Towards a Lightweight Multi-Cloud DSL for Elastic and Transferable Cloud-native Applications

Peter-Christian Quint and Nane Kratzke
Lübeck University of Applied Sciences, Center of Excellence for Communication, Systems and Applications (CoSA), 23562 Lübeck, Germany

Keywords: Cloud-native Applications, TOSCA, Docker Compose, Swarm, Kubernetes, Domain Specific Language, DSL, Cloud Computing, Elastic Container Platform.

Abstract: Cloud-native applications are intentionally designed for the cloud in order to leverage cloud platform features like horizontal scaling and elasticity – benefits coming along with cloud platforms. In addition to classical (and very often static) multi-tier deployment scenarios, cloud-native applications are typically operated on much more complex but elastic infrastructures. Furthermore, there is a trend to use elastic container platforms like Kubernetes, Docker Swarm or Apache Mesos. However, especially multi-cloud use cases are astonishingly complex to handle. In consequence, cloud-native applications are prone to vendor lock-in. Very often TOSCA-based approaches are used to tackle this aspect. But, these application topology defining approaches are limited in supporting multi-cloud adaption of a cloud-native application at runtime. In this paper, we analyzed several approaches to define cloud-native applications being multi-cloud transferable at runtime. We have not found an approach that fully satisfies all of our requirements. Therefore we introduce a solution proposal that separates elastic platform definition from cloud application definition. We present first considerations for a domain specific language for application definition and demonstrate evaluation results on the platform level showing that a cloud-native application can be transfered between different cloud service providers like Azure and Google within minutes and without downtime. The evaluation covers public and private cloud service infrastructures provided by Amazon Web Services, Microsoft Azure, Google Compute Engine and OpenStack.

1 INTRODUCTION

Elastic container platforms (ECP) like Docker Swarm, Kubernetes (k8s) and Apache Mesos received more and more attention by practitioners in recent years (de Alfonso et al., 2017) – and this trend still seems to continue (Kratzke and Quint, 2017). Elastic container platforms fit very well with existing cloud-native application (CNA) architecture approaches (Kratzke and Quint, 2017). Corresponding system designs often follow a microservice-based architecture (Sill, 2016; Kratzke and Peinl, 2016). Nevertheless, the reader should be aware that the effective and elastic operation of such kind of elastic container platforms is still a question in research – although there are interesting approaches making use of bare metal (de Alfonso et al., 2017) as well as public and private cloud infrastructures (Kratzke and Quint, 2017). What is more, there are open issues how to design, define and operate cloud applications on top of such container platforms pragmatically. This is especially true for multi-cloud contexts. Such open issues in scheduling microservices to the cloud come along with questions regarding interoperability, application topology and composition aspects (Saatkamp et al., 2017) as well as elastic runtime adaption aspects of cloud-native applications (Fazio et al., 2016). The combination of these three aspects (multi-cloud interoperability, application topology definition/composition and elastic runtime adaption) is – to the best of the authors' knowledge – not solved satisfactorily so far. These three problems are often seen in isolation. In consequence, **topology based multi-cloud approaches often do not consider elastic runtime adaption** of deployments (Saatkamp et al., 2017) and **multi-cloud capable elastic solutions being adaptive at runtime do not make use of topology based approaches** as well (Kratzke and Quint, 2017). And finally, (topology-based) cloud-native applications making use of elastic runtime adaption are often inherently bound to specific cloud infrastructure services (like cloud provider specific monitoring, scaling

Quint, P. and Kratzke, N.
Towards a Lightweight Multi-Cloud DSL for Elastic and Transferable Cloud-native Applications.
DOI: 10.5220/0006683804000408
In *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018)*, pages 400-408
ISBN: 978-989-758-295-0

and messaging services) making it hard to transfer these cloud applications easily to another cloud provider or even operate them across providers at the same time (Kratzke and Peinl, 2016). Furthermore, Heinrich et al. mention several research challenges and directions like microservice focused performance monitoring under runtime adaption approaches (Heinrich et al., 2017). All in all, it seems like cloud engineers (and researchers as well) just trust in picking only two out of three options. *Is this really the best approach?*

Therefore, this contribution strives for a more integrated point of view to overcome the observable isolation of these mentioned engineering and research trends (Kratzke and Quint, 2017) and tries to analyze how and whether the mentioned approaches can be combined. We intentionally strive for a pragmatic and practitioner acceptance instead of richness of expression like this is done by approaches like CAMEL (Rossini, 2015). Other than CAMEL, we focus microservice architectures and elastic container platforms only to reduce language complexity. So, we do not follow an holistic approach considering every imaginable architectural style of cloud applications.

We present a prototype for a domain-specific language (DSL) that enables to describe cloud-native applications being transferable at runtime without downtimes according to the following **outline**. The key idea is to describe the platform independently from the application. Our DSL has been developed according to a three step DSL design methodology: analysis, implementation and use as proposed by (Van Deursen et al., 2000; Mernik et al., 2005; Strembeck and Zdun, 2009). In Section 2 we analyze common characteristics of elastic container platforms and derive concepts that have to be covered by cloud-native application definition DSLs. In Section 3 we refine these concepts into more concrete requirements and analyze related work and existing DSLs like TOSCA. We found no existing language that fulfills all of our identified requirements completely. Accordingly, we propose and present a prototypic implementation for such a DSL and evaluate it in Section 4. Our evaluation shows, that cloud-native applications can be transferred between different cloud service providers like Azure and Google within minutes and without downtime. We have executed our experiments on public and private cloud service infrastructures provided by Amazon Web Services, Microsoft Azure, Google Compute Engine and OpenStack. This section closes with a critical discussion. Finally, we conclude our considerations and provide an outlook in Section 5.

2 CONTAINERIZATION TRENDS

According to (Kratzke and Quint, 2017), a CNA runs on top of an elastic runtime environment. This can be straightaway an Infrastructure-as-a-Service (IaaS) or an elastic platform (Fehling et al., 2014). Container-based elastic platforms are getting more and more widespread. Such kind of elastic ECP are shown in Table 1. For the aim to avoid vendor lock-in (Kratzke and Peinl, 2016) we propose to make use of basic and standardized IaaS service concepts only. Such concepts are virtual machines, virtualized (block-)storage devices, virtualized networks and security groups. Elastic container platforms can be deployed on top of these basic IaaS service concepts (Kratzke, 2017). And on top of elastic container platforms arbitrary cloud-native applications can be deployed (Kratzke and Peinl, 2016).

Figure 1 illustrates such kind of ECP based CNA deployments. (Kratzke, 2017) showed that arbitrary ECPs can be operated using a descriptive cluster definition model based on an intended and a current state. Such kind of defined clusters can be operated or even transfered across different cloud service provider infrastructures at runtime. Obviously, this is a great foundation to avoid vendor lock-in situations. While a descriptive cluster definition model can be used for describing the elastic platform (Kratzke, 2017), there is also the need to describe the application topology without dependency to a specific ECP. This paper proposes to do this using a domain-specific language which focuses on the layer 5 and 6 of the cloud-native application reference model proposed by (Kratzke and Peinl, 2016). This DSL is the focal point of this paper. The central idea is to split the migration problem into two independent engineering problems which are too often solved together.

1. The **infrastructure aware** deployment and operation of ECPs: These platforms can be deployed and operated in a way that they can be transferred across IaaS infrastructures of different private and public cloud services as (Kratzke, 2017) showed.
2. The **infrastructure agnostic** deployment of applications on top of these kind of transferable container platforms which is the focus of this paper.

In order to enable an ECP-based CNA deployment by a domain-specific language that is not bound to a specific ECP, the particular characteristics and commonalities of the target systems have to be identified. Therefore, the architectures and concepts of elastic container platforms have to be analyzed and compared. As representatives, we have chosen the three most often used elastic container platforms Kubernetes, Docker Swarm Mode and Apache Mesos with

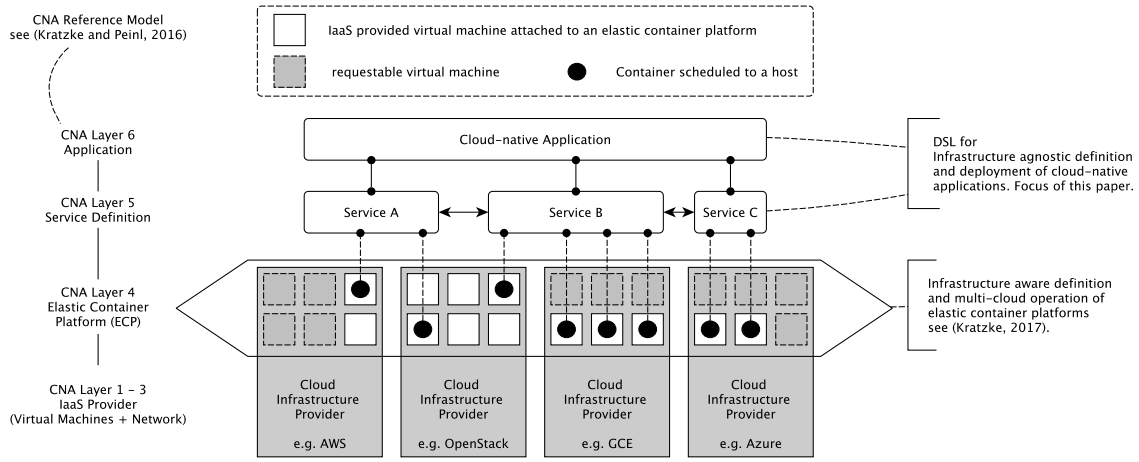

Figure 1: An ECP based CNA deployment for elastic and multi-cloud capable operation.

Table 1: Some popular open source elastic platforms. *These kind of platforms can be used as a kind of cloud infrastructure unifying middleware.*

Platform	Contributors	URL
Kubernetes	CNCF	http://kubernetes.io
Swarm	Docker	https://docker.io
Mesos	Apache	http://mesos.apache.org

Marathon listed in Table 1. As Table 2 shows all of these ECPs provide comparable concepts (from a bird's eye view).

Application Definition. All platforms define applications as a set of deployment units. The dependencies of these deployment units are expressed in a descriptive way. Apache Mesos uses Application Groups to partition multiple applications into sets. The dependencies are modeled as n-ary trees of groups with applications as leaves. Kubernetes manages an application basically as a set of services composed of pods. A pod can contain one or more containers. All containers grouped in a pod run on the same machine in the cluster. ReplicationSet Controllers take care that the number of running pods is equal to the amount of pods defined in replication controller configurations (Verma et al., 2015). The numbers of running instances of a pod is defined in a so called deployment (YAML-file). Docker Swarm supports application description using a single YAML-file that defines a multi-container deployment consisting of the container and there connections. YAML based definition formats seem to be common for all ECPs and *a DSL should provide something like a model-to-model transformation (M2M) to these ECP specific application definition formats **[AD]**.*

Service discovery is the task to get service endpoints by name and not by a (permanently) changing address. All analyzed ECPs supported service discovery by DNS based solutions (Mesos, Kubernetes) or using the service names defined in the application definition format (Docker Swarm). *Thus, a DSL must consider to name services in order to make them discoverable via DNS or ECP-specific naming services **[SD]**.*

Deployment units. The basic units of execution are named different by the ECPs. However, they mostly based on containers. Docker Swarm is intentionally designed for deploying Docker containers. A Kubernetes deployment unit is called a pod. And a pod whose the container can be operated by arbitrary container runtime environments. But Rocket (rkt) and Docker are the main container runtime environments at the time of writing this paper. Only Apache Mesos supports by its design arbitrary binaries as deployment units. However, the Marathon framework supports container workloads based on Docker containers and emerges as a standard for the Mesos platform to operate containerized workloads. *A DSL should consider that the deployment unit concept (whether named application group, pod or container) is the basic unit of execution for all ECPs **[DU]**.*

Scheduling. All ECPs provide some kind of a scheduling service that mostly runs on the master nodes of these platforms. The scheduler assigns deployment units to nodes of the ECP considering the current workload and resource efficiency. The scheduling process of all ECPs can be constrained using scheduling constraints or so called (anti-)affinities (Verma et al., 2015; Naik, 2016; Hindman et al., 2011). *These kind of scheduling constraints must be considered and expressable by a DSL **[SCHED]**.*

Load Balancing. Like scheduling, load balancing is supported by almost all analyzed platforms using special add-ons like Marathon-lb-autoscale (Mesos), kube-proxy (Kubernetes), or Ingress service (Docker

Table 2: Concepts of analyzed ECP Architectures.

Concept	Mesos	Docker Swarm	Kubernetes
Application Definition	Application Group	Compose	Service + Namespace Controller (Deployment, DaemonSet, Job, ...) All K8S concepts are described in YAML
Service discovery	Mesos DNS	Service names Service links	KubeDNS (or replacements)
Deployment Unit	Binaries Pods (Marathon)	Container (Docker)	Pod (Docker, rkt)
Scheduling	Marathon Framework Constraints	Swarm scheduler Constraints	kube-scheduler Affinities + (Anti-)affinities
Load Balancing	Marathon-lb-autoscale	Ingress load balancing	Ingress controller kube-proxy
Autoscaling	Marathon-autoscale	-	Horizontal pod autoscaling
Component Labeling	key/value	key/value	key/value

Swarm). These load balancers provide basic round-robin load balancing strategies and they are used to distribute and forward IP traffic to the deployment units under execution. *However, more sophisticated load balancing strategies should be considered as future extensions for a DSL **[LB]**.*

Autoscaling. Except for Docker Swarm, all analyzed ECPs provide (basic) autoscaling features which rely mostly on measuring CPU or memory metrics. In case of Docker Swarm this could be extended using an add-on monitoring solution triggering Docker Compose file updates. The Mesos platform provides Marathon-autoscale for this purpose and Kubernetes relies on a horizontal pod autoscaler. Furthermore, Kubernetes supports even making use of custom metrics. *So, a DSL should provide support for autoscaling supporting custom and even application specific metrics **[AS]**.*

Component Labeling. All ECPs provide a *key/value* based labeling concept that is used to name components (services, deployment units) of applications. This labeling is used more (Kubernetes) or less (Docker Swarm) intensively by concepts like service discovery, schedulers, load balancers and autoscalers. These concepts could be also of use for the operator of the cloud-native application to constrain scheduling decisions in multi-cloud scenarios. This component labeling can be used to code datacenter regions, prices, policies and even enable to deploy services only on specific nodes (Kratzke, 2017). *In consequence, a DSL should be able to label application components in key/value style **[CL]**.*

3 A DSL FOR CNA

For deploying arbitrary CNAs on specific elastic container platforms, we have developed a model-to-model (M2M) generator. As shown in Figure 2, the generated, ECP specific CNA description can be used by a ECP scheduler to deploy a new application or update a still running one. The operation of the ECPs can be done in a multi-cloud way (Kratzke, 2017). The elastic container platforms can be transferred across different cloud service platforms or they can be operated in a multi- or hybrid-cloud way. More details can be found in (Kratzke, 2017).

To define a universal CNA definition DSL, we followed established methodologies for DSL development as proposed by (Van Deursen et al., 2000; Mernik et al., 2005; Strembeck and Zdun, 2009). According to Section 2 the following requirements arise for a DSL with the intended purpose to define elastic, transferable, multi-cloud-aware cloud-native applications being operated on elastic container platforms.

- **R1: Containerized deployments**. Containers are self-contained deployment units of a service and the core building block of every modern cloud-native application. **The DSL must be designed to describe and label a containerized deployment of discoverable services**. This requirement comprises **[SD], [DU], and [CL]**.

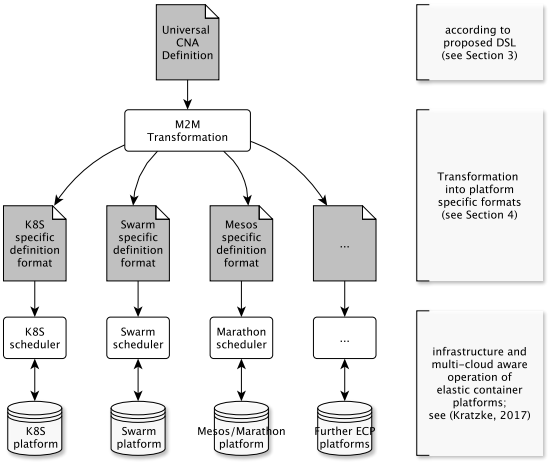

Figure 2: Separation of concerns in deploying a CNA.

Table 3: Requirement Matching.

Requirement #	R1 Container	R2 Application Scaling	R3 Compendiously	R4 Multi-Cloud Support	R5 Independence	R6 Elastic Runtime Env.	Implementations	Describtion
CAMEL		+	-	+	+	+	R	CAMEL is designed for modeling and execution of multi-cloud applications (Rossini, 2015). It integrates and extends existing DSLs like CloudML and supports models@run-time (Blair et al., 2009),(Chauvel et al., 2013), an environment to provide a model-based representation of the underlying running system, which facilitates reasoning and adaptation of multi-cloud applications.
CAML			-	-	+		R	CAML enables the use of provider-dependent services (described in the CAML Profiles) and the deployment (described in the CAML Library). The cloud applications deployment configuration can be reused by using CAML templates (Bergmayr et al., 2014)
CloudML	+	+	-	+	+	+	R	A DSL for multi-cloud application deployments. The CloudMF (Lushpenko et al., 2015) framework consists of CloudML (Brandtzæg et al., 2012) and models@run-time (see row above)
Docker Compose	+	+	+	+	-	+	P	Is an orchestration DSL and tool for defining, linking, and running multiple containers on any docker host, also on the container cluster Docker Swarm
Kubernetes	+	+	+	+	-	+	P	Kubernetes (former Google Borg) is a cluster platform for deploying container applications. All configurations like the scheduling units (pods) and the scaling properties (replication controller) can be described in YAML files (Verma et al., 2015)
TOSCA	+	+	-	+	+	-	R&P	A specification for describing the topology and orchestration of cloud webservices, their relations and components of composition and how to manage them (Binz et al., 2014)
MODACloudML	+	+	-	+	+	+	R	Designed to specify the provision and deployment of applications in multi-cloud environments (Artač et al., 2016). MODACloudsML is the DSL part of MODACloud and also uses CloudMF (see row CloudML)
MULTICLAPP		-		+	+	-	-	A framework for modeling cloud applications on multi-cloud environments, independent from the IaaS-provider. Applications can be modeled with an UML profile

Legend for column *Implementation*: P: Productive useable implementations available; R: Research implementations available

- **R2: Application Scaling**. Elasticity and scalability are one of the major advantages using cloud computing (Vaquero et al., 2011). Scalability enables to follow workloads by request stimuli in order to improve resource efficiency (Mao and Humphrey, 2011). **The DSL must be designed to describe elastic services.** This requirement comprises **[SCHED], [LB], and [AS]**.
- **R3: Compendiously**. To simplify operations the DSL should be pragmatic. Our approach is based on a separation between the description of the application and the elastic container platform. **The DSL must be designed to be lightweight and infrastructure-agnostic.** This requirement comprises **[AD], [SD], and [CL]**.
- **R4: Multi-Cloud-Support**. Using multi-cloud-capable ECPs for deploying CNAs is a major requirement for our migration approach. Multi-cloud support also enables the use of Hybrid-cloud infrastructures. **The DSL must be designed to support multi-cloud operations**. This requirement comprises **[SCHED], [CL]** and the necessity to be applied on ECPs operated in a way described by (Kratzke, 2017).
- **R5: Independence**. To avoid dependencies, the CNA should be deployable independently to a specific ECP and also to specific IaaS providers. **The DSL must be designed to be independent from a specific ECP or cloud infrastructure.** This requirement comprises **[AD]** and the necessity to be applied on ECPs operated in a way described by (Kratzke, 2017).
- **R6: Elastic Runtime Environment**. Our approach provides a CNA deployment on an ECP which is transferable across multiple IaaS cloud infrastructures. **The DSL must be designed to define applications being able to be operated on an elastic runtime environment**. This requirement comprises **[SD], [SCHED], [LB], [AS], [CL]** and should consider the operation of ECPs in way that is described in (Kratzke, 2017) .

According to these requirements, we examined existing domain-specific languages for similar kind of purposes. By investigating literature and conducting practical experiments in expressing a reference application, we analyzed whether the DSL fulfills our requirements. The results are shown in Table 3. No of the examined DSLs covered all of the requirements. TOSCA, Docker Compose and Kubernetes fulfill the most of our requirements. But Docker Compose and the Kubernetes DSL are designed for a specific ECP (Docker Swarm and Kubernetes). We decided against TOSCA because of its tool-chain complexity and its tendency to cover all layers of abstraction (especially the infrastructure layer). Accordingly, we identified the need for creating a new DSL (at least for our research activities). Furthermore, to define a new DSL provides the maximum flexibility in covering all of the mentioned and derived requirements.

Figure 3 summarizes the core language model for the resulting DSL. An **Application** *provides* a set of **Service**s. A **Service** can have an **Endpoint** on which its features are exposed. One **Endpoint** *belongs* exactly to one **Service** and is *associated* with a **Load Balancing Strategy**. A **Service** can *use* other **Endpoint**s of other **Service**s as well. These **Service**s can be external Services that are not part of the application deployment itself. However, each internal **Service** *executes* at least one **DeploymentUnit** which is *composed of* one or more **Container**s. Furthermore, schedulers of ECPs should *consider* **DeploymentPolicies** for **DeploymentUnit**s. Such **DeploymentPolicies** can be workload considering **Scaling Rules** but

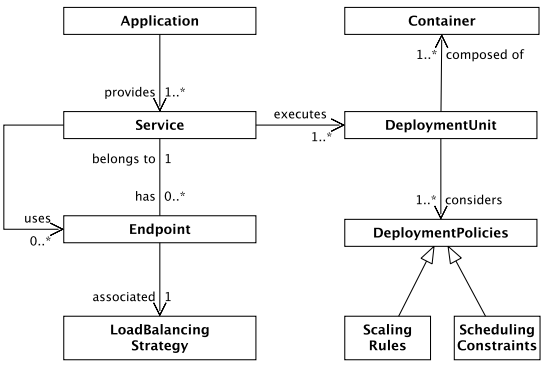

Figure 3: DSL Core Language Model.

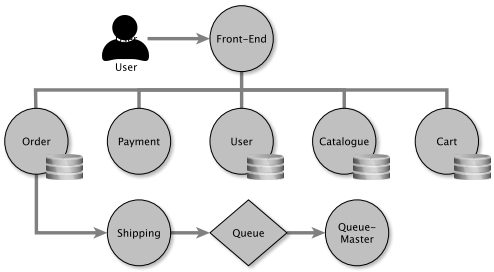

Figure 4: Architecture of the reference application *Sock Shop*, according to (Weaveworks, 2017).

also general **Scheduling Constraints**.

Table 4 relates these DSL concepts to identified requirements of Section 3 and initially identified trends in containerization of Section 2. Multi-Cloud support (requirement R4) is not directly mapped to a specific DSL concept. It is basically supported by separating the description of the ECP (Kratzke, 2017) and the description of the CNA. Therefore, multi-cloud support must not be a part of the CNA DSL itself, what makes the CNA description less complex. Multi-cloud support is simply delegated to the lower level ECP. This kind of multi-cloud handling for ECPs is explained in more details by (Kratzke, 2017).

According to (Van Deursen et al., 2000) we implemented this core language model as a declarative, internal DSL in Java. Although Java is a quite uncommon language to build a DSL, as a full purpose programming language it provides maximum flexibility to find DSL internal solutions. On the other hand, making use of proven software patterns makes it even possible to provide human readable forms of application definitions (see Listing 1). To keep the description of a CNA simple to use and also short, we used the *Builder Pattern* (Gamma et al., 2000). The usage of this pattern allows a flexible definition of a CNA without having to pay attention to the order of the description. The concrete syntax is shown exemplary using an example service as part of a *SockShop* reference application that we used for our evaluation (Listing 1).

4 EVALUATION

We validated that our DSL fulfills all requirements we defined in Section 3 by three evaluation steps:

E1. To evaluate the usability of the DSL for describing a containerized (**R1**) , auto-scalable (**R2**) deployment in a pragmatic way (**R3**), we described a microservice demonstration application. Therefore we selected *Sock Shop*, a reference microservice e-commerce application for demonstrating and testing of microservice and cloud-native technologies (Weaveworks, 2017). Sock Shop is developed using technologies like Node.js, Go, Spring Boot and MongoDB and is one of the most complete reference applications for cloud-native application research according to (Aderaldo et al., 2017). As shown in Figure 4, the application consists of nine services. Due to page limitations, we only provide one description of the payment-service as example in Listing 1.

E2. To evaluate multi-cloud-support (**R4**) and ECP independence (part of **R5**) we deployed and operated the *Sock Shop* on two ECPs hosted on several IaaS infrastructures. As type representatives we selected Docker Swarm Mode (Version 17.06) and Kubernetes (Version 1.7). The ECPs consist of five working machines (and one master) hosted on the IaaS infrastructures OpenStack, Amazon AWS, Google GCE and Microsoft Azure.

E3. For demonstrating IaaS independence (**R5**) we migrated the deployment between various IaaS infrastructures of Amazon Web Services, Microsoft Azure, Google Compute Engine and a research institution specific OpenStack installation. To validate all migration possibilities we have done the following experiments:

- **E3.1**: Migration OpenStack[1] ⇔ AWS [2]
- **E3.2:** Migration OpenStack ⇔ GCE [3]
- **E3.3**: Migration OpenStack ⇔ Azure [4]
- **E3.4**: Migration ⇔ and GCE
- **E3.5**: Migration AWS ⇔ Azure
- **E3.6**: Migration GCE ⇔ Azure

We have used Kubernetes and Docker Swarm[5] as ECP for the Sock Shop deployment. Every experiment is a set of migrations in both directions. E.g., evaluation experiment E3.1 includes migrations from OpenStack to AWS and from AWS to OpenStack. All migrations with OpenStack as source or target infrastructure (E3.1-E3.3) have been carried out ten times, all other (E3.4-E3.6) five times. The transfer times of the infrastructure migrations are shown in Figure 5. As the reader can see, the needed time for a infrastructure migration stretches from 3 minutes (E3.1 OpenStack ⇒ AWS) to more than 18min (E3.6 Azure ⇒ AWS). Moreover, the transfer time for migrating also depends on the transfer direction between the *source*

[1] *Own Plattform*, machines with 2vCPUs
[2] Region *eu-west-1*, Worker node type *m4.xlarge*
[3] Region *europe-west1*, Worker node type *n1-standard-2*
[4] Region *europewest*, Worker node type *Standard_A2*
[5] Due to page limitations we only present Kubernetes data. However, our experiments revealed that most runtime is spent in infrastructure specific handling and not due to the choice of the elastic container platform.

Table 4: Mapping DSL concepts to derived requirements (R1-R6) and containerization trends (AD, SD, DU, ..., CL).

Concept	R1	R2	R3	R4	R5	R6	AD	SD	DU	SCHED	LB	AS	CL
Application					x		x						
Service	x							x					x
Endpoint			x					x			x		x
DeploymentUnit	x								x	x			x
Container	x								x	x			
DeploymentPolicies				x		x		x		x		x	x
LoadBalancingStrategy						x		x			x	x	
Scaling Rules		x		x		x		x		x		x	x
Scheduling Constraints		x		x		x		x		x		x	x

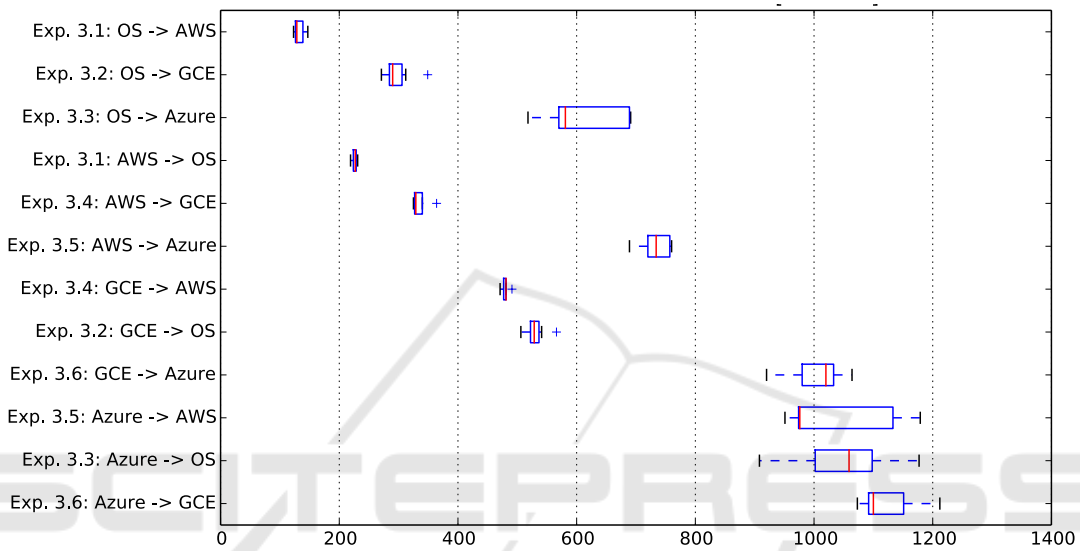

Figure 5: Measured durations of application migrations [seconds].

and the *target* infrastructure. E.g., as seen in E3.3, the migration Azure ⇒ AWS takes four times longer than the reversed migration AWS ⇒ Azure. Our analysis turned out, that the differences in the transfer times are mainly due to different blocking behavior of the IaaS API operations of different providers. Especially providers whose terminating operation of virtual machines or security groups are blocking operations show significantly longer reaction times. E.g., IaaS terminating operations of GCE and Azure wait until an operation is finished completely before starting the next one. This takes obviously longer than just waiting for the confirmation that an infrastructure operation has started (IaaS API behavior of OpenStack and AWS). However, and in all cases the reference application could be transferred completely and without downtime between all mentioned providers. The differences in transfer times are due to different involved IaaS cloud service providers and not due to the presented DSL.

Limitations and Critical Discussion. In our current work we have not evaluated the migration of a stateful applications deployment with a mass of data. This would involve the usage of a storage cluster like Ceph or GlusterFS. The transfer of such kind of storage clusters will be investigated separately. We also rated the DSL pragmatism and practitioner acceptance higher than the richness of possible DSL expressions. This was a result according to discussions with practitioners (Kratzke and Peinl, 2016). This results in some limitations. For instance, our DSL is intentionally designed for container and microservice architectures, but has limitations to express applications out of this scope. This limits language complexity but reduces possible use cases. For applications outside the scope of microservice architectures, we recommend to follow more general TOSCA or CAMEL based approaches.

5 CONCLUSION

Open issues in deploying cloud-native applications to cloud infrastructures come along with the combination of multi-cloud interoperability, application topology definition/composition and elastic runtime adap-

Listing 1: The payment service of the Sock Shop reference application expressed in the proposed DSL.

```
DeploymentPolicy dPolicy = new DeploymentPolicy.Builder()
    .rule(DeploymentPolicy.Type.NUMBER, 3)
    .rule(DeploymentPolicy.Type.SELECTOR, "openStack.dc1")
    .build();
Container paymentContainer = new Container.Builder("payment")
    .image("weaveworksdemos/payment:0.4.3")
    .port(new Endpoint.Builder().containerPort(80).build())
    .build();
DeploymentUnit deploymentUnit = new DeploymentUnit.Builder("payment")
    .container(paymentContainer)
    .tag("app", "nginx")
    .deploymentPolicy(dPolicy)
    .build();
Service service = new Service.Builder("payment")
    .deploymentUnit(deploymentUnit)
    .port(new Port.Builder("http")
    .protocol(Port.Protocol.TCP).containerPort(80).targetPort(80).build())
    .build();

new Generator.Builder().targetECP(Generator.ECP_TYPES.KUBERNETES)
    .deyploment(service)
    .build()
    .write(new File("/path/to/folder"));
```

tion. This combination is – to the best of the authors' knowledge – not solved satisfactorily so far, because these three problems are often seen in isolation. It seems that cloud engineers (and researchers as well) just trust in picking only two out of these three options. Therefore, this paper strived for a more integrated point of view to overcome the observable isolation of these mentioned engineering and research trends (Kratzke and Quint, 2017). The key idea is to describe the platform independently from the application. According to our lessons learned, the **infrastructure aware** deployment and operation of ECPs should be separated from **infrastructure and platform agnostic** deployment of applications.

This paper focused on DSL design for the application level. However, if we take further research for the ECP and infrastructure level into consideration (Kratzke, 2017), we are able to demonstrate that a cloud-native application can be defined in a descriptive and infrastructure and platform-agnostic way simply using a specialized DSL. Our reference application composed of nine services could be expressed using the proposed prototypic version of such kind of a DSL. Furthermore, the application could be transferred between different cloud infrastructures within minutes and without downtimes.

Our DSL core language is implemented as internal DSL in Java to fulfill our own special demands in a fast and pragmatic way. But we see the need for a representation of our core language model without the overhead of a full purpose language like Java. Further research will investigate whether it is useful to make use of more established topology DSLs like TOSCA and how to realize a comparable expressiveness like CAMEL. However, we do not strive for the technological possible, but also considere the balance between language expressiveness, pragmatism, complexity and practitioner acceptance.

ACKNOWLEDGEMENTS

This research is funded by German Federal Ministry of Education and Research (13FH021PX4). Let us thank all the anonymous reviewers and their comments that improved this paper.

REFERENCES

Aderaldo, C. M., Mendonça, N. C., Pahl, C., and Jamshidi, P. (2017). Benchmark requirements for microservices architecture research. In *Proceedings of the 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering*, pages 8–13. IEEE Press.

Artač, M., Borovšak, T., Di Nitto, E., Guerriero, M., and Tamburri, D. A. (2016). Model-driven continuous deployment for quality devops. In *Proceedings of the 2nd International Workshop on Quality-Aware DevOps*, pages 40–41. ACM.

Bergmayr, A., Wimmer, M., Kappel, G., and Grossniklaus, M. (2014). Cloud modeling languages by example. In *Proceedings - IEEE 7th International Conference on*

Service-Oriented Computing and Applications, SOCA 2014.

Binz, T., Breitenbücher, U., Kopp, O., and Leymann, F. (2014). TOSCA: Portable Automated Deployment and Management of Cloud Applications. In *Advanced Web Services*, pages 527–549. Springer New York, New York, NY.

Blair, G., Bencomo, N., and France, R. B. (2009). Models@ run. time. *Computer*, 42(10).

Brandtzæg, E., Mosser, S., and Mohagheghi, P. (2012). Towards cloudml, a model-based approach to provision resources in the clouds. In *8th European Conference on Modelling Foundations and Applications (ECMFA)*, pages 18–27.

Chauvel, F., Ferry, N., Morin, B., Rossini, A., and Solberg, A. (2013). Models@ runtime to support the iterative and continuous design of autonomic reasoners. In *MODELS@ Run. time*, pages 26–38.

de Alfonso, C., Calatrava, A., and Moltó, G. (2017). Container-based virtual elastic clusters. *Journal of Systems and Software*, 127(January):1–11.

Fazio, M., Celesti, A., Ranjan, R., Liu, C., Chen, L., and Villari, M. (2016). Open Issues in Scheduling Microservices in the Cloud. *IEEE Cloud Computing*, 3(5):81–88.

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

Gamma, E., Hehn, R., Johnson, R., et al. (2000). Design patterns: Elements of reusable design. soil.

Heinrich, R., van Hoorn, A., Knoche, H., Li, F., Lwakatare, L. E., Pahl, C., Schulte, S., and Wettinger, J. (2017). Performance Engineering for Microservices: Research Challenges and Directions. In *8th ACM/SPEC on Int. Conf. on Performance Engineering Companion*, pages 223–226.

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I. (2011). Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA. USENIX Association.

Kratzke, N. (2017). Smuggling Multi-cloud Support into Cloud-native Applications using Elastic Container Platforms. In *8th. Int. Conf. on Cloud Computing and Service Sciences*, Porto, Portugal.

Kratzke, N. and Peinl, R. (2016). ClouNS - a Cloud-Native Application Reference Model for Enterprise Architects. In *2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW)*, pages 198–207, Vienna. IEEE.

Kratzke, N. and Quint, P. C. (2017). Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study. *Journal of Systems and Software*, 126(January):1–16.

Lushpenko, M., Ferry, N., Song, H., Chauvel, F., and Solberg, A. (2015). Using adaptation plans to control the behavior of models@ runtime. In *MoDELS@ Run. time*, pages 11–20.

Mao, M. and Humphrey, M. (2011). Auto-scaling to minimize cost and meet application deadlines in cloud workflows. *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11*, page 1.

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344.

Naik, N. (2016). Building a virtual system of systems using docker swarm in multiple clouds. In *Systems Engineering (ISSE), 2016 IEEE International Symposium on*, pages 1–3. IEEE.

Rossini, A. (2015). Cloud application modelling and execution language (camel) and the paasage workflow. In *Advances in Service-Oriented and Cloud Computing—Workshops of ESOCC*, volume 567, pages 437–439.

Saatkamp, K., Breitenbücher, U., Kopp, O., and Leymann, F. (2017). Topology Splitting and Matching for Multi-Cloud Deployments. In *8th Int. Conf. on Cloud Computing and Service Sciences (CLOSER 2017)*.

Sill, A. (2016). The Design and Architecture of Microservices. *IEEE Cloud Computing*, 3(5):76–80.

Strembeck, M. and Zdun, U. (2009). An approach for the systematic development of domain-specific languages. *Software: Practice and Experience*, 39(15):1253–1292.

Van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36.

Vaquero, L. M., Rodero-Merino, L., and Buyya, R. (2011). Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review*, 41(1):45.

Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. (2015). Large-scale cluster management at Google with Borg. *Proceedings of the Tenth European Conference on Computer Systems - EuroSys '15*, pages 1–17.

Weaveworks (2017). Sock shop: A microservices demo application. https://www.weave.works/blog/sock-shop-microservices-demo-application/ . Accessed: 2017-12-18.