Architectural Design of a Deployment Platform to Provision Mixed-tenancy SaaS-Applications

Matthias Reinhardt[1], Stefan T. Ruehl[2], Stephan A. W. Verclas[3], Urs Andelfinger[1] and Alois Schütte[1]
[1]*University of Applied Sciences Darmstadt, Haardtring 100, 64295 Darmstadt, Germany*
[2]*Clausthal University of Technology, Albrecht-von-Groddeck-Str. 7, 38678 Clausthal-Zellerfeld, Germany*
[3]*T-Systems International GmbH, Heinrich-Hertz-Str. 1, 64295 Darmstadt, Germany*

Keywords: Software Architecture, Software Design, Web and Internet Services.

Abstract: Software-as-a-Service (SaaS) is a delivery model whose basic idea is to provide applications to the customer on demand over the Internet. In contrast to similar but older approaches, SaaS promotes multi-tenancy as a tool to exploit economies of scale. This means that a single application instance serves multiple customers. However, a major throwback of SaaS is the customers' hesitation of sharing infrastructure, application code, or data with other tenants. This is due to the fact that one of the major threats of multi-tenancy is information disclosure due to a system malfunction, system error, or aggressive actions by individual users. So far the only approach in research to counteract on this hesitation has been to enhance the isolation between tenants using the same instance. Our approach (presented in earlier work) tackles this hesitation differently. It allows customers to choose if or even with whom they want to share the application. The approach enables the customer to make that choice not just for the entire application but specifically for individual application components and the underlying infrastructure. This paper contributes to the mixed-tenancy approach by introducing a software pattern that allows to establish communication between Application Components that are deployed following the mixed-tenancy paradigm. The contributions of this paper are evaluated based on a representative example that employs all possible kinds of communication.

1 INTRODUCTION

Software-as-a-Service (SaaS) is a delivery model whose basic idea is to provide applications to the customer on demand over the Internet. SaaS promotes multi-tenancy (MT) as a tool to exploit economies of scale. This means that a single application instance serves multiple customers. However, even though multiple customers use the same instance, each of them has the impression that the instance is designated only to themselves. This is achieved by isolating the tenants' data from each other (Chong and Carraro, 2006).

In contrast to single-tenancy, MT has the advantage that IT-Infrastructure may be used most efficiently as it is possible to host as many tenants as possible on the same instance. Thus, operational and maintenance cost of the application is decreased. However, one of the major threats of MT applications is information disclosure due to data breach (Bezemer and Zaidman, 2010; Cloud Security Alliance, 2013). This may occur through a system error, malfunction, or destructive action. So far this problem has only been tackled by proposing new approaches to implement and improve the tenant isolation on a single instance.

The approach presented in (Ruehl et al., 2012; Ruehl et al., 2013), however, is different as it strives to solve the problem by finding a hybrid solution between multi-tenancy and single-tenancy. We refer to this approach as *mixed-tenancy*. The approach tries to emphasize both the customers' concerns about sharing infrastructure as well as the operator's desire to utilize infrastructure as efficiently as possible.

The approach that is supposed to answer these questions starts by building an MT application based on the concept of Service-oriented Architecture (SOA). This means the so-called composite SaaS-applications are composed of a number of application components (AC) that each offer atomic functionality. The approach enables the customer to choose if or even with whom they want to share a specific AC

Reinhardt M., Ruehl S., Verclas S., Andelfinger U. and Schütte A..
Architectural Design of a Deployment Platform to Provision Mixed-tenancy SaaS-Applications.
DOI: 10.5220/0004813803950402
In *Proceedings of the 4th International Conference on Cloud Computing and Services Science* (CLOSER-2014), pages 395-402
ISBN: 978-989-758-019-2

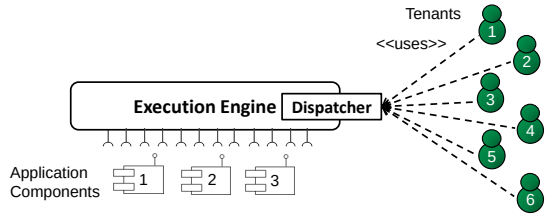

Figure 1: Architectural overview.

and the underlying infrastructure stack. This way customers may declare their requirements toward a deployment of their application variant.
As an example please refer to Figure 1. It illustrates a setting in which six tenants are using an application that is composed of three ACs. Example requirements that would be possible are:

1. AC 1 is not handling any critical data, thus all tenants feel comfortable to share instances and the underlying Virtual Machines.
2. AC 2, however, handles semi-sensitive data. Thus, tenant 2, for example, only wants to share an instance with European companies but not with competitors. However, on a Virtual Machine (VM) level, tenant 2 agrees to share with all customers that are not competitors.
3. AC 3 is handling very sensitive data. Thus, tenants will require quite restrictive deployments. Tenant 4, for example, stated that they do not wish to share an instance with competitors and only with companies from the USA.
4. Tenant 6 is special in the sense that they demand a single instance for their private use of every AC and VM.

In (Ruehl et al., 2013) a semantic model is presented that allows customers to describe their constraints towards a mixed-tenancy deployment. Based on this model the Deployment Configuration Generator (DCG) may be used to capture customer constraints. Furthermore, once the constraints were added into the DCG, a deployment is calculated that applies to all expressed customer constraints and is optimal according to resource consumption. This deployment is described in a transfer document that we refer to as deployment configuration (DC). This will be discussed in detail by Section 3. Based on the DC, a given multi-tenancy enabled composite application may be deployed according to the mixed-tenancy paradigm.
This paper's contribution is a software pattern that may be applied to establish communication between ACs deployed in a mixed-tenancy environment. The pattern shall be able to solve the special challenges that appear in such an environment. First of all, this pattern shall be able to enforce the customer given constraints for the data communication between components. And secondly, the pattern promotes the possibility to migrate existing multi-tenancy applications into a mixed-tenancy environment without having to change the application.

In order to do that the paper is structured as follows. After a discussion of related work, the paper will start with an analysis of requirements that architecture of the mixed-tenancy platform has to meet. The paper continues by developing the pattern to establish communication. This is done by first discussing existing patterns with respect to the defined requirements, and combine those to satisfy the requirements. In order to be capable of evaluating the communication pattern, it is necessary to develop a mechanism that can automatically deploy an application according to a DC. This is presented by Section 5. Section 6 will conduct an evaluation of the presented pattern by analyzing specifically built scenarios that represent a wide variety of possible cases.

The paper concludes with summarizing the results and drawing a conclusion.

2 RELATED WORK

SaaS in general as well as the configuration issues and challenges related to it were explored in (Sun et al., 2008). The aspect of functional variability has been discussed in some work. As, for example, in (Mietzner et al., 2009) the author discusses bringing such flexibility to the process layer of process-based, service-oriented SaaS-applications. This is done by creating variability descriptors that are transformed into a BPEL process model. In (Lizhen et al., 2010) an approach is presented to describe variability for SaaS-applications. This approach can systematically describe variability points and their relationships, and it assures the quality of the configuration inputs made by the customers. This work focuses on the creation of descriptions of functional variability. Furthermore, there are approaches with the goal to realize MT that meets privacy and performance requirements. In (Guo et al., 2007), for example, a framework for multi-tenant aware SaaS-applications including data isolation, performance isolation or configuration is described. Our approach, in contrast, allows the customer to give input for how and with whom their ACs are deployed. Data security is achieved as tenants do not share the same AC instances. To the best of our knowledge there is no similar approach. In addition, approaches that realize tenants' isolation on an MT instance may be applied.

3 REQUIREMENTS ANALYSIS

This section's purpose is to discuss the necessary requirements towards realizing a mixed-tenancy platform. The platform consists of two major tasks. The first is the automatic deployment, whose job is to deploy the application according to the description of the DC. A prototypical implementation of the deployment task will be discussed in section 5. This may require that multiple VMs need to be created and ACs, that make up the application, will be instantiated multiple times. The DC is the output of the aforementioned DCG and describes the deployment that is to be realized. Thus, the DC defines how many instances of a specific component shall be instantiated on a particular server and which tenants are allowed to access them. It captures a deployment that applies to all constraints of all tenants using the application. Additionally, the DC was created to be optimal according to resource consumptions in order to satisfy the service providers' need for efficient use of infrastructure. Figure 2 illustrates the structure of the DC in Entity-Relationship Notation. The figure captures the relationship between the important entities (Servers, ACs, ACIs, Tenants and Users). These entities have been included in an XML-schema, from which a valid DC can be created by defining an XML-file which refers to the schema. This allows us to decouple the DCG from the rest of the platform.

The second important task of the platform is the communication. This task of the platform is in charge of establishing communication between the deployed component instances. Since it is our aim to keep the adaptation progress of an application at a minimum, one requirement of the communication task is the separation of the application logic and the platform logic. This may be realized by introducing a new layer above the original application logic. This platform logic layer manages the communication. Furthermore, it is important, that the platform's communication applies to the customers' wishes. If a tenant denies the common deployment with another tenant, the platform has to make sure that this can happen under no circumstances. Hence, it has to be avoided, that the data communication of two tenants, who may are competitors, is redirected through a common point of intersection. This is the second requirement that needs to be fulfilled by the communication pattern. It is based on the respect for the customers' wishes. If a customer denies to share a component with his competitor, it is not legitimate to redirect both their data traffic through a global interstation. Another reason is a possible high amount of data traffic which could slow down the running application.

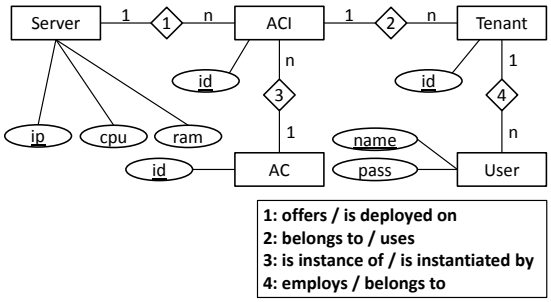

Figure 2: Structure of Deployment Configuration (ERM).

4 PATTERN SELECTION

The main goal of this paper is to develop a proper design pattern that allows to establish communication by realizing the prerequisites, that were introduced in the last chapter. To achieve this, we investigate well-established design patterns as well as intuitive approaches. We chose a simple example of a DC which contains a special case. In the following, the example will be applied to every pattern. This allows to illustrate how the platform would behave in common situations as well as in the special case. Additionally, it will be possible to compare the individual results in order to select the most appropriate pattern. The comparison is shown in Figure 3 via visualizations of the structure and the sequence (UML sequence diagrams).

The example consists of two tenants who are using an application with three components. There are five application component instances (ACI) where their border color indicates the tenant belonging. The regular case is demonstrated at component 1 and 3 where both tenants use exclusively one instance. As to component 2 there is only one instance which is shared between tenant 1 and 2. As discussed in the last section it is important that the platform maintains decentralized communication and separation of the platform logic in both cases. Another element which exists in all diagrams is the Execution Engine (EE). It belongs to the platform and has the knowledge of all component-instance locations and tenant belongings. It may fulfill additional tasks depending on the specific patterns it is used in.

The first pattern we came up with we called **delegator**. Within this pattern the EE's job is to delegate communication. When a user or a component instance wants to use a service of another component it asks the delegator which instance of that component belongs to the current tenant. This control communication is represented by the dashed arrows in Figure 3. The EE responds with the whereabouts in form of

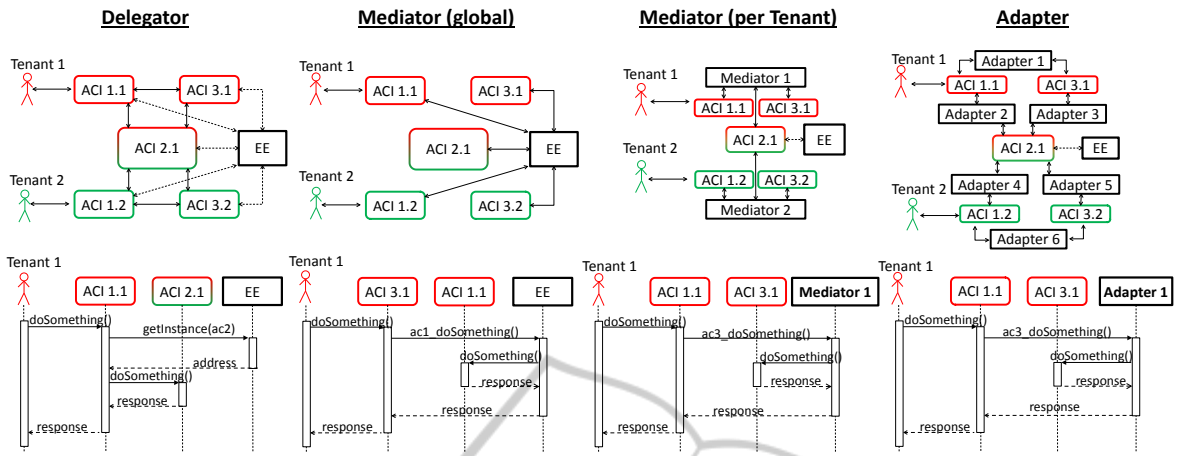

Figure 3: Overview of different Patterns to tackle the Communication Challenge.

an address. Thereby a contact can be established between the two instances or the user and the instance. This direct communication is represented by the solid lines. Changes of the locations are no issue, because the contact to the EE will happen before every function call of another component instance.

With respect to the requirements, there is a problem with the separation of the platform logic. The single instance of component 2 knows multiple instances of a component because it is shared by multiple tenants. Thus, we need platform logic in the component, which is not desired because it complicates the adaptation process of an application to the platform. However, the pattern meets the requirement for decentralized communication that applies to customers' constraints.

Secondly, we engaged in with the **mediator** pattern provided by Gamma et al. (Gamma et al., 1994). The Mediator is an object which encapsulates the interaction between an amount of objects. Thus, the objects are able to communicate without directly connecting to each other. Through our investigation two ways to implement the pattern into the platform emerged – global and concrete mediator.

The **global mediator** is characterized by a single mediator object which is represented by the EE. There are solid lines in Figure 3 between the component instances and the mediator. This means that all data communication between the component instances passes through the global mediator, unlike the delegator pattern where the data communication happens directly between the component instances.

With the use of the global mediator, there is no communication logic inside the components. They just have to be able to communicate with the EE. Thus, there is no problem with the special case of component 2, where one instance is used by two tenants at once. But because of the passing of the complete data communication through the EE, we detected a problem with the other requirement. If the customer requested the separation to another customer, it is not legitimate to redirect both their data communication through a common point of intersection, which, unfortunately, is the case here. Another negative aspect is a possible overload of the global mediator in a larger scenario, which could slow down the system as the global mediator becomes a bottleneck for all communication.

Another way to adapt the mediator pattern is shown by Figure 3 by the use of multiple mediator objects, one **mediator per tenant**. In contrast to the global mediator the data communication which passes a mediator is related to the same tenant. Because of this, we have successfully respected the requirement of decentralized communication.

In case of component 2 there exists only one instance used by both tenants. The instance has to decide which mediator to contact if it wants to call a function of another component. Clearly, there is communication logic necessary inside the components which stands in conflict with the second requirement.

Another pattern we ported onto the platform is the adapter pattern which was also given by Gamma et al. (Gamma et al., 1994). Usually, it will be applied to establish communication despite incompatible interfaces. For example, when third-party components should be included into a software without the possibility to adjust the interfaces, an adapter is an appropriate solution. In figure 3 there exists an adapter between every two component instances that will eventually need to communicate. Such adapters encapsulate the functionality of their related component instances.

In this setup we follow the requirement of de-

centralized communication. The data communication passing an adapter is related to the same tenant. With respect to the second requirement we have noticed a violation in the case of component 2. A common used component instance must decide which adapter to address. This decision has to be made depending on the current accessing tenant. Thus, communication logic is necessary inside the components.

So far none of the investigated patterns have matched both previously defined requirements. On this account we created a new pattern resulting the positive aspects of the previous patterns. It is called Connector because of its behavior. It has no relation to other homonymous design patterns which can be found in literature (e.g. Acceptor and Connector (Schmidt, 1995)).

Figure 4 describes the structure of the pattern. For every component instance there exists a related connector element. The component instances are solely able to initiate a connection to their connector. This is geared to the global adapter pattern, where there also was no platform logic located in the component instances. In contrast to the global mediator, this approach is distributing the platform intelligence into multiple connector elements and not into a single object like a mediator. Besides, there is a direct communication between the connector-elements. This works just like in the delegator pattern.

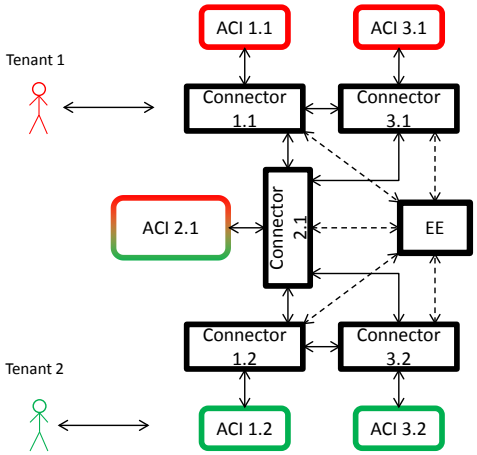

Figure 4: Example of the connector pattern.

In Figure 5 it is demonstrated how the communication works. If a component instance respectively a user wants to retrieve data from another component, it is calling the specific function at the related connector. Thus, here is no platform logic necessary. The component instance treats the function just like a local one. The connector is now handling the establishment of the connection. The EE acts just like the delegator element. The only difference is that the connector is requesting the address or the reference of the corresponding connector of the target component instance - not of the component-instance. After the actual function call took place, the result will be returned to the initial calling component instance. As illustrated by Figure 5, a connector encapsulates the needed functions of all other components for the related component instance and the provided functions to be called by the other component instances.

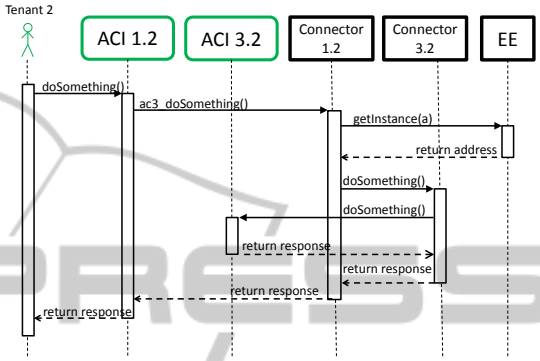

Figure 5: Sequence diagram of the connector pattern.

The pattern fits the requirements. As already explained in the last section, we have achieved a decentralized communication and the separation of platform logic. Because of this, we elect the connector pattern to the best way for solving the discussed problem. A summary of the investigated patterns and their ratings is shown in Table 1.

Table 1: Overview of the results of pattern analysis.

Pattern	Decentralized Communication	Separation of Platform Logic
Delegator	✔	✗
Mediator (global)	✗	✔
Mediator (per tenant)	✔	✗
Adapter	✔	✗
Connector	✔	✔

5 PROTOTYPE REALIZATION

Since we were able to identify a mixed-tenancy architecture, there is the need to prove its functionality and practicability. The intent of this section is the description of a case-study which has been realized using the connector pattern. This provides a base for an evaluation of the architecture in the next section. This

section addresses the requirements of the deployment task which have been defined in Section 3.

The main focus of the project relied on how to automatically deploy multiple instances of different components, which afterwards shall communicate with each other. In Figure 6 you can see the automatic deployment of an exemplary mixed-tenancy application. There is a basic VM that is in charge of the deployment. At first the deployment of multiple target VMs starts, which is followed by the deployment of application component instances (ACI) and the related connector elements. The locations and number of deployments executed by the EE are given. They result from the customers' requirements.

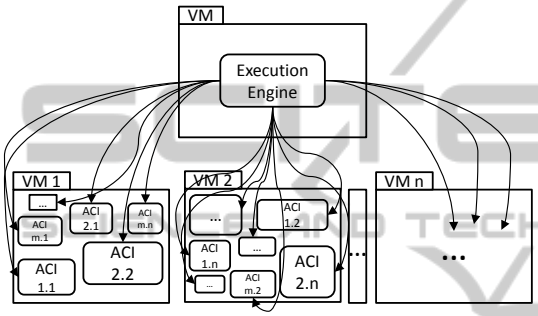

Figure 6: Deployment of application components.

The idea was to develop a mixed-tenancy platform. This means, that it is possible to attach existing multi-tenancy applications to the platform in order to make it mixed-tenancy capable. Thus, we realized the deployment task essentially using VMware vSphere[1] technolog and GlassFish[2] application servers. We used the VMware Virtual Infrastructure Java API (VI java)[3], which is an open source project initiated by VMware. The advantages against the official VMware vSPhere Management SDK are its higher performance and easier handling (Jin, 2013).

The ACs are encapsulated in web archives (WAR-files) and located at the same VM as the EE. During the execution process of a DC, these WAR-files will be duplicated and modified before they will automatically be uploaded and deployed on the target VM. To deploy an ACI we used the GlassFish asadmin tool. To allow or deny the access to an ACI we set the deployment descriptor of the WAR-file according to the information of the submitted DC. The modification of each concrete ACI is necessary, because the deployment constructor consists of many entries that cannot be identical if two ACIs should be deployed on the same server. This way each ACI is unique and can be identified.

The restricted access of AC instances gains security benefits over a pure multi-tenancy deployment. This is due to the fact that access restrictions to different instances of the same AC increases tenants' isolation.

The communication between the connector elements is realized via REST interfaces. To transmit the necessary number of VMs and ACIs we have defined a specific XML schema. A document following this schema contains a number of servers and instances of each component of the application. Furthermore, for each instance it is declared which tenants are allowed to access it. The implementation of SSL encryption would be a further step towards increasing tenants' isolation. This will be tackled in future.

6 EVALUATION

Based on the realized prototype of a platform introduced in the previous section, it is now possible to evaluate the planned mixed-tenancy architecture. Obviously, the platform is not runnable on its own, as it is supposed to deploy and run a multi-tenancy application. Because of this, we need to develop a dummy multi-tenancy application and port it onto the platform. We made sure, that the application comprises all possible special cases as well as the regular cases. This means that it is built op of multiple components that interact with each other directly or indirectly. An indirect interaction occurs whenever a component instance or a tenant advises a second component instance to call a function of a third component instance. When it comes to the properties of mixed-tenancy, we need to test a row of things.

In order to test the functionality of the developed platform, we made sure that all possible forms of customer constraints will be deployed without malfunctions. At first we have to distinguish between the server and the component layer. At both layers there exist multiple isolation levels.

At first we describe the levels at the component layer. *Shared* means that there is only one instance of the component, which is used by multiple tenants simultaneously. *Separate* on the contrary represents single instances of the component owned by each tenant. At last there is a level called *mixed* which comprises all remaining cases. At this level, there is at least one component instance that is used by more than one but not all existing tenants.

Regarding the server level, this is quite similar. *Shared* means that all tenants share one server where all component instances are deployed. At the *separate*

[1]http://www.vmware.com/de/products/vsphere/
[2]https://glassfish.java.net/
[3]http://vijava.sourceforge.net/

level each tenant needs a single server where all his component instances are deployed. *Mixed* means that at least two tenants are using component instances on the same server.

After the identification and the classification of all realistic cases, we combined the server and component layer. Since each of the layers has three levels, it is resulting in 9 permutations. Three of the 9 permutations are not realistic, e.g. if two tenants share one component instance, they cannot possess their own servers. This finally leads to 6 possibilities.

All test cases that belong to the same of the 6 resulting classes are equivalent. This means that we just need to test one to prove its functionality. For example, if we can successfully deploy a shared instance of component A on a server, then this result represents the operativeness of the shared deployment of component B. Thus, we elected one test case for each of the 6 classes. For each one we have tested direct and indirect access of the deployed component instances and verified the effect was as anticipated.

Due to the fact that all our test cases run through with success, we can say that our realized mixed-tenancy architecture which is embodied by the connector pattern is a working solution for the discussed problem. The evaluation has proven that our defined prerequisites were successfully enforced for the dummy application.

However, there are throwbacks coming from the utilized pattern. First of all performance will be decreased. This is due to the communication behavior, where ACs do not communicate directly with each other anymore. Instead communication is handled by connectors. This causes overhead. Secondly, if an operator decides to host a multi-tenancy application following the mixed-tenancy approach, it is necessary to create the connectors for all application components involved. The effort necessary to do that could only be minimized if it would be possible to automatically generate connectors through code inspection.

7 SUMMARY AND CONCLUSION

In the first chapter we introduced this paper with the characteristics of multi-tenancy as well as the advantages and disadvantages of cloud providers and their tenants. We explained why there is a demand for a better solution when it comes to the client's privacy needs and the resulting consequences for the provider. For example, when a client does not want to share a part of an application with his competitor, the provider needs to deploy the whole application twice, which leads to higher costs.

After that we introduced the concepts of mixed-tenancy as a possible solution. In order to verify these concepts as a solution we developed a prototypical platform which is capable of deploying multi-tenancy applications in the form of mixed-tenancy. The main goal of this paper was to identify or design an architecture for this platform. After defining several requirements, we investigated common design patterns regarding their applicability. This leads to the connector pattern which is a combination of the previously analyzed patterns. Since it matched the requirements, we selected the connector pattern as mixed-tenancy architecture.

Due to an evaluation of the realized platform, we were able to make sure that it is possible to port simple multi-tenancy applications onto the platform. During the porting process it was obvious that the connector pattern separates the platform logic from the application logic as well as it decentralizes the data communication.

Furthermore, we successfully verified that none of the tenants can access a component instance that does not belong to him. Through carefully selected test cases, we made sure that all kinds of DCs were covered.

However, these results come with a price. The pattern decreases the performance of an application since it produces an overhead in the communication among ACs. Furthermore, it is required that for every AC a connector is created. That creates an additional effort for an operator that wants to deploy a multi-tenancy application following the mixed-tenancy paradigm.

The main goal in the future should be the realization of a stable mixed-tenancy deployment platform. It shall be able to migrate an existing real-world multi-tenancy application onto this platform. For this paper we only evaluated an example application that realizes certain styles of communication. Due to its logical partition in a few components, it turned out to be a good test application and it was possible to gain indications that mixed-tenancy may be realized. However, in order to gain further conclusions and the applicability of the mixed-tenancy approach in real-world scenarios, it will be necessary to evaluate the migration of a real application.

Although, the focus of this paper was not based on performance, we aspire to optimize it in future research. Thus, a detailed performance analysis is still open for future research. So far the only security measures that have been implemented are basic authentication and user impersonation. The mixed-tenancy concept allows to deploy further security measures to isolate tenants even further (e.g. on a network level). However, this has not been investigated further yet,

as it was our goal to create a working platform with initial security.

Finally, one additional point that is still open for future research is the platform's lifecycle. In a real-world environment it is highly desired to have a environment that is both scalable and elastically adaptive. As a first step these characteristics were not addressed by this paper but bear great opportunities for future research.

REFERENCES

Bezemer, C.-P. and Zaidman, A. (2010). Multi-tenant SaaS applications: maintenance dream or nightmare? In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, IWPSE-EVOL '10, page 88–92, New York, NY, USA. ACM.

Chong, F. and Carraro, G. (2006). Architecture strategies for catching the long tail. *Microsoft MSDN*.

Cloud Security Alliance (2013). The notorious nine: Cloud computing top threats in 2013. Technical report.

Gamma, E., Helm, R., and Johnson, R. E. (1994). *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman, Amsterdam, 1st ed., reprint. edition.

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

Jin, S. (2013). VMware infrastructure (vSphere) java API. http://vijava.sourceforge.net/faq.php.

Lizhen, C., Haiyang, W., Lin, J., and Pu, H. (2010). Customization modeling based on metagraph for multi-tenant applications. In *5th International Conference on Pervasive Computing and Applications*, pages 255–260, Maribor, Slovenia.

Mietzner, R., Metzger, A., Leymann, F., and Pohl, K. (2009). Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In *Workshop on Principles of Engineering Service Oriented Systems, ICSE*, volume 0, pages 18–25, Los Alamitos, CA, USA. IEEE Computer Society.

Ruehl, S. T., Andelfinger, U., Rausch, A., and Verclas, S. A. (2012). Toward realization of deployment variability for software-as-a-service applications. In *2012 IEEE 5th International Conference on Cloud Computing (CLOUD)*, pages 622 –629.

Ruehl, S. T., Wache, H., and Verclas, S. A. W. (2013). Capturing customers requirements towards mixed-tenancy deployments of SaaS-Applications. In *2013 IEEE 6th International Conference on Cloud Computing (CLOUD)*, pages 462 –469.

Schmidt, D. C. (1995). Acceptor and connector: A family of object creational patterns for initializing communication services. In *In Proceedings of the European Pattern Language of Programs conference*, pages 10–14.

Sun, W., Zhang, X., Guo, C. J., Sun, P., and Su, H. (2008). Software as a service: Configuration and customization perspectives. In *Services Part II, IEEE Congress on*, volume 0, pages 18–25, Los Alamitos, CA, USA. IEEE Computer Society.