Native Cloud Applications: Why Monolithic Virtualization Is Not Their Foundation

Frank Leymann(✉), Uwe Breitenbücher, Sebastian Wagner, and Johannes Wettinger

IAAS, University of Stuttgart, Stuttgart, Germany
{leymann,breitenbucher,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 as a monolithic virtual machine or 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, which is an extension of a previous paper, 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. We also 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. Finally, we discuss the management of native cloud applications using container orchestration approaches as well as the cloud standard TOSCA.

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 [1–3]. 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, which is an extension of a previous paper [4] presented at the *6th International Conference on Cloud Computing and Services Science (CLOSER)*, we show that although virtualization lays the groundwork for cloud

M. Helfert et al. (Eds.): CLOSER 2016, CCIS 740, pp. 16–40, 2017.
DOI: 10.1007/978-3-319-62594-2_2

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 [5]. 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. Compared to the original article, we additionally discuss the deployment and management aspects of native cloud applications. Therefore, we describe how Kubernetes[1] can be employed to manage native cloud applications that are deployed using fine-grained Docker containers. We also point out specific limitations of Kubernetes and similar container orchestration approaches and discuss how to overcome those using TOSCA as cloud management standard. Note that we provide a definition of native cloud applications and how to deploy and manage them; 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 [2,6,7].

Section 2 introduces a reference application that reflects the core of the architectures of our industry use cases. Based on the reference application, Sect. 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. Section 5 discusses how the reference application itself can be offered as a cloud service. How to deploy and run a native cloud application using container orchestration is discussed in Sect. 6 by example of Kubernetes. In Sect. 7 a more holistic and technology-agnostic management approach of native cloud application based on TOSCA is described. Finally, Sect. 8 concludes the article.

2 Reference Application

Throughout the article, the application shown in Fig. 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 [8]: 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.

[1] http://kubernetes.io.

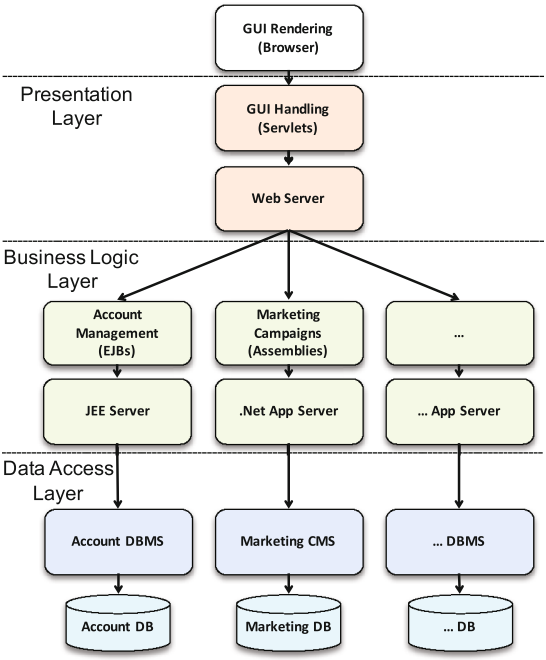

Fig. 1. Reference application to be moved to the cloud.

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[2] (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[3] 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

[2] https://jcp.org/aboutJava/communityprocess/final/jsr318.
[3] http://www.microsoft.com/net.

properties. The properties of the cloud environment have been defined by the NIST [9]: *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 [5]: *Isolation of state, Distribution, Elasticity, Automated Management and Loose 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.

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)[4]. 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.

Customers now start using the application through their Web browsers. As shown in Fig. 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 application stack as shown in Fig. 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

[4] 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.

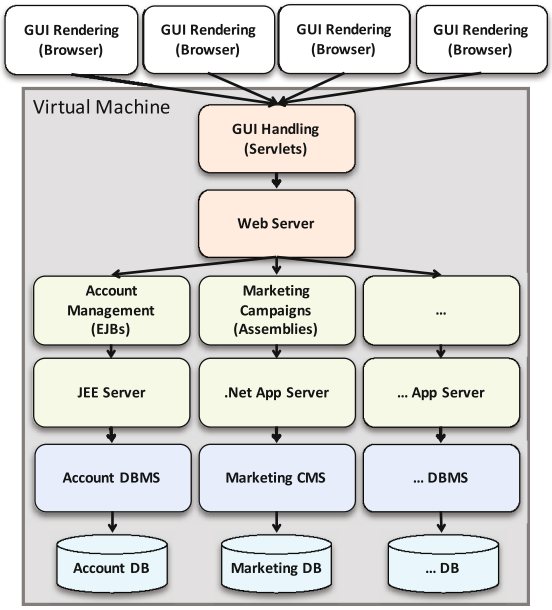

Fig. 2. Packaging the application into one VM.

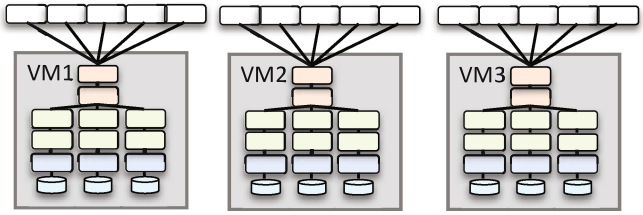

Fig. 3. Scaling based on complete VMs.

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 VMs 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.

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 Fig. 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.

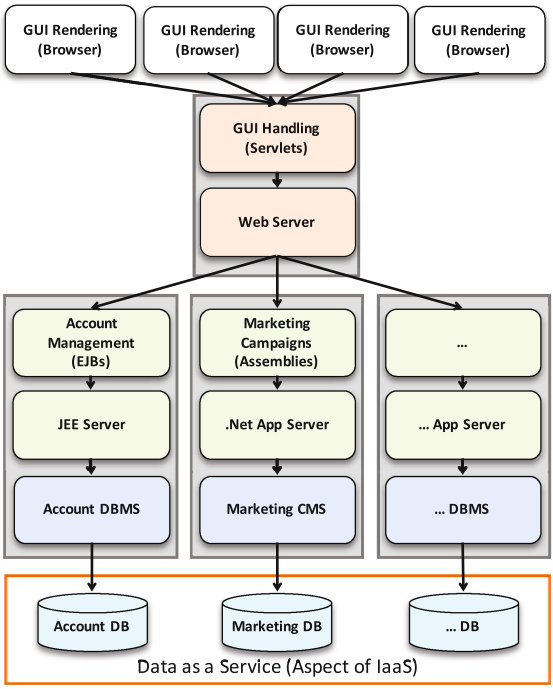

Fig. 4. Packaging stacks into VMs.

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 (Fig. 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.

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

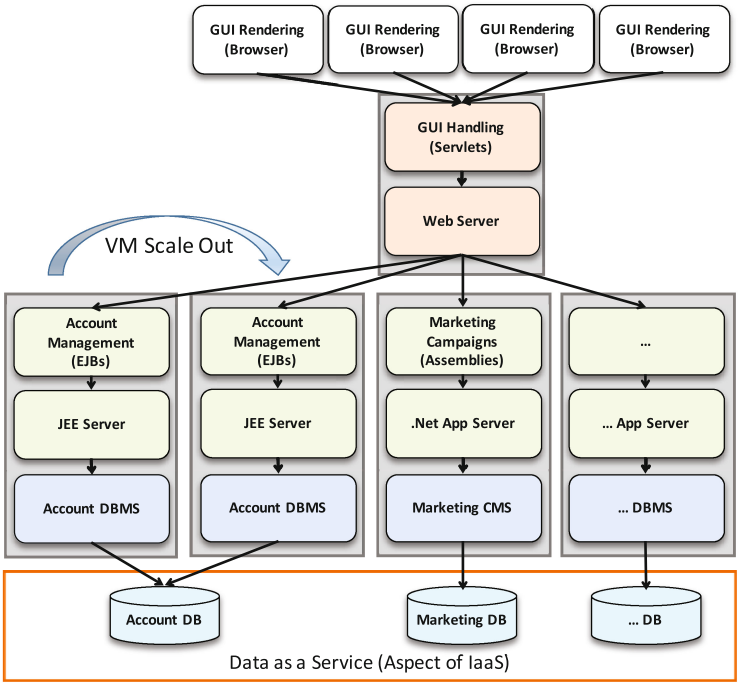

Fig. 5. Scaling stacked VMs.

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 Fig. 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.

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

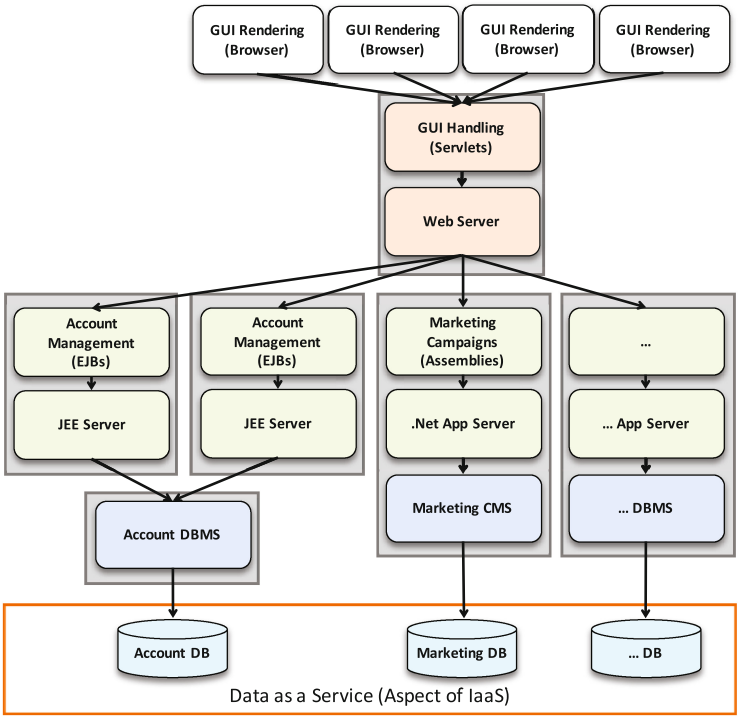

Fig. 6. Middleware-VMs for scaling.

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 [6]: 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

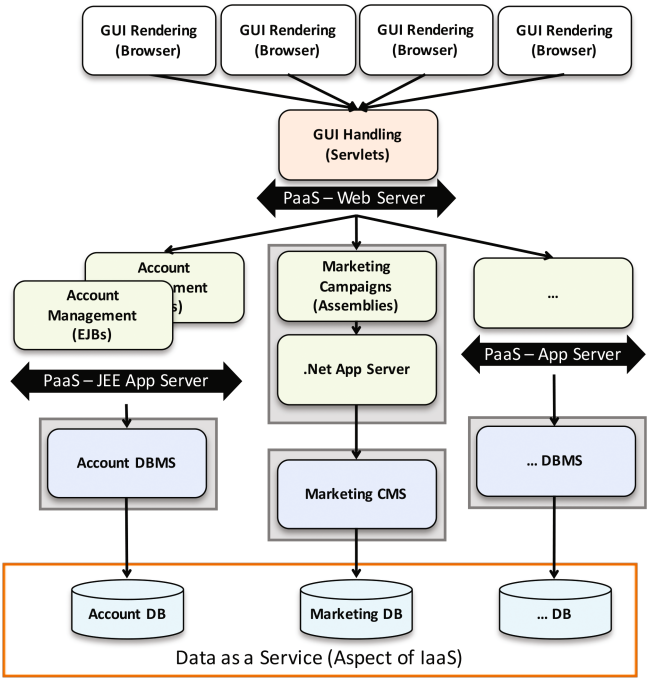

Fig. 7. Making use of cloud resources and features.

by using cloud communication offerings enabling asynchronous communication between components through a messaging intermediate as shown in Fig. 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 [8,10] and elasticity management components [11]. After this step, the reference application became cloud native, thus, supporting the IDEAL cloud application properties: ***I****solation of state,* ***D****istribution,* ***E****lasticity,* ***A****utomated Management and* ***L****oose coupling* [5].

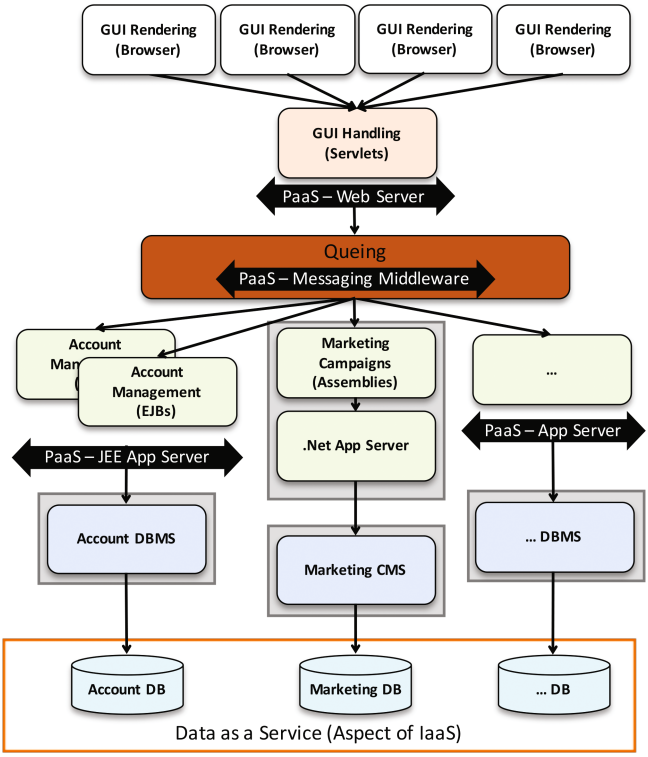

Fig. 8. Making use of cloud communication features.

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 [12]. 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.

4 Microservices and 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 [13,14]. 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 [12]. 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 [15].

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' [13] 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' [13], 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 [16]. Finally, the architectural paradigm of self-contained systems [17] 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 [15] can now be implemented individually per service to completely decouple their deployment.

Our previously presented approach to transition the reference 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 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 [18], 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 [11,19]: 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. [11,19] 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 [20] which is capable 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 components* may be implemented [5]. 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 [21].

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 [11].

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 Native Cloud Applications and Container Orchestration

The previous sections of this paper aim to point out the key concepts of native cloud applications. Applications that are simply packaged as a virtual machine image (VMI) or monolithic container do not benefit from the cloud properties discussed previously. Therefore, such applications do not represent native cloud applications. However, instead of packaging existing legacy applications as monolithic virtual machines, this paper presented an approach to systematically split such an application into fine-grained building blocks (Sect. 3). This approach is compatible with the emerging microservice architecture style (Sect. 4), which essentially is a modernized flavor and opinionated implementation of service-oriented architecture (SOA). Container virtualization (containerization) approaches such as Docker can be utilized to package and deploy applications comprising of such fine-grained building blocks, i.e. components or microservices: each component runs inside a separate container instead of hosting the entire application inside a monolithic container. The latter option would be conceptually the same as putting everything inside a monolithic virtual machine. This is technically different from using virtual machines because the overhead of VMs is significantly larger: each VM runs a completely dedicated instance of an operating system. While this approach leads to strong isolation, it implies the overhead of running each component on top of a dedicated operating system. Containers share the underlying operating system of their host and are, therefore, often referred to as 'lightweight' virtualization. Docker[5] and Rocket[6] are two popular container virtualization approaches. The Open Container Initiative (OCI)[7] aims to standardize a packaging format for containers.

While containers help to package, deploy, and manage loosely coupled components such as microservices that make up a specific application, their orchestration is a major challenge on its own. Container orchestration approaches such as Docker Swarm[8], Kubernetes, and Apache Mesos[9] are emerging to address this issue and simplify the process of combining and scaling containers. The Cloud Native Computing Foundation (CNCF)[10] is an emerging standard in this field, which promotes Kubernetes as one of the most actively developed container orchestration frameworks. Therefore, the remainder of this section discusses how the split application outlined in Fig. 7 can be deployed based on containers that are managed using Kubernetes.

Figure 9 outlines how the application topology can be distributed across a diverse set of containers, e.g. Docker containers. Persistent data is stored in container volumes[11], which are connected to the corresponding containers that run

[5] http://www.docker.com.
[6] http://coreos.com/rkt.
[7] http://www.opencontainers.org.
[8] http://docs.docker.com/engine/swarm.
[9] http://mesos.apache.org.
[10] http://cncf.io.
[11] http://docs.docker.com/engine/tutorials/dockervolumes.

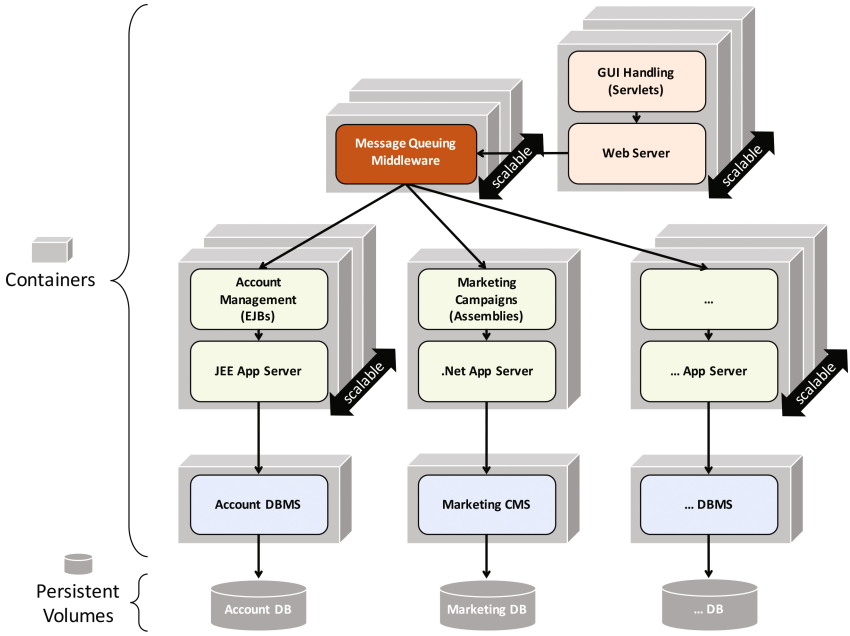

Fig. 9. Container-based application topology.

DBMS and CMS components. This approach makes data persistence more reliable because if a container crashes, a new one can be started and wired with the persistent volume. Some of the outlined containers such as the account management container can be scaled independently, i.e. container instances can be added and removed depending on the current load of this part of the application. This mechanism is the foundation for providing elasticity, an essential property of native cloud applications.

While Docker and similar solutions provide the basic mechanisms, tooling, and APIs to manage and wire containers, the orchestration and management of containers at scale add further challenges that have to be addressed. This is where large-scale container orchestration frameworks such as Kubernetes come into play. They provide key features that would have to be implemented using custom solutions on top of container virtualization solutions such as Docker. These features include the optimized distribution of containers inside a cluster, self-healing mechanisms to ensure that crashed containers are replaced, automated scaling mechanisms based on current load to ensure elasticity, and load balancing of incoming requests among container instances. Figure 10 sketches the simplified architecture using Kubernetes by example of the split application discussed in this paper. A cluster consists of *worker nodes* and at least one *master node*. Technically, a node is either a physical or virtual machine running on-premises or in the cloud. Each worker node runs Docker to host and execute containers. *Pods* are groups of containers that inherently belong together

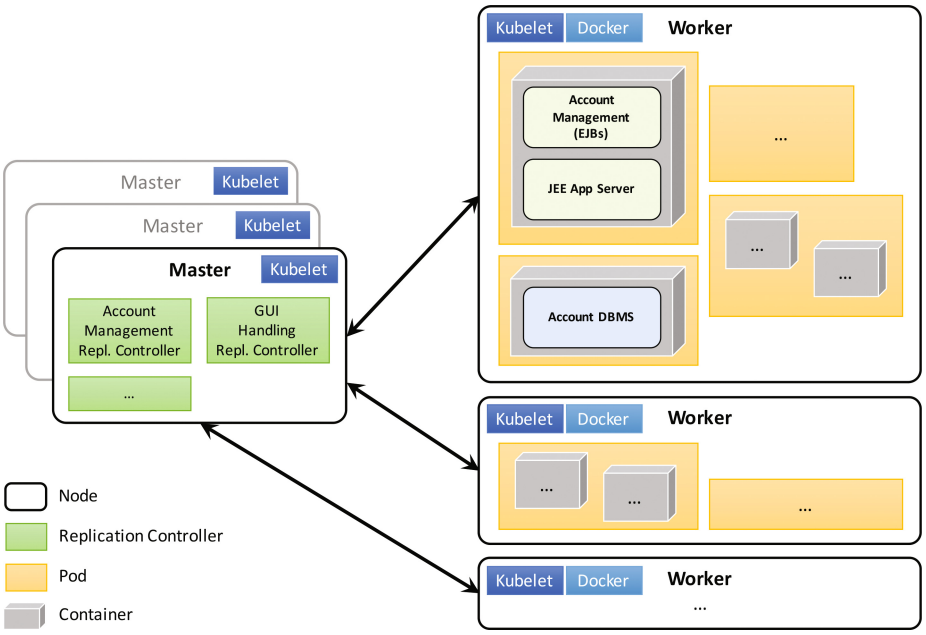

Fig. 10. Simplified Kubernetes architecture overview.

and cannot be deployed and scaled individually. Each pod consists of at least one container. Consequently, pods are the smallest units in Kubernetes that can be managed individually. Each node runs a *kubelet*, which acts as Kubernetes agent for cluster management purposes. The primary master node connects to these kubelets to coordinate and manage the other nodes, i.e. monitoring and scheduling containers, removing crashed ones, etc. Pods can either be maintained manually through the Kubernetes API of a cluster or their management is automated through application-specific *replication controllers* running on the master nodes. A replication controller such as the *account management replication controller* shown in Fig. 10 ensures that the defined number of replicas of a specific pod are up and running as part of the cluster. For example, the previously mentioned replication controller is associated with the pod comprising the container that runs the account management EJBs on top of the JEE application server. If, for instance, a replica of this pod fails on a particular node, the replication controller immediately schedules a new replica on any node in the cluster. While a single master node is technically sufficient, additional master nodes can be added to a cluster in order to ensure high availability: in case the primary master node fails, another master node can immediately replace it.

Technically, JSON or YAML files are used to define pods and replication controllers. The following YAML listing provides an example how the *account management pod* can be specified as outlined in Fig. 10 by the pod that consists

of a single container (containing account management EJBs and JEE application server), which runs on the upper worker:

```
apiVersion: v1
kind: Pod
metadata:
  # Labels describe properties of this pod
  labels:
    name: account-management-pod
spec:
  containers:
  - name: account-management
    # Docker image running in this pod
    # Contains EJBs and JEE application server
    image: account-mgmt-jee:stable
    ports:
    - containerPort: 8080
```

In order to maintain pods in an automated manner by Kubernetes, a replication controller can be created instead of defining a manually managed pod. The *account management replication controller* denoted in Fig. 10 can be specified, for example, using the following YAML listing:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: account-management-controller
spec:
  # Three replicas of this pod must run in the cluster
  replicas: 3
  # Specifies the label key and value on the pod that
  # this replication controller is responsible
  selector:
    app: account-management-pod
  # Information required for creating pods in the cluster
  template:
    metadata:
      labels:
        app: account-management-pod
    spec:
      containers:
      - name: account-management
        image: account-mgmt-jee:stable
        ports:
        - containerPort: 8080
```

Each pod dynamically gets an IP address assigned. Especially when dealing with multiple replicas that are managed through a replication controller, it is a major challenge to keep track of the IP addresses. Moreover, the transparent load balancing of requests between existing replicas is not covered. Kubernetes tackles these issues by allowing the definition of *services*, which can be specified as outlined by the following YAML listing:

```
apiVersion: v1
kind: Service
metadata:
  name: account-management-service
spec:
  ports:
  - port: 80 # port exposed by the service
    targetPort: 8080 # port pointing to the pod
  selector:
    app: account-management-pod
```

Each service provides a single and stable endpoint that transparently distributes requests between running replicas of the pod specified by the given selector, for example, the previously defined `account-management-pod`. In addition to pods, replication controllers, and services, Kubernetes allows to attach *volumes* to pods. For example, an Amazon Elastic Block Store volume[12] can be mounted into a pod to provide persistent storage. This is key for database back-ends of applications as outlined in Fig. 9.

Kubernetes provides a broad variety of additional features such as liveness and readiness probes[13], also known as health checks. These checks are not only important for monitoring purposes, e.g. to find out whether a particular container is up and running, they are also key to wire different containers. For example, container 1 connects to an endpoint exposed by container 2. However, the component started in container 2 that exposes the required endpoint takes some time until the connection can be established. Therefore, container 1 uses a readiness probe (e.g. an HTTP endpoint that must return the success response code 200) to find out when container 2 is ready for establishing the actual connection between the two components. Moreover, Kubernetes provides mechanisms to perform rolling updates[14] on a cluster, which is a key enabler for continuously delivering updated application instances into production as discussed in Sect. 4.

Although container orchestration such as Kubernetes provide a comprehensive framework to deploy and manage native cloud applications at scale, there are several constraints and assumptions made by such container orchestration approaches. (i) It is typically assumed that each application component runs inside a stateless container, i.e. a container can be terminated and replaced by a new instance of it at any point in time. Many existing legacy applications were not designed this way, so significant effort is required to wrap them as stateless containers. Moreover, the scaling mechanisms of some legacy application components rely on specific features provided by application servers such as Java EE servers. Migrating them toward a container orchestration is a non-trivial challenge. (ii) A container is considered immutable, i.e. container instances are not patched or updated (security fixes, etc.), but instances are dropped and replaced by updated containers. (iii) A Kubernetes cluster is based on a pretty homogenous infrastructure, e.g. a set of virtual servers running at a single cloud provider. Consequently, multi-cloud or hybrid cloud deployments are hard to implement. However, the Kubernetes community is working on cluster federation solutions[15]. (iv) Logic cannot be added to relationships between containers, i.e. the relations between containers are dumb (e.g. sharing TCP ports), so the containers must be smart enough to establish corresponding connections between them. This is achieved by utilizing readiness and liveness probes as discussed previously in this section. (v) Application components cannot be split into arbitrarily fine-grained

[12] http://kubernetes.io/docs/user-guide/volumes/#awselasticblockstore.
[13] http://kubernetes.io/docs/user-guide/production-pods.
[14] http://kubernetes.io/docs/user-guide/rolling-updates.
[15] http://github.com/kubernetes/kubernetes/blob/master/docs/proposals/federation.md.

containers. For example, the account management EJBs outlined in Fig. 9 must run in the same container as the JEE application server because the application server exposes the port that is used by other components to utilize the functionality provided by those EJBs. An EJB itself would not be capable of providing and exposing such a port through a separate container. (vi) Finally, external services such as platform-as-a-service offerings cannot be immediately managed as part of a container orchestration solution such as Kubernetes. Although containerized application components running inside the cluster can connect to external services such as a hosted database instance, their configuration and management is another challenge that needs to be tackled in addition to managing the cluster.

7 Standards-Based Modeling, Orchestration, and Management of Native Cloud Applications

In the previous sections, we discussed how native cloud applications can be developed, how the approach is compatible with the microservice architecture style, and how container virtualization approaches can be utilized to package and deploy such applications. However, we have seen that there are several constraints in terms of orchestrating containers, for example, application components must be stateless, strong assumptions about the communication in and between containers, and the difficulties regarding multi- and hybrid cloud applications. Moreover, the automation of holistic management processes, such as migrating the whole application including its data to another cloud provider, is an unsupported challenge. Therefore, in this section, we describe a solution to tackle these issues by discussing how the TOSCA [22–25] standard supports modeling, orchestrating, and managing various kinds of applications components.

The *Topology and Orchestration Specification for Cloud Applications (TOSCA)* is an OASIS standard released in 2013 that enables modeling cloud

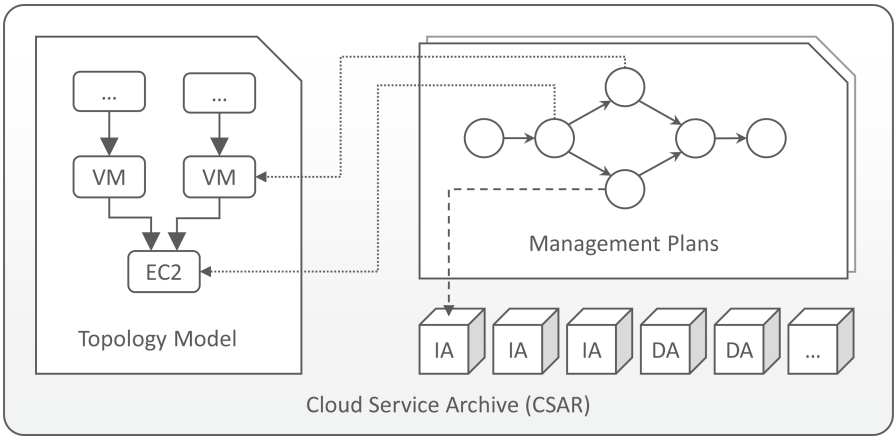

Fig. 11. The TOSCA concepts: topology model and management plans.

applications and their management in a portable manner. Beside portability, the automation of application provisioning and management is one of the major goals of this standard. TOSCA models conceptually consist of two parts as depicted in Fig. 11: (i) application topology model and (ii) management plans. An *application topology model* is a directed, possibly cyclic graph that describes the structure of the application to be provisioned. The nodes of this graph represent the application components (virtual machines, etc.), edges represent relationships between those. The topology model also specifies information about components and their relationships in terms of types, properties, available management interfaces, etc. Especially, TOSCA provides an extensible type system for describing the types of components and relationships. Thus, the standard allows modeling arbitrary components and relationship types, even the definition of proprietary types for supporting legacy applications—a specific feature that gets important in the following discussion.

The second part of TOSCA models are *management plans*, which are executable process models implementing management functionality for the described application, for example, to backup all application data or to migrate the application from one provider to another. Management plans are typically implemented using the workflow technology [26] in order to provide a reliable execution environment. In addition, standardized workflow languages such as BPEL [27] and BPMN [28] support this choice. To implement their functionality, plans invoke operations provided by the nodes of the topology model. Therefore, additional artifacts are required: (i) *deployment artifacts (DAs)* implement business logic whereas (ii) *implementation artifacts (IAs)* implement management operations provided by the components of the topology model. Typical deployment artifacts are, for example, the binaries of an application server whereas a corresponding implementation artifact may be a shell script that starts the application server. Moreover, TOSCA defines a portable packaging format, namely *Cloud Service Archive (CSAR)*, that contains the topology model, all management plans, as well as all other required artifacts. Thus, the archive is completely self-contained and provides all required information to automatically provision and manage the application using a *TOSCA Runtime Environment*, e. g., *OpenTOSCA* [29]. For more details about TOSCA we refer to Binz et al. [25], who provide a compact overview on the TOSCA standard.

These concepts of TOSCA tackle the orchestration issues mentioned in the previous section. First of all, the extensible type system allows modeling arbitrary types of application components and relationships that shall be provisioned. Thus, also stateful and non-containerized resources such as databases can be included in the provisioning process and do not have to be handled separately. Moreover, the extension concept enables modeling hybrid- and multi-cloud applications because arbitrary types can be defined and used in the topology model. For example, one part of the topology may be hosted on Amazon EC2, another part of the application on an on-premise OpenStack installation. This also enables including legacy components into the overall provisioning that shall not or cannot be adapted for the cloud following the approach presented

in Sect. 3. To specify non-functional requirements regarding the provisioning or management of the application, TOSCA employs the concept of *policies* [22] that may be used, for example, to specify the regions of a cloud provider in which an application is allowed to be deployed [30].

However, it is important to emphasize that the concept of topology models does not compete with containerization and container orchestration: containerized components, e. g., in the form of Docker containers, can be modeled as part of a topology model. Furthermore, container orchestration technologies such as Kubernetes can be modeled in the form of a component that hosts other components, in this case, Docker containers. Thus, arbitrary technologies can be included in these topology models since the type system is extensible. As a result, container technologies may be used as building blocks for TOSCA models. Moreover, using this modeling concept, they can be combined with other technologies that are not supported natively by container technologies: for example, a web application that is hosted on a web server contained in a Docker container may be connected to a legacy, non-containerized application that gets provisioned, too. In addition, containers hosted in different clouds may be wired with each other using this higher-level modeling concept. Therefore, TOSCA can be seen as an orchestration language on top of existing provisioning, virtualization, and management technologies.

To automate the provisioning of such *complex composite cloud applications*, TOSCA employs two different approaches: (i) declarative and (ii) imperative [23]. Using the declarative approach, the employed TOSCA Runtime Environment interprets the topology model and derives the required provisioning logic by itself. Based on well-defined interfaces, such as the *TOSCA lifecycle interface* [23] which defines operations to install, configure, start, stop, and terminate components, runtimes are enabled to provision arbitrary types of components as long as the Cloud Service Archive provides implementations for the required operations. Moreover, there are approaches that are capable of automatically generating provisioning plans out of topology models [31–33]. As the semantics of component types and relationship types—or at least of their management interfaces—must be defined and understood by the employed runtime, the declarative approach is limited to provisioning scenarios that consist of common components such as web servers, web applications, and databases. However, if the provisioning of a complex application requires custom provisioning logic for individual components, for example, legacy components, and an application-specific orchestration of those components, automatically deriving all required provisioning steps is often not feasible.

TOSCA tackles these issues by the imperative approach, which employs management plans to model all activities that have to be executed, including their control and data flow. As management plans, in this case *Build Plans* that automate the provisioning of the application [22], describe all steps to be executed explicitly, this concept enables customizing the provisioning for arbitrary applications including custom, application-specific logic. Thus, also complex applications that possibly employ containerization technologies hosted on different clouds connected

to stateful non-containerized legacy components can be provisioned automatically using this approach as any required activity can be described explicitly. The TOSCA-based management plan modeling language BPMN4TOSCA [34,35] additionally supports creating management plans based on the management operations provided by the components in the topology model. In addition, combined with the aforementioned plan generation, Build Plans may be generated even for complex applications and refined afterwards for custom, application-specific needs.

8 Summary

Based on the IDEAL cloud application properties we have shown how an existing application can be transformed to a native cloud application. Moreover, we discussed the relation of native cloud applications 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 native cloud application to be offered as a service require 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 using 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 in a monolithic manner does not satisfy all cloud application properties. Hence, it is insufficient to simply move an application into a monolithic virtual machine and call it a native cloud application. Furthermore, we outlined how emerging container orchestration approaches such as Kubernetes and Docker Swarm can be utilized to manage native cloud applications. Since these container orchestration approaches make specific assumptions about containerized application components, we further discussed how TOSCA can be used as standards-based modeling, orchestration, and management approach for native cloud applications. More specifically, TOSCA provides overarching and extensible mechanisms to integrate and orchestrate legacy, typically stateful, and non-containerized components with stateless and containerized application components.

References

1. Fehling, C., Leymann, F., Schumm, D., Konrad, R., Mietzner, R., Pauly, M.: Flexible process-based applications in hybrid clouds. In: Liu, L., Parashar, M., (eds.) IEEE International Conference on Cloud Computing, CLOUD 2011, Washington, DC, USA, 4–9 July 2011, pp. 81–88. IEEE (2011)
2. Fehling, C., Leymann, F., Ruehl, S.T., Rudek, M., Verclas, S.A.W.: Service migration patterns - decision support and best practices for the migration of existing service-based applications to cloud environments. In: 2013 IEEE 6th International Conference on Service-Oriented Computing and Applications, Koloa, HI, USA, 16–18 December 2013, pp. 9–16. IEEE Computer Society (2013)

3. Brandic, I., Dustdar, S., Anstett, T., Schumm, D., Leymann, F., Konrad, R.: Compliant cloud computing (C3): architecture and language support for user-driven compliance management in clouds. In: IEEE International Conference on Cloud Computing, CLOUD 2010, Miami, FL, USA, 5–10 July 2010, pp. 244–251 (2010)
4. Leymann, F., Fehling, C., Wagner, S., Wettinger, J.: Native cloud applications: why virtual machines, images and containers miss the point! In: Proceedings of the 6th International Conference on Cloud Computing and Service Science (CLOSER 2016), Rome, pp. 7–15. SciTePress (2016)
5. Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P.: Cloud Computing Patterns - Fundamentals to Design, Build, and Manage Cloud Applications. Springer, Heidelberg (2014)
6. Andrikopoulos, V., Binz, T., Leymann, F., Strauch, S.: How to adapt applications for the cloud environment - challenges and solutions in migrating applications to the cloud. Computing **95**, 493–535 (2013)
7. Binz, T., Breitenbücher, U., Kopp, O., Leymann, F.: Migration of enterprise applications to the cloud. IT - Information Technology, Special Issue: Architecture of Web Application, vol. 56, pp. 106–111 (2014)
8. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
9. Mell, P.M., Grance, T.: The NIST definition of cloud computing. Technical report, Gaithersburg, MD, United States (2011)
10. Ornstein, S.M., Crowther, W.R., Kraley, M.F., Bressler, R.D., Michel, A., Heart, F.E.: Pluribus: a reliable multiprocessor. In: AFIPS Conference Proceedings of American Federation of Information Processing Societies: 1975 National Computer Conference, Anaheim, CA, USA, 19–22 May 1975, vol. 44, pp. 551–559. AFIPS Press (1975)
11. Freemantle, P.: Cloud Native (2010). http://pzf.fremantle.org/2010/05/cloud-native.html
12. Mouat, A.: Using Docker: Developing and Deploying Software with Containers. O'Reilly Media, Sebastopol (2015)
13. Fowler, M.: Microservices Resource Guide (2016). http://martinfowler.com/microservices
14. Newman, S.: Building Microservices: Designing Fine-Grained Systems, 1st edn. O'Reilly Media, Sebastopol (2015)
15. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation, 1st edn. Addison-Wesley Professional, Upper Saddle River (2010)
16. Schermann, G., Cito, J., Leitner, P.: All the services large and micro: revisiting industrial practices in services computing. PeerJ PrePrints **3**, 36–47 (2015)
17. SCS: Self-contained System (SCS) Assembling Software from Independent Systems (2016). http://scsarchitecture.org
18. Cloud9 IDE Inc.: Cloud9 website (2016). https://c9.io
19. Badger, M.L., Grance, T., Patt-Corner, R., Voas, J.M.: Cloud computing synopsis and recommendations. Technical report, Gaithersburg, MD, USA (2012)
20. Azeez, A., Perera, S., Gamage, D., Linton, R., Siriwardana, P., Leelaratne, D., Weerawarana, S., Fremantle, P.: Multi-tenant SOA middleware for cloud computing. In: IEEE International Conference on Cloud Computing, CLOUD 2010, Miami, FL, USA, 5–10 July 2010, pp. 458–465 (2010)
21. Amazon: Amazon Elastic Compute Cloud (EC2) Pricing (2016). http://aws.amazon.com/ec2/pricing

22. OASIS: Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0. Organization for the Advancement of Structured Information Standards (OASIS) (2013)
23. OASIS: Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0. (2013)
24. OASIS: TOSCA Simple Profile in YAML Version 1.0 - Committee Specification (2016)
25. Binz, T., Breitenbücher, U., Kopp, O., Leymann, F. Advanced web services. In: Bouguettaya, A., Sheng, Q.Z., Daniel, F. (eds.) TOSCA: Portable Automated Deployment and Management of Cloud Applications, pp. 527–549. Springer, New York (2014)
26. Leymann, F., Roller, D.: Production Workflow: Concepts and Techniques. Prentice Hall PTR, Upper Saddle River (2000)
27. OASIS: Web Services Business Process Execution Language (WS-BPEL) Version 2.0. Organization for the Advancement of Structured Information Standards (OASIS) (2007)
28. OMG: Business Process Model and Notation (BPMN) Version 2.0. Object Management Group (OMG) (2011)
29. Binz, T., Breitenbücher, U., Haupt, F., Kopp, O., Leymann, F., Nowak, A., Wagner, S.: OpenTOSCA – a runtime for TOSCA-based cloud applications. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) ICSOC 2013. LNCS, vol. 8274, pp. 692–695. Springer, Heidelberg (2013). doi:10.1007/978-3-642-45005-1_62
30. Waizenegger, T., et al.: Policy4TOSCA: a policy-aware cloud service provisioning approach to enable secure cloud computing. In: Meersman, R., Panetto, H., Dillon, T., Eder, J., Bellahsene, Z., Ritter, N., Leenheer, P., Dou, D. (eds.) OTM 2013. LNCS, vol. 8185, pp. 360–376. Springer, Heidelberg (2013). doi:10.1007/978-3-642-41030-7_26
31. EI Maghraoui, K., Meghranjani, A., Eilam, T., Kalantar, M., Konstantinou, A.V.: Model driven provisioning: bridging the gap between declarative object models and procedural provisioning tools. In: Steen, M., Henning, M. (eds.) Middleware 2006. LNCS, vol. 4290, pp. 404–423. Springer, Heidelberg (2006). doi:10.1007/11925071_21
32. Mietzner, R.: A method and implementation to define and provision variable composite applications, and its usage in cloud computing. Dissertation, Universitt Stuttgart, Fakult ät Informatik, Elektrotechnik und Informationstechnik (2010)
33. Breitenbücher, U., Binz, T., Képes, K., Kopp, O., Leymann, F., Wettinger, J.: Combining declarative and imperative cloud application provisioning based on TOSCA. In: International Conference on Cloud Engineering (IC2E 2014), pp. 87–96. IEEE (2014)
34. Kopp, O., Binz, T., Breitenbücher, U., Leymann, F.: BPMN4TOSCA: a domain-specific language to model management plans for composite applications. In: Mendling, J., Weidlich, M. (eds.) BPMN 2012. LNBIP, vol. 125, pp. 38–52. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33155-8_4
35. Kopp, O., Binz, T., Breitenbücher, U., Leymann, F., Michelbach, T.: A domain-specific modeling tool to model management plans for composite applications. In: Proceedings of the 7th Central European Workshop on Services and their Composition (ZEUS 2015), CEUR Workshop Proceedings, pp. 51–54 (2015)