. Adaptations are managed to preserve the consistency of the instance.
- A systematic process for modeling customer configurations at feature-level and service-level.
- An automated end-to-end tool, which fully implements this process.

II. Background.

A. Software Engineering for Cloud Applications

Software product line engineering (SPLE) is the discipline that supports modeling commonality and variability among software family variants. It emphasizes on *Variability management* as a key principle of developing SPLs [8]. After identifying the set of common and varying features of SPL products, variability management is used to specify the dependencies among features and support their instantiations process [8]. Feature models are used to capture the commonality and variability of an SPL in the requirement and design stages [9] [10].

The development process of SPLs goes through two engineering stages: *domain engineer*ing and *application engineering*. *Domain engineering* refers to the process of domain analysis, which may include identifying components, methods or functionalities and the relationships among them to support the development process of an application in the domain [11]. The underlying infrastructure of a product line is developed through domain engineering. That is specifying the set of commonality and variability among the whole set of product variants. The artifacts of domain engineering include the set of core assets and SPL multiple-view architecture [12]. *Application engineering* focuses on constructing individual products based on the infrastructure produced using domain engineering [10]. This includes the process of integrating the shared core assets with varying functionalities distinguishing each individual product in a particular context [11].

B. Data Scheme in the Cloud: Definition

A data scheme is the type of data storage that can be selected by a tenant for every feature requiring data access in its configuration. A data scheme can vary between: a separate database, a shared database-shared schema, or a shared database - separate schema. A *separate database* scheme creates a new database per tenant and provides the highest security level among all the other types. Usually, a separate database is selected whenever security is a real concern. A *Shared database – shared schema* is another scheme in which multiple tenants share the same database and the same schema. In this case, the tenant's data is uniquely identified by the tenant ID. All the tenants selecting shared database – shared schema are affected by the whole database size in terms of database response time and performance.

A *shared database - separate* schema creates a new schema per tenant, which is, in fact, more secure and customizable than Shared database – shared schema.

Throughout the paper, we will be presenting a running example that is part of the case study that we conducted to validate our approach. It uses a realistic SaaS application that is provided by Tailspin, which is one of the companies known for using Microsoft technologies to provide cloud solutions [13]. Tailspin provides online services that enable customers to create customized surveys, publish them and collect their results for analysis. The application offers 20 features and three different data schemes (27 services in total). Two tenant organizations are presented to exercise the platform on different scenarios. ***Tenant 1*** has 10 features while **Tenant *2*** has 8 features. Both tenants have 5 core features while opting for different data schema.

III. Architecture and Methodology

A. Multi-Tenant Single Instance Definition

In this paper, we define a Multi-Tenant Single Instance SaaS application to be the composition of:

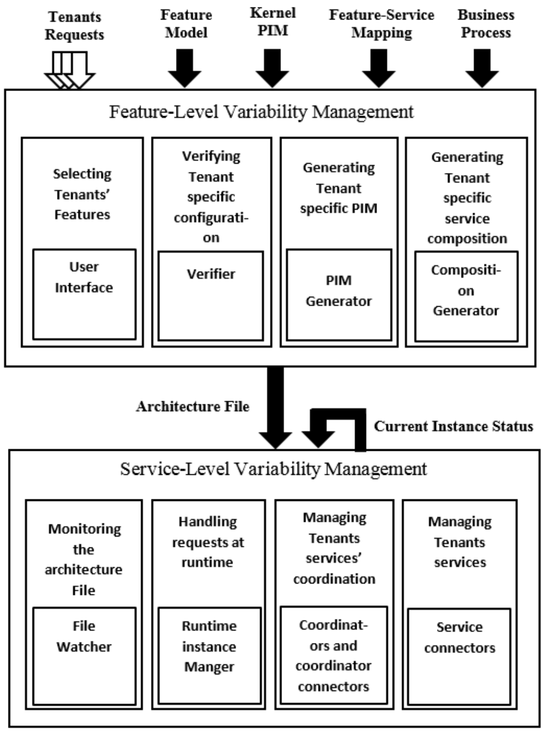
- The minimum set of services' instantiations that are needed to serve multiple tenants while preserving a QoS coefficient ϴ specified by the SaaS service provider based on the tenants' SLAs, and
- The set of tenants' service compositions.

The platform starts by instantiating a service per multiple tenants and it keeps on providing the same service instance to the new tenants until it reaches a certain ϴ that reveals the need for another service instantiation to preserve a certain QoS for the subscribing tenants. The QoS factor ϴ is assigned per service by a SaaS provider to guarantee a minimum QoS for all the subscribing tenants. Whenever the service instance's parameters drop below the specified ϴ, the service is re-instantiated to re-establish the desired quality of service value.

B. Platform Overview

Our approach spans over feature-level and service level variability management. The SPL feature model is designed based on the SaaS application general market requirements and specific tenants' requirements if they exist at this stage. Consequently, the commonality among features is identified through domain engineering. Then, tenants' specific applications are derived through application engineering. Each feature in the feature model is mapped to a service/set of services in underlying service architecture. A tenant configuration that consists of a set of features is mapped to a service composition by the end of this stage.

At runtime, service compositions are instantiated. A number of services start to run and interact with each other following tenants' requests. We adopt the adaptation patterns discussed in [14] [15], mainly to support the adaptation of tenants at runtime after their first deployment. These patterns describe how the architectural elements cooperate with each other, at runtime, to change the current software configuration to a new one based on the changing requirements of tenants. The patterns use State Machines, and adaptation interaction models.



Tenants Requests
Feature Model
Kernel PIM
Feature-Service Mapping
Business Process
Feature-Level Variability Management
Selecting Tenants' Features
User Interface
Verifying Tenant specific configurati-on
Verifier
Generating Tenant specific PIM
PIM Generator
Generating Tenant specific service composition
Compositi-on Generator
Architecture File
Current Instance Status
Service-Level Variability Management
Monitoring the architecture File
File Watcher
Handling requests at runtime
Runtime instance Manger
Managing Tenants services' coordination
Coordinat-ors and coordinator connectors
Managing Tenants services
Service connectors

Figure 1. Platform Main Activities

Figure 1 depicts the platform's main activities. The Feature Variability Management phase generates two main artifacts: a multi-tenant PIM (Platform Independent Model) that is kept as a record for what features are utilized by the instance, and an architecture file. The architecture file contains services and service references that specify how the services interact at runtime. In the Service Variability Management phase, a monitoring entity, the *File Watcher*, keeps observing the architecture file for any changes. If an adaptation is detected, the monitoring entity triggers an update and sends an adaptation request labeled by the tenant ID to the responsible *Coordinator Connector* to handle the request at runtime.

It should be noted that at the feature model level, the features requiring data access are annotated with <<data access>>. A feature-to-service mapping is used to identify services requiring data access.

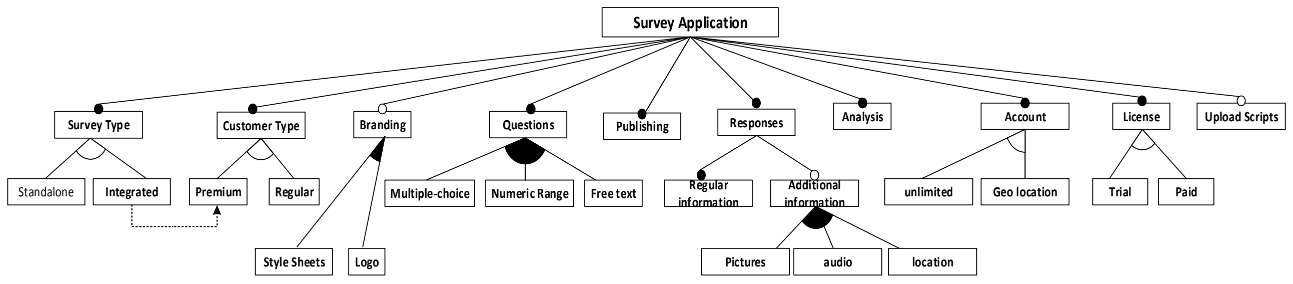


Figure 2. Survey Application Feature Model

When orchestrating services, a Coordinator injects data services in the services' workflow based on the annotated features and the data schemes selected by the tenant. Consequently, multiple tenants may have the same service in their configurations, however, the same service is given the accessibility to different data services based on tenants' different requirements. At the implementation level, tenants' data are stored in databases that are compatible with the chosen data schemes.

IV. Platform Description

This section provides a general description of the proposed solution. It discusses the feature-level and service-level variability management main activities.

A. Feature-Level Variability Management

The goal of this phase is to employ a feature model of a SaaS application to derive tenant-specific configurations, model service compositions, and construct the instance architecture that will be used during runtime adaptation. The SaaS SPL feature model is constructed first. Then customizable configurations are derived following the two SPLE activities: domain engineering and application engineering. Tenants specify their desired data schemes during the feature selection phase. After that, we use a feature-to-service mapping, and the data schemes selected earlier to generate tenant-specific service compositions. Finally, the SaaS instance architecture that includes all tenants' compositions is constructed; each of which has a defined set of services and service references that specify the services' interaction per tenant. The feature-level variability management activities serve as the grounding of the adaptation process at runtime. All the adaptations are validated, at runtime, based on the initial feature model and the mapped service architecture. After validation, adaptation steps are reflected on the architecture file of the instance before effecting changes on the running instance.

A.1. Domain Engineering: Constructing SaaS Feature Model

In SPLE, domain engineering is the phase where variability and commonality among the SaaS family are defined and the core assets are developed [11]. In our platform, this phase includes capturing the SaaS functional requirements at feature level by constructing a SaaS feature model. Then, specific non-functional requirements, such as data scheme, are captured during application engineering. Figure 2 shows the resulting Survey example feature model.

Consequently, the kernel (common) features are composed to obtain a kernel PIM. This is done via MDA executable transformation. The transformation takes the SaaS SPL feature model as an input, traverses it for kernel features, and returns a kernel PIM containing the kernel features of the SPL. This model will be merged later on with every tenant-specific configuration, during application engineering, to construct a valid SPL member; namely a tenant-specific PIM as we will detail next.

In the Survey example, the kernel PIM consists of Survey Type, **Customer Type**, Questions Type, **Publishing**, **Regular Info responses**, **Analysis**, **Account** and **License** features. The feature model identifies several features' dependencies where the existence of one feature depends on another. For example, to create **Integrated** surveys, a tenant should be a **Premium** customer; otherwise, he/she will be able to create **Standalone** surveys only. The following features require data access: **Customer Type**, **Analysis**, **Geo Location** and **Upload Scripts**. Let's further assume that the SaaS provider of the Survey application offers the three data schemes.

A.2. Application Engineering: Deriving Tenants' Specific PIMs

As mentioned previously, application engineering is the phase where a valid SPL member per tenant is derived. This process consists of integrating the kernel defined during domain engineering with the assets selected by the tenant. An onboarding tenant specifies its desired features via a feature selection phase. Tenant Feature Selection is the process where a tenant selects its desired features from the feature model along with the desired data schemes.

Unlike current state of the art that suggests a data scheme per instance or, a data scheme per tenant [16] [17] [18], our approach leaves the choice for the tenant to select the data scheme at the feature level. This means that a tenant may have different data schemes for different features in the same configuration. In our running example, some tenants may select storing their **Payment** information in a shared database with a separate-schema. Others, who are concerned about security and system performance, may request a separate database to keep their data fully isolated form other tenants.

The set of selected features, their mappings, and the associated data schemes are called a tenant configuration. Using another MDA transformation, the kernel PIM, derived earlier, is merged with the tenant configuration to obtain a tenant-specific PIM, which is a valid SaaS SPL member. An SPL member is considered as a valid member if and only if it satisfies the constraints defined in the feature model. For each generated tenant-specific PIM, we perform an assessment to ensure its validity. This process is necessary to guarantee the consistency of the derived configurations and the consistency of the requested adaptations. We have formulated several feature-level rules for such assessment. Table 1 describes two examples of these rules. For space limitation, we don't list all of the 14 rules. If the consistency checking rules reveal some contradictions, the tenant's adaptation request will be rejected since the requested configuration is not valid.

TABLE 1. FEATURE-LEVEL ADAPTATION RULES

Rule No.	Description
Rule 1	A feature requested by an onboarding tenant is added to the tenant-specific PIM if and only if the feature-level consistency of the tenant-specific PIM can be maintained. The tenant-specific configuration complies with the feature model constraints according to the following sub-rules: • At least one feature should be selected from an OR feature group. • One and only one feature should be selected from Alternative feature group. • Zero or more features should be selected from an Optional feature group. • Exclusive features can't be selected at the same time. • If a selected feature requires any other feature, the required features are added automatically to the configuration.
Rule 2	Committing changes to a tenant configuration entails updating the tenant-specific PIM and validating/re-validating it, updating the multi-tenant PIM and updating the architecture file

For example, let's say that tenant 1 has selected a shared database separate-schema, separate database and shared database for **Analysis**, **Premium** and **Upload Scripts** respectively. **Tenant 2** has requested a shared database separate-schema for all features requiring data access in its configuration.

A.3. Constructing Tenants Service Compositions

Generating a valid tenant-specific PIM triggers the construction of a new Service Coordinator. A *Coordinator* is an independent entity created for each tenant and assigned the role of a service orchestrator and a data services' injector. The Coordinator uses a feature-to-service mapping and a business process cut of the SPL to compose services based on the tenant configuration. The provided feature-to-service mapping maps each feature to its corresponding services (one-to-many). The application business process specifies the workflow of tasks that delivers a value for the subscripting tenants [19].

To know where to inject data services, we need to know which services require data access. In our approach, the features requiring data access are annotated at the feature level with data access stereotypes <<data access>>. Consequently, the Coordinator injects data services, gleaned from the feature-to-service mapping into the appropriate step in the workflow.

Back to our running example, features **Customer Type**, **Analysis**, **Geo Location** and **Upload Scripts** require data access. This implies that services **S1**, **S2**, **S3**, **S6**, **S7**, **S15**, **S18** and **S24** require data access according to the feature-service mapping. **Tenant 1** has selected a shared database separate-schema, separate database, and shared database for **Analysis**, **Premium** and **Upload Scripts** respectively. Every time a tenant requests to change its selected features or their respective data schemes, its Coordinator reconstructs the service workflow based on the new requirements. The new changes in the service flow are reflected in the multi-tenant PIM and the instance architecture file, which is fed back to runtime.

A.4. Generating the Architecture File

The architecture of the running instance is defined in terms of loosely-coupled services and service composition. Initially, the instance architecture is defined in an Architecture File at feature-level. The architecture file contains a set of composites each of which enclose a set of services and service references that describes how the enclosed services interact.

Every time a Service Coordinator generates/re-generates a new service composition, the services composing the orchestration along with the composition itself are added to the file. All the services in the architecture file are the ones that need to be activated at runtime. Service compositions specify how services interact with each other. If a tenant, for example, unsubscribe from its SaaS app, its service composition is removed from the architecture file. However, shared services used by the remaining tenants are not removed. Services are de-activated only if they are not used by any other tenants.

B. Service-level Variability Management

The goal of this phase is to manage the adaptation of the SaaS instance at runtime according to the adaptation requests. This necessitates monitoring the incoming requests to detect desired adaptations instantly and implement them accordingly and carefully to avoid runtime inconsistencies.

At runtime, the architecture described by the architecture file gets instantiated from a set of services that reference each other.

B.1. Detecting Changes

A file watcher is the entity responsible for monitoring and detecting changes in the architecture file caused by the evolving requirements of the subscribing tenants. The modifications detected represent the adaptations required to accommodate the changing requirements of the subscribing tenants. The file watcher identifies which services in which tenant composition are added/removed to/from the architecture file. The result of this analysis is passed to the instance manager, which directs all adaptation requests to the responsible entities.

B.2. Instantiating Adaptation Requests

Once the Instance Manager is notified about an architectural change, it forms an adaptation request with the updated service composition along with tenant ID and redirects it to the responsible Coordinator Connector to implement the requested adaptation. Once the Coordinator Connector receives the request, the runtime adaptation starts.

B.3. Managing Service Coordination at Runtime

A Coordinator is instantiated per tenant (one-to-one mapping) and is responsible for managing the coordination of tenant’s composition. Every Coordinator is assigned a Coordinator Connector that keeps the state of the Coordinator and buffers adaptation requests (one-to-one mapping). Some adaptation requests, for example, get buffered in the Coordinator Connector because the Coordinator is busy implementing a previous adaptation request. Therefore, for the same tenant, a new adaptation is never started unless the previous adaptation request is processed. Once the Coordinator is back to its idle state, its connector passes a new adaptation request, if any.

To evolve a tenant composition, the tenant’s Coordinator sends multiple requests for the Service Connector with the services to be added/removed/updated along with the requesting tenant ID and the type of the adaptation request, wither it is ADD, REMOVE or UPDATE service. It is important to clarify that tenants express the desired changes at the feature level then they get reflected at the service level. It is worth mentioning that the EXECUTE service requests, responsible for service invocation, get received directly by Service Connectors because they don’t result in architectural changes.

V. Prototype Implementation and Evaluation

In order to validate our approach, we implemented and evaluated a prototype of the platform.

A. Prototype Implementation

The prototype was implemented on Windows 7, 64-bit OS with 8.00 GB RAM, and Intel(R) Core (TM) i5-3230M, 2.60 GHz CPU. The tools/SDKs used are the following: Eclipse RCP Luna 4.4.0, Eclipse Modeling Framework SDK 2.10.1 (EMF), OCL Classic SDK 5.0.2, QVT Operational SDK 3.4.0, MySQL DBMS and Java OSGI stack.

The platform is built on top of iPOJO, which is an extension of Apache Felix framework, an implementation of OSGI specifications. The main element of iPOJO is a Component. A Component is an OSGI bundle that provides dynamic mechanisms at runtime. We have extended iPOJO with Composite entities. Each Composite includes a collection of components referencing each other. In addition, we have implemented a façade above iPOJO that allows the platform to use an explicit architectural model at runtime.

We start by constructing the SaaS SPL feature model, the business process, and the feature-service mapping. In previous research [12] [20], we have implemented the meta-models of the feature model and the multi-views model. Eclipse Modeling Framework (EMF) was used to define the meta-models as ECORE files.

Model-to-Model (M2M) transformations were developed and reused by the platform for each tenant to generate the Kernel PIM, tenant-specific PIMs, and the multi-tenant PIM. A *M2M transformation* transforms a model A that conforms to meta-model MA to a model B that conforms to meta-model MB, where MA and MB could be the same or different [35]. All M2M transformations carried out by the platform are specified using Query-View-Transformation (QVT) language, which is a standard language OMG transformation language.

For the sake of simplicity, each service in our SaaS application example is mapped to a Component in iPOJO. This implies that a feature can be mapped to a single or multiple Components at service-level each of which corresponds to a service fulfilling part of the feature functionality as specified by the feature-service mapping. In addition, we map each tenant to a Composite such that a Composite contains a set of components referencing each other based on the initial service composition specified at feature-level. Though OSGI, JAVA, and QVT were used, our solution is not limited to such technologies. The platform can be realized by other technologies.

B. Prototype Evaluation

We have used 24 services of the Survey case study [13] and we have simulated the onboarding, customization and removal of tenants with different configurations. Compared to similar research [21], where only 10 tenants have been used to evaluate their approach, we did the adaptation using 50 tenants, which is considered a more reasonable number that can be supported by real SaaS providers such as in [22].

B.1. Isolation of tenant Customization

To study the effect of customizing a tenant on the other tenants, we have considered 50 tenants that are up and running. Different requests to add/remove/update the same service are issued. Tenants' response times were recorded while evolving the running instance. Figure 3 illustrates the response times of the 50 tenants while no adaptation is taking place. It shows also the average response time for the 50 tenants while evolving Tenant 5 by removing and re-adding the same service to its composition. Evaluation results show that evolving Tenant 5 has no effect on the response time for the rest of the running tenants. However, the response time for the tenant requesting the adaptation has increased, an expected behavior since the execution of the upcoming requests have been buffered until the adaptation is processed. Other tenants, such as Tenant 30 and 50, may encounter some response time delays because of the requests queueing at service connector.

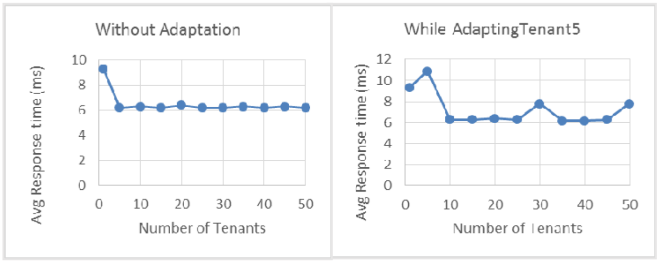


Figure 3. Service Response Time without/with Dynamic Adaptation

B.2. Adaptation Delays

We measure the average time needed to customize tenants using our platform. The experiment has been repeated ten times and average values have been represented.

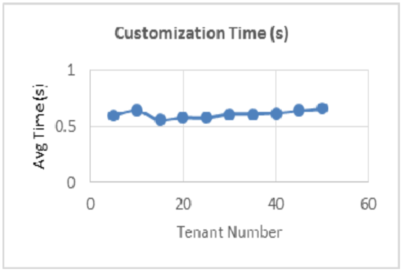


Figure 4. Tenant Customization Times

In this scenario, we have considered tenant customization. In fact, we may have two types of customization: feature customization, when a tenant adds a feature for example, and data schemes customization, when a tenant does not change any feature but decides to modify the data schema related to them. The time needed to perform a data schema customization cannot be measured in our current runtime prototype as the database handler is not yet implemented. Therefore, for this stage of evaluation, we only consider feature customization.

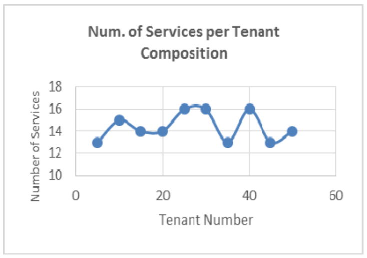


Figure 5. Number of Services per Tenant

We have measured the time needed to customize tenants by adding a service to their configurations (feature customization). Results showed that the average time to customize a tenant is 0.61 s. Unlike the onboarding and removal times, the customization time (Figure 4) is not affected by the number of services in the tenant configuration (Figure 5). It was noticed also that the time needed to customize a tenant by adding a service to the configuration doesn't change by increasing the number of subscribing tenants. The two factor affecting the customization time are the number of services to be customized and the current services' references that need to be rearranged after the addition of the new service. This explains the slight variation in the customization time need to add a single new service for each tenant in Figure 4.

VI. Related Work

Authors in [23] have introduced a Service Line Engineering (SLE) method to facilitate the development and customization of SaaS Multi-tenant Single Instance apps. The SLE method spans over four main phases: SL development, deployment, composition and configuration. Tenants' specific configurations are automatically transformed into software configurations. After deployment, they use versioning of features and SL artifacts to enable adaptation. This is similar to our work, however, their solution doesn't provide tools to support the actual adaptation and maintenance of SaaS applications.

In [24], the authors proposed a dynamic configuration framework for cloud applications based on extended feature models (EFM), view model (VM) and configuration process model (CPM). They defined a set of configuration management requirements for cloud applications. Moreover, they introduced stakeholders' views via staged configuration to define the operation that a stockholder can carry to the specified EMF. Their solution focuses on the problem space, however our approach provides an integrated approach that spans over the problem space and the solution space.

Authors in [21] proposed a model for realizing Multi-Tenant Single-Instance SaaS application using a service-oriented Dynamic Software Product Line (SO-DSPL). They rely on the use of service orchestration middleware to implement their approach. In our solution, the service orchestrations are generated and managed by the platform's service-level variability management modules.

In the surveyed literature, runtime management frameworks are not intensively specified and discussed. Most of the work concentrates on validating tenant configurations at the early stages. Another weakness is not having end-to-end tools and prototypes that illustrate the customization of tenants' lifecycle. We attempt to rectify these weaknesses in this paper.

VII. Conclusions and Future Work

In this paper, we have proposed a systematic approach to support dynamic adaptation of Multi-Tenant Single Instance SaaS applications based on feature-level and service level variability management. By using a refined definition of Multi-Tenant Single Instance SaaS applications in the context of service-orientation, our suggested methodology is able to facilitate SaaS single instance runtime adaptations by accommodating the changing requirements of the subscribing tenants at runtime without affecting the operational integrity of the running tenants or services. Adaptations are reflected first on the feature level, validated and then propagated to the service level. Evaluation results have showed that tenants' adaptation requests affect only the requesting tenants. However, the scalability of the platform can be affected by the service design. Services need to be designed as coarse-grained as possible without affecting the flexibility of the application to minimizes the number of services needed and maximizes their effect.

In the future, we aim at extending our platform to handle Feature Model Evolution. This evolution will depend on two important factors: the relevance of the new requirements to the existing ones and the number of tenants requesting them.

Acknowledgment

This work was funded by the Information and Communication Technology (ICT) fund, UAE.

References

[1] F. Durao, J. F. Carvalho, A. Fonseka and V. C. Garcia, "A systematic review on cloud computing," The Journal of Supercomputing, vol. 68, no. 3, pp. 1321-1346, 2014.

[2] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," Future Generation Computer Systems, vol. 25, no. 6, p. 599–616, 2009.

[3] E. F. Coutinho, F. Sousa, P. Rego, D. Gomes and J. Souza, "Elasticity in cloud computing: a survey," Annals of telecommunications, 2014.

[4] C.-P. Bezemer, "Performance Optimization of Multi-Tenant Software Systems," Technical University of Delft, 2014.

[5] A. Ganek and T. Corbi, "The dawning of the autonomic computing era," IBM Systems Journal, vol. 42, no. 1, pp. 5-18, 2003.

[6] K. Schmid and E. S. de Almeida, "Product Line Engineering," Software, IEEE, vol. 30, no. 4, pp. 24-30, 2013.

[7] "OMG Model Driven Architecture," [Online]. Available: http://www.omg.org/mda/. [Accessed 19 January 2014].

[8] M. Babar, L. Chen and F. Shull, "Managing Variability in Software Product Lines," Software, vol. 27, pp. 89-91, May-June 2010.

[9] M. Hinchey, S. Park and K. Schmid, "Building Dynamic Software Product Lines," Computer, vol. 45, p. 22 – 26, 2012.

[10] N. Bencomo, S. Hallsteinsen and E. Almeida, "View of the Dynamic Software Product Line Landscape," Computer, vol. 45, pp. 36-41, Oct. 2012.

[11] K. Pohl, G. Böckle and F. Linden, Software Product Line Engineering, Springer-Verlag Berlin Heidelberg, 2005.

[12] M. Abu-Matar, R. Mizouni and S. AlZahmi, "Towards Software Product Lines Based Cloud Architectures," 2013.

[13] "The Tailspin Scenario," Tailspin , [Online]. Available: https://msdn.microsoft.com/en-us/library/hh534482.aspx. [Accessed 16 April 2015].

[14] H. Gomaa, K. Hashimoto, M. Kim, S. Malek and D. Menascé, "Software adaptation patterns for service-oriented architectures," in 2010 ACM Symposium on Applied Computing, 2010.

[15] H. Gomaa and K. Hashimoto, "Dynamic Self-Adaptation for Distributed Service-Oriented Transactions," in ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), Zurich, 2012.

[16] L. heng, Y. dan and Z. xiaohong, "Survey on Multi-Tenant Data Architecture for SaaS," IJCSI International Journal of Computer Science Issues, vol. 9, no. 6, 2012.

[17] P. Kashyap and R. Singh , "Crypto Multi Tenant: An Environment of Secure Computing Using Cloud SQL," International Journal of Distributed and Parallel Systems (IJDPS), vol. 5, 2014.

[18] N. H. Bien and T. D. Thu , "Hierarchical multi-tenant pattern," in International Conference on Computing, Management and Telecommunications (ComManTel), 2014.

[19] C. Ouyang, M. Dumas, W. Van Der Aalst, A. Ter Hofstede and J. Mendling, "From business process models to process-oriented software systems," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 19, no. 1, 2009.

[20] S. Alzahmi, M. Abu-Matar and R. Mizouni, "A Practical Tool for Automating Service Oriented Software Product Lines Derivation," in Proceedings of the 8th international Symposium on service-Oriented System Engineering (SOSE '14), Oxford, UK, 2014.

[21] I. Kumara, J. Han, A. Colman, T. Nguyen and M. Kapuruge, "Realizing Service-based SaaS Applications with Runtime Sharing and Variation in Dynamic Software Product Lines," in Services Computing (SCC), 2013 IEEE International Conference, Santa Clara, CA, July 2013.

[22] "Techcello," [Online]. Available: http://www.techcello.com/product/features. [Accessed October 2015].

[23] S. Walraven, E. Truyen, K. Handekyn, W. Joosen and D. Landuyt, "The Efficient Customization of Multi-tenant Software-as-a-Service Applications with Service Lines," The Journal of Systems and Software , 2014.

[24] Y. Liu, B. Zhang, G. Liu, M. Zhang and J. Na, "Evolving SaaS Based on Reflictive Petri Nets," in 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution, Solvenia EU, 2010.