Institute of Architecture of Application Systems

Native Cloud Applications:
Why Virtual Machines, Images and Containers Miss the Point!

Frank Leymann, Christoph Fehling, Sebastian Wagner, Johannes Wettinger

Institute of Architecture of Application Systems,
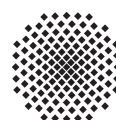University of Stuttgart, Germany,
lastname@iaas.uni-stuttgart.de

Universität Stuttgart
Germany

Native Cloud Applications:
Why Virtual Machines, Images and Containers Miss the Point!

Frank Leymann, Christoph Fehling, Sebastian Wagner and Johannes Wettinger

Institute of Architecture of Application Systems, Universität Stuttgart, Universitätsstraße 38, Stuttgart, Germany

{leymann, fehling, wagner, wettinger}@iaas.uni-stuttgart.de

Abstract: Due to the current hype around cloud computing, the term "native cloud application" becomes increasingly popular. It suggests an application to fully benefit from all the advantages of cloud computing. Many users tend to consider their applications as cloud native if the application is just bundled in a virtual machine image or a container. Even though virtualization is fundamental for implementing the cloud computing paradigm, a virtualized application does not automatically cover all properties of a native cloud application. In this work, we propose a definition of a native cloud application by specifying the set of characteristic architectural properties, which a native cloud application has to provide. We demonstrate the importance of these properties by introducing a typical scenario from current practice that moves an application to the cloud. The identified properties and the scenario especially show why virtualization alone is insufficient to build native cloud applications. Finally, we outline how native cloud applications respect the core principles of service-oriented architectures, which are currently hyped a lot in the form of microservice architectures.

1 INTRODUCTION

Cloud service providers of the early days, such as Amazon, started their Infrastructure as a Service (IaaS) cloud business by enabling customers to run virtual machines (VM) on their datacenter infrastructure. Customers were able to create VM images that bundled their application stack along with an operating system and instantiate those images as VMs. In numerous industry collaborations we investigated the migration of existing applications to the cloud and the development of new cloud applications (Fehling *et al.*, 2013; Fehling *et al.*, 2011; Brandic *et al.*, 2010). In the investigated use cases we found that virtualization alone is not sufficient for fully taking advantage of the cloud computing paradigm.

In this article we show that although virtualization lays the groundwork for cloud computing, additional alterations to the application's architecture are required to make up a "cloud native application". We discuss five essential architectural properties we identified during our industry collaborations that have to be implemented by a native cloud application (Fehling *et al.*, 2014).

Based on those properties we explain why an application that was simply migrated to the cloud in the form of a VM image does not comply with these properties and how the application has to be adapted to transform it into a native cloud application. These properties have to be enabled in any application that is built for the cloud. Note that we provide a definition of native cloud applications by specifying their properties; we do not aim to establish a migration guide for moving applications to the cloud. Guidelines and best practices on this topic can be found in our previous work (Andrikopoulos *et al.*, 2013; Fehling *et al.*, 2013).

Section 2 introduces a reference application that reflects the core of the architectures of our industry use cases. Based on the reference application, Section 3 focuses on its transformation from a VM-bundled to a native cloud application. We also discuss why virtualization or containerization alone is not sufficient to fully benefit from cloud environments. Therefore, a set of architectural properties are introduced, which a native cloud application has to implement. Section 4 discusses how native cloud applications are related to microservice architectures, SOA, and continuous

delivery. Furthermore, Section 5 discusses how the reference application itself can be offered as a cloud service. Finally, Section 6 concludes the article.

2 REFERENCE APPLICATION

Throughout the article, the application shown in Figure 1 is used as running example for transforming an existing application into a cloud native application. It offers functionality for accounting, marketing, and other business concerns. The architecture specification of this application and the following transformation uses the concept of layers and tiers (Fowler, 2002): the functionality of an application is provided by separate components that are associated with logical layers. Application components may only interact with other components on the same layer or one layer below. Logical layers are later assigned to physical tiers for application provisioning. In our case, these tiers are constituted by VMs, which may be hosted by a cloud provider.
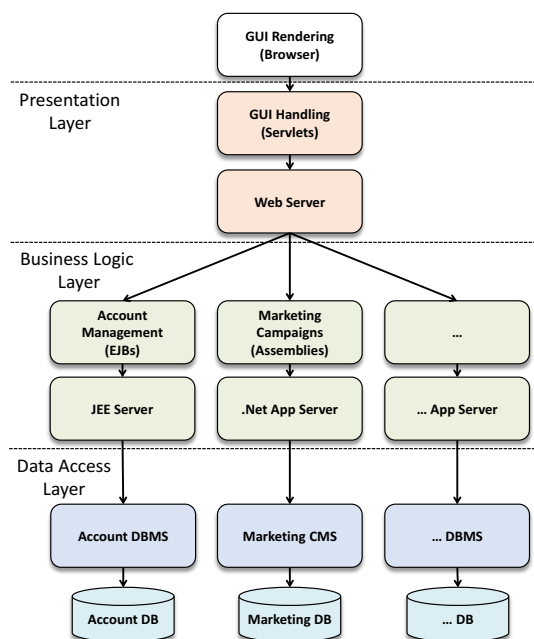

Figure 1: Reference Application to be Moved to the Cloud

The reference application is comprised of three layers. Each layer has been built on different technology stacks. The accounting functions are implemented as Enterprise Java Beans[1] (EJB) on a Java Enterprise Edition (JEE) server making use of a Database Management Systems (DBMS); the

marketing functions are built in a .Net environment[2] using a Content Management System (CMS). All application functions are integrated into a graphical user interface (GUI), which is realized by servlets hosted on a Web server.

The servlet, EJB and .Net components are *stateless*. In this scope, we differentiate: (i) *session state* – information about the interaction of users with the application. This data is provided with each request to the application and (ii) *application state* – data handled by the application, such as a customer account, billing address, etc. This data is persisted in the databases.

3 TRANSFORMING THE REFERENCE APPLICATION TO A CLOUD NATIVE APPLICATION

When moving the reference application to a cloud environment, the generic properties of this environment can be used to deduct required cloud application properties. The properties of the cloud environment have been defined by the NIST (2011): *On-demand self-service* – the cloud customer can independently sign up to the service and configure it to his demands. *Broad network access* – the cloud is connected to the customer network via a high-speed network. *Resource pooling* – resources required to provide the cloud service are shared among customers. *Rapid elasticity* – resources can be dynamically assigned to customers to handle currently occurring workload. *Measured service* – the use of the cloud by customers is monitored, often, to enable pay-per-use billing models.

To make an application suitable for such a cloud environment, i.e. to utilize the NIST properties, we identified the *IDEAL* cloud application properties (Fehling *et al.*, 2014): **I**solation of state, **D**istribution, **E**lasticity, **A**utomated Management and **L**oose coupling. In this section, we discuss why VM-based application virtualization and containerization alone is rather obstructive for realizing them. Based on this discussion, the steps for enabling these properties are described in order to transform a VM-based application towards a native cloud application. As we start our discussion on the level of VMs, we first focus on the Infrastructure as a Service (IaaS) service model. Then we show how it can be extended to use Platform as a Service (PaaS) offerings of a cloud provider.

---

[1] https://jcp.org/aboutJava/communityprocess/final/jsr318

[2] http://www.microsoft.com/net

3.1 Complete Application per Virtual Machine

To provide an application to customers within a cloud environment as quickly as possible, enterprises typically bundle their application into a single virtual machine image (VMI)[3]. Such VMIs are usually self-contained and include all components necessary for running the application. Considering the reference application, the data access layer, the business logic layer, and the presentation layer would be included in that VMI. Figure 2 shows an overview of that package.
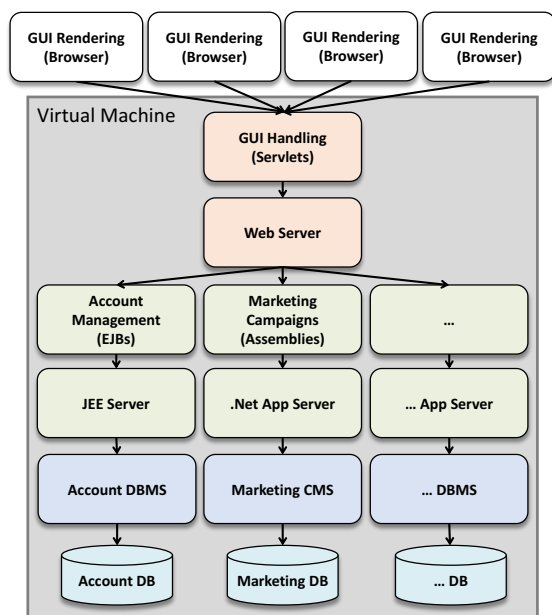

Figure 2: Packaging the Application into one VM

Customers now start using the application through their Web browsers. As shown in Figure 2, all requests are handled by the same VM. Consequently, the more customers are using the application, the more resources are required. At some point in time, considering an increasing amount of customer requests, the available resources will not be able to serve all customer requests any more. Thus, the application needs to be scaled in order to serve all customers adequately.

The first approach to achieve scalability is to instantiate another VM containing a copy of your

[3] From here on we do not mention containerization explicitly by considering them as similar to virtual machine images – well recognizing the differences. But for the purpose of our discussion they are very similar.

application stack as shown in Figure 3. This allows you to serve more customers without running into any bottleneck. However, the operation of multiple VMs also has significant downsides. You typically have to pay for licenses, e.g. for the database server, the application server, and the content management system, on a per VM basis. If customers use the account management features mostly, why should you also replicate the marketing campaigns stack and pay for the corresponding licenses? Next, what about your databases that are getting out of sync because separate databases are maintained in each of the VMs? This may happen because storage is associated to a single VM but updates need to be synchronized across those VM to result in consistent data.

Therefore, it should be possible to scale the application at a finer granular, to ensure that its individual functions can be scaled independently instead of scaling the application as a whole. This can be achieved by following the *distribution property* in the application architecture. This property requires the application functionality to be distributed among different components to exploit the measured service property and the associated pay-per-use pricing models more efficiently. Due to its modularized architecture comprising of logical layers and components, the distribution property is met by the reference application. However, by summarizing the components into one single VM, i.e. in one tier, the modularized architecture of the application gets lost.

Moreover, this leads to the violation of the *isolated state* property, which is relevant for the application to benefit from the resource pooling and elasticity property. This property demands that session and application state must be confined to a small set of well-known stateful components, ideally, the storage offerings and communication offerings of the cloud providers. It ensures that stateless components can be scaled more easily, as during the addition and removal of application component instances, no state information has to be synchronized or migrated, respectively.

Another IDEAL property that is just partly supported in case the application is bundled as a single VM is the *elasticity property*. The property requires that instances of application components can be added and removed flexibly and quickly in order to adjust the performance to the currently experienced workload. If the load on the components increases, new resources are provisioned to handle the increased load. If, in turn, the load on the resources decreases, under-utilized components are decommissioned. This *scaling out* (increasing the number of resources to adapt to workload) as

opposed to *scaling up* (increasing the capabilities of a single cloud resource) is predominantly used by cloud applications as it is also required to react to component failures by replacing failed components with fresh ones. Since the distribution is lost, scaling up the application by assigning more resources to the VM (e.g. CPU, memory, etc.) is fully supported, but not *scaling out* individual components. Hence, the elasticity property is just partly met if the application is bundled as a single VM. The incomplete support of the elasticity property also hinders the full exploitation of the cloud resource pooling property, as the elasticity property enables unused application resources to be decommissioned and returned to the resource pool of the cloud if they are not needed anymore. These resources can then be used by other customers or applications.
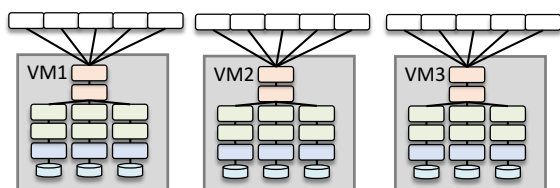

Figure 3: Scaling based on complete Virtual Machines

3.2 Stack-Based Virtual Machines with Storage Offerings

Because of the drawbacks of a single VM image containing the complete application, a suitable next step is to extract the different application stacks to separate virtual machines. Moreover, data can be externalized to storage offerings in the cloud ("Data as a Service"), which are often associated to the IaaS service model. Such services are used similar to hard drives by the VMs, but they are stored in a provider-managed scalable storage offering. Especially the stored data can be shared among multiple VMs when they are being scaled out, thus, avoiding the consistency problems indicated before and hence fostering the isolated state property. Figure 4 shows the resulting deployment topology of the application, where each stack and the Web GUI is placed into a different virtual machine that accesses a Data as a Service cloud offering.

When a particular stack is under high request load, it can be scaled out by starting multiple instances of the corresponding VM. For example, in Figure 5 another VM instance of the accounting stack is created to handle higher load. However, when another instance of a VM is created the DBMS is still replicated which results in increased license costs.

Figure 4: Packaging Stacks into VMs
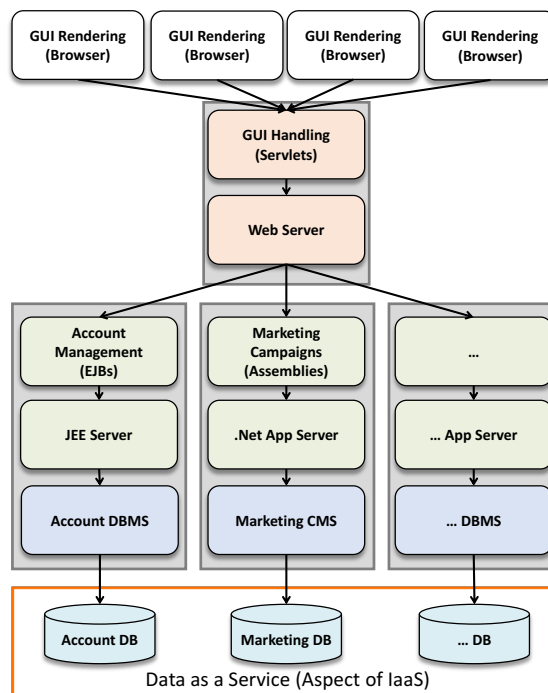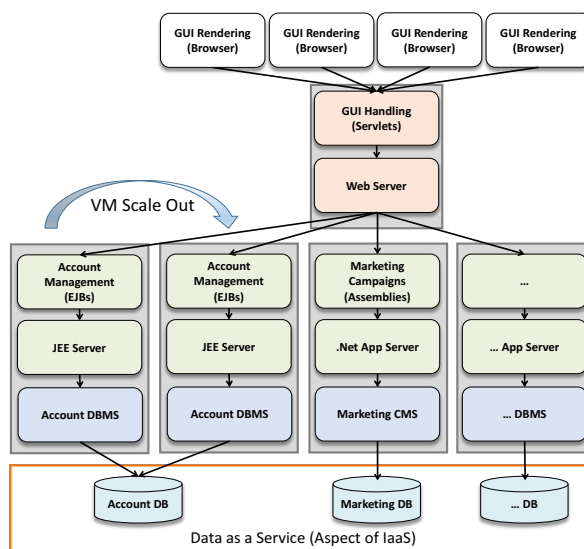

Figure 5: Packaging Stacks into VMs

3.3 Using Middleware Virtual Machines for Scaling

The replication of middleware components such as a DBMS can be avoided by placing these components again into separate VMs that can be scaled out independently from the rest of the application stack on demand. The middleware component is then able to serve multiple other components. In case of the

reference application the DBMS associated with the account management is moved to a new VM (Figure 6), which can be accessed by different instances of the JEE Server. Of course, also the JEE server or the .Net server could be moved into separate VMs. By doing so, the distribution property is increased and elasticity can be realized at a finer granular.
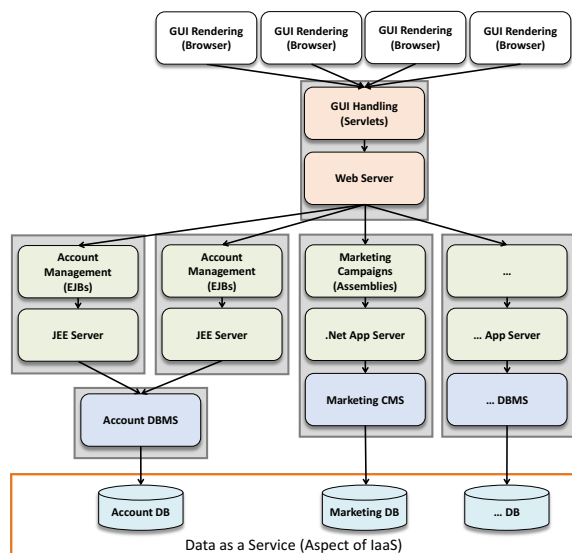

Figure 6: Middleware-VMs for Scaling

Even though the single components are now able to scale independently from each other, the problem of updating the application components and especially the middleware installed on VMs still remains. Especially, in large applications involving a variety of heterogeneous interdependent components this can become a very time- and resource-intensive task. For example, a new release of the JEE application server may also require your DBMS to be updated. But the new versions of the DBMS may not be compatible with the utilized .Net application server. This, in turn, makes it necessary to run two different versions of the same DBMS. However, this violates an aspect of the *automated management property* demanding that required human interactions to handle management tasks are reduced as much as possible in order to increase the availability and reactiveness of the application.

3.4 Resolving Maintenance Problems

To reduce management efforts, we can substitute components and middleware with IaaS, PaaS, or SaaS offerings from cloud providers. In Figure 7, the VMs providing the Web server and application server middleware are replaced with corresponding PaaS offerings. Now, it is the cloud provider's responsibility to keep the components updated and to

rollout new releases that contain the latest fixes, e.g. to avoid security vulnerabilities.

In case of the reference application, most components can be replaced by cloud offerings. The first step already replaced physical machines, hosting the application components with VMs that may be hosted on IaaS cloud environments. Instead of application servers, one may use PaaS offerings to host the application components of the business logic layer. The DBMS could be substituted by PaaS offerings such as Amazon SimpleDB; marketing campaign .Net assemblies could be hosted on Microsoft Azure, as an example.
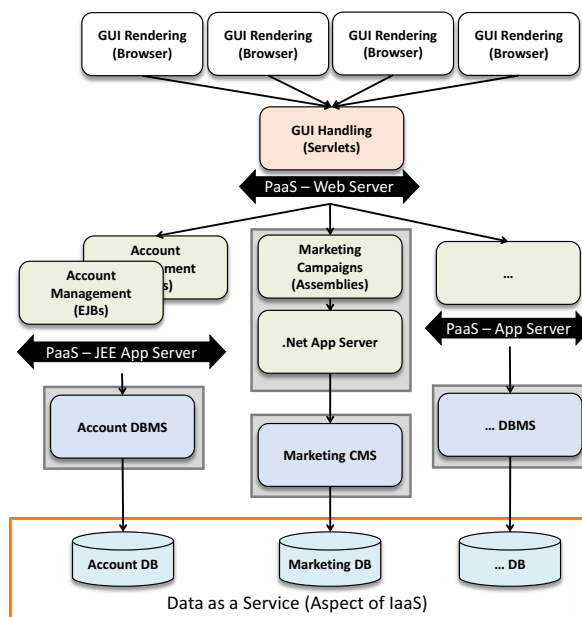
Figure 7: Making Use of Cloud Resources and Features

To offload the management (and even development) of your .Net assemblies one could even decide to substitute the whole marketing stack by a SaaS offering that provides the required marketing functionality. In this case, the Web GUI is integrated with the SaaS offering by using the APIs provided by the offering.

Of course, before replacing a component with an *aaS offering, it should be carefully considered how the dependent components are affected (Andrikopoulos *et al.*, 2013): adjustments to components may be required to respect the runtime environment and APIs of the used *aaS offering.

3.5 The Final Steps towards a Cloud Native Application

The reference application is now decomposed into multiple VMs that can be scaled individually to

fulfill the distribution and elasticity property. Isolation of state has been enabled by relying on cloud provider storage offerings. The software update management has been addressed partially.

However, the addition and removal of virtual machine instances can still be hindered by dependencies among application components: if a VM is decommissioned while an application component hosted on it interacts with another component, errors may occur. The dependencies between application components meaning the assumptions that communicating components make about each other can be reduced by following the *loose coupling property*. This property is implemented by using cloud communication offerings enabling asynchronous communication between components through a messaging intermediate as shown in Figure 8. This separation of concerns ensures that communication complexity regarding routing, data formats, communication speed etc. is handled in the messaging middleware and not in components, effectively reducing the dependencies among communication partners. Now, the application can scale individual components easier as components do not have to be notified in case other components are provisioned and decommissioned.

To make elastic scaling more efficient, it should be automated. Thus, again the automated management property is respected. This enables the application to add and remove resources without human intervention. It can cope with failures more quickly and exploits pay-per-use pricing schemes more efficiently: resources that are no longer needed should be automatically decommissioned. Consequently, the resource demand has to be constantly monitored and corresponding actions have to be triggered without human interactions. This is done by a separate *watchdog* component (Ornstein *et al.,* 1975; Fowler, 2002) and elasticity management components (Freemantle, 2010). After this step, the reference application became cloud native, thus, supporting the IDEAL cloud application properties: *Isolation of state*, *Distribution*, *Elasticity, Automated Management* and *Loose coupling* (Fehling *et al.*, 2014).

In terms of virtualization techniques and technologies, fully fledged VMs with their dedicated guest operation system could also be replaced by more lightweight virtualization approaches such as containers, which recently became popular with Docker (Mouat, 2015). However, such approaches may not provide the same degree of isolation, so depending on the specific requirements of an application, the one or the other virtualization approach fits better.
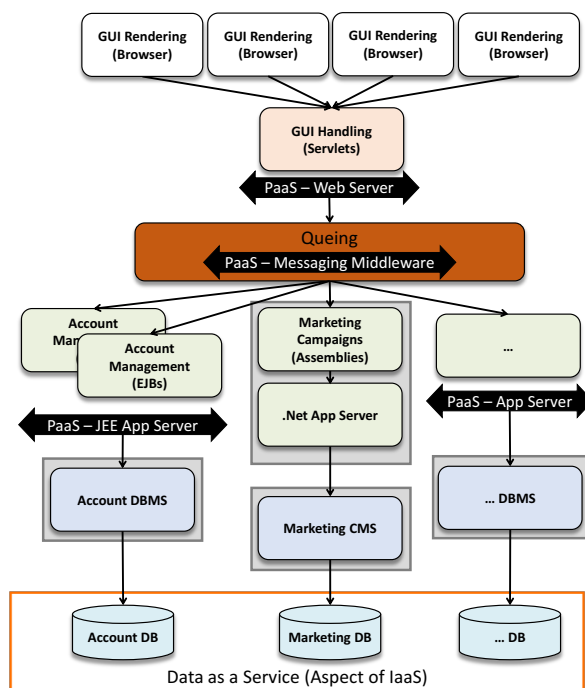

Figure 8: Making use of Cloud Communication Features

4 MICROSERVICES & CONTINUOUS DELIVERY

Microservice architectures provide an emerging software architecture style, which is currently discussed and hyped a lot. While there is no clear definition of what a microservice actually is, some common characteristics have been established (Fowler, 2016; Newman, 2015). Microservice architectures are contrary to monolithic architectures. Consequently, a specific application such as a Web application (e.g. the reference application presented in this paper) or a back-end system for mobile apps is not developed and maintained as a huge single building block, but as a set of 'small' and independent services, i.e. microservices. As of today, there is no common sense how 'small' a microservice should be. To make them meaningful, these services are typically built around business capabilities such as account management and marketing campaigns as outlined by the reference application. Their independence is implemented by running each service in its own process or container (Mouat, 2015). This is a key difference to other component-based architecture styles, where the entire application shares a process, but is internally modularized, for instance, using Java libraries.

The higher degree of independence in case of microservices enables them to be independently deployable from each other, i.e. specific parts of an application can be updated and redeployed without touching other parts. For non-trivial and more complex applications, the number of services involved quickly increases. Consequently, manual deployment processes definitely do not scale anymore for such architectures, because deployment happens much more often and independently. Therefore, fully automated deployment machinery such as continuous delivery pipelines are required (Humble and Farley, 2015).

As a side effect of the services' independence, the underlying technologies and utilized programming languages can be extremely diverse. While one service may be implemented using Java EE, another one could be implemented using .Net, Ruby, or Node.js. This enables the usage of 'the best tool for the job', because different technology stacks and programming languages are optimized for different sets of problems. The interface, however, which is exposed by a particular service must be technology-agnostic, e.g. based on REST over HTTP, so different services can be integrated without considering their specific implementation details. Consequently, the underlying storage technologies can also differ, because 'decentralized data management' (Fowler, 2016) is another core principle of microservice architectures. As outlined by the reference application, each service has its own data storage, so the data storage technology (relational, key-value, document-oriented, graph-based, etc.) can be chosen according to the specific storage requirements of a particular service implementation.

In addition, microservice architectures follow the principle of 'smart endpoints and dumb pipes' (Fowler, 2016), implying the usage of lightweight and minimal middleware components such as messaging systems ('dumb pipes'), while moving the intelligence to the services themselves ('smart endpoints'). This is confirmed by reports and surveys such as carried out by Schermann et al.: REST in conjunction with HTTP as transport protocol is used by many companies today. JSON and XML are both common data exchange formats. There is a trend to minimize the usage of complex middleware towards a more choreography-style coordination of services (Schermann et al., 2015). Finally, the architectural paradigm of self-contained systems (SCS, 2016) can help to treat an application, which is made of a set of microservices, in a self-contained manner.

In this context, an important fact needs to be emphasized: most of the core principles of microservice architectures are not new at all. Service-oriented architectures (SOA) are established in practice for some time already, sharing many of the previously discussed core principles with microservice architectures. Thus, we see microservice architectures as one possible opinionated approach to realize SOA, while making each service independently deployable. This idea of establishing independently deployable units is a focus of microservice architectures, which was not explicitly a core principle in most SOA-related works and efforts. Therefore, continuous delivery (Humble and Farley, 2015) can now be implemented individually per service to completely decouple their deployment.

Our previously presented approach to transition the references application's architecture towards a native cloud application is based on applying the IDEAL properties. The resulting architecture owns the previously discussed characteristics of microservice architectures and SOA. Each part of the reference application (account management, marketing campaigns, etc.) now represents an independently (re-)deployable unit. Consequently, if an existing application is transitioned towards a native cloud application architecture by applying the IDEAL properties, the result typically is a microservice architecture. To go even further and also consider development as part of the entire DevOps lifecycle, a separate continuous delivery pipeline (Humble and Farley, 2015) can be implemented for each service to perform their automated deployment when a bug fix or new feature is committed by a developer. Such pipelines combined with Cloud-based development environments such as Cloud9 (Cloud9, 2016) also make the associated application development processes cloud-native in addition to deploying and running the application in a cloud-native way.

5 MOVING TOWARDS A SAAS APPLICATION

While the IDEAL properties enable an application to benefit from cloud environments and (micro)service-oriented architectures, additional properties have to be considered in case the application shall be offered as a Service to a large number of customers (Freemantle, 2010; Badger *et al.*, 2011): Such applications should own the properties *clusterability*, *elasticity*, *multi-tenancy*, *pay-per-use* and *self-service*. Clusterability summarizes the above-mentioned isolation of state, distribution, and loose coupling. The elasticity discussed by Freemantle and Badger et al. is identical to the elasticity mentioned above. The remaining properties have to be enabled in an application-specific manner as follows.

5.1 Multi-Tenancy

The application should be able to support multiple tenants, i.e. defined groups of users, where each group is isolated from the others. Multi-tenancy does not mean isolation by associating each tenant with a separate copy of the application stack in one or more dedicated VMs. Instead, the application is adapted to have a notion of tenants to ensure isolation. The application could also exploit multi-tenant aware middleware (Azeez *et al.*, 2010) as this type of middleware is able to assign tenant requests to the corresponding instance of a component.

In scope of the reference application, the decomposition of the application into loosely coupled components enables the identification of components that can easily be shared among multiple tenants. Other components, which are more critical, for example, those sharing customer data likely have to be adjusted in order to ensure tenant isolation. In previous work, we discussed how such *shared components* and *tenant-isolated component*s may be implemented (Fehling *et al.*, 2014). Whether an application component may be shared among customers or not may also affect the distribution of application components to VMs.

5.2 Pay-Per-Use

Pay-per-use is a property that fundamentally distinguishes cloud applications from applications hosted in traditional datacenters. It ensures that tenants do only pay when they are actually using an application function, but not for the provisioning or reservation of application resources. Pay-per-use is enabled by fine-grained metering and billing of the components of an application stack. Consequently, the actual usage of each individual component within the application stack must be able to be monitored, tracked, and metered. Depending on the metered amount of resource usage, the tenant is billed. What kind of resources are metered and billed depends on the specific application and the underlying business model. Monitoring and metering can also be supported by the underlying middleware if it is capable to relate the requests made to the application components with concrete tenants.

In scope of the reference application, sharing application component instances ensures that the overall workload experienced by all instances is leveled out as workload peaks of one customer happen at the same time where another customer experiences a workload low. This sharing, thus, enables flexible pricing models, i.e. charging on a per-access basis rather than on a monthly basis. For instance, the reference application may meter and bill a tenant for the number of marketing campaigns he persists in the CMS. Other applications may meter a tenant based on the number requests or the number of CPUs he is using. Amazon, for instance, provides a highly sophisticated billing model for their EC2 instances (Amazon, 2016).

5.3 Self-Service

The application has to ensure that each tenant can provision and manage his subscription to the application on his own, whenever he decides to do so. Especially, no separate administrative staff is needed for provisioning, configuring, and managing the application. Self-service capability applies to each component of the application (including platform, infrastructure, etc.). Otherwise, there would not be real improvements in time-to-market. The self-service functionality can be provided by user interfaces, command line interfaces, and APIs to facilitate the automated management of the cloud application (Freemantle, 2010).

In scope of the reference application, automated provisioning and decommissioning of application component instances is enabled by the used cloud environment. Therefore, customers may be empowered to sign up and adjust subscriptions to the cloud-native application in a self-service manner, as no human management tasks are required on the application provider side anymore.

6 SUMMARY

Based on the IDEAL cloud application properties we have shown how an existing application can be transformed to a cloud native application. Moreover, we discussed the relation of cloud native application to (micro)service-oriented architectures and continuous delivery. Additional properties defined by Freemantle and Badger et al. – multi-tenancy, pay-per-use, and self-service – enabling a cloud-native application to be offered as a Service requiring significant adjustments of the application functionality. Multi-tenancy commonly requires adaptation of application interfaces and storage structures to ensure the isolation of tenants. Functionality to support pay-per-use billing and self-service commonly has to be newly created with application-specific knowledge.

Based on the transformation of the reference application we have shown that virtualization is a mandatory prerequisite for building a native cloud application, but just virtualizing an application does not satisfy all cloud application properties. Hence, it is insufficient to simply move an application into a VM and call it a cloud native application.

REFERENCES

Freemantle, P., 2010. Cloud Native. http://pzf.fremantle.org/2010/05/cloud-native.html

Fehling, C., Leymann F., Retter, R., Schupeck, W., Arbitter, P., 2014. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications.* Springer.

Moore, G., 2011. Systems of Engagement and The Future of Enterprise IT A Sea Change in Enterprise IT. Aim. white paper.

Mell, P., Grace, T., 2011. *The NIST Definition of Cloud Computing.* National Institute of Standards and Technology NIST, Gaithersburg, MD.

Badger, L., Grance, T., Patt-Corner, P., Voas, J., 2011. DRAFT: Cloud Computing Synopsis and Recommendation, NIST. http://csrc.nist.gov/publications/drafts/800-146/Draft-NIST-SP800-146.pdf

Ornstein, S. M., Crowther W. R., Kraley, M. F., Bressler, R. D., Michel, A., Heart, F. E., 1975. *Pluribus: a reliable multiprocessor.* In Proceedings of the 1975 American Federation of Information Processing Societies National Computer Conference (AFIPS).

Fowler, M., 2002. Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Azeez, A., Perera, S., Gamage, D., Linton, R., Siriwardana, P., Leelaratne, D., Weerawarana, S., Fremantle, P., 2010. *Multi-tenant SOA Middleware for Cloud Computing.* In Proceedings of the 2010 IEEE International Conference on Cloud Computing (CLOUD).

Andrikopoulos, V., Binz, T., Leymann, F., Strauch, S., 2013. *How to Adapt Applications for the Cloud Environment Challenges and Solutions in Migrating Applications to the Cloud.* In Computing. Vol. 95(6), Springer.

Andrikopoulos, V., Song, Z., Leymann, F., 2013. *Supporting the Migration of Applications to the Cloud through a Decision Support System.* In Proceedings of the 6th IEEE International Conference on Cloud Computing (CLOUD).

Fehling, C., Leymann, F., Ruehl, S. T., Rudek, M., Verclas, S., 2013. *Service Migration Patterns - Decision Support and Best Practices for the Migration of Existing Service-based Applications to Cloud Environments.* In Proceedings of the 6th IEEE International Conference on Service Oriented Computing and Applications (SOCA).

Fehling, C., Konrad, R., Leymann, F., Mietzner, R., Pauly, M., Schumm, D., 2011. *Flexible Process-based Applications in Hybrid Clouds.* In Proceedings of the 2011 IEEE International Conference on Cloud Computing (CLOUD).

Brandic, I., Anstett, T., Schumm, D., Leymann, F., Dustdar, S., Konrad, R., 2010. *Compliant Cloud Computing (C3): Architecture and Language Support for User-driven Compliance Management in Clouds.* In Proceedings of the 3rd International Conference on Cloud Computing (Cloud).

Amazon, 2016. Amazon Elastic Compute Cloud (EC2) Pricing. http://aws.amazon.com/ec2/pricing

Fowler, M., 2016. Microservices Resource Guide. http://martinfowler.com/microservices

Newman, S., 2015. Building Microservices. O'Reilly Media.

Humble, J., Farley, D., 2010. Continuous Delivery. Addison-Wesley Professional.

Mouat, A., 2015. Using Docker. O'Reilly Media.

Schermann, G.; Cito, J., Leitner, P., 2015. All the services large and micro: Revisiting industrial practices in services computing. PeerJ.

SCS, 2016. Self-contained System (SCS) – Assembling Software from Independent Systems. http://scs-architecture.org

Cloud9 IDE, Inc., 2016. Cloud9 website. https://c9.io

All links were last followed on 9th of February, 2016.