Smuggling Multi-cloud Support into Cloud-native Applications using Elastic Container Platforms

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

Keywords: Cloud-native Application, Multi-cloud, Elastic Platform, Container, Microservice, Portability, Transferability, MAPE, AWS, GCE, OpenStack, Kubernetes, Docker, Swarm.

Abstract: Elastic container platforms (like Kubernetes, Docker Swarm, Apache Mesos) fit very well with existing cloud-native application architecture approaches. So it is more than astonishing, that these already existing and open source available elastic platforms are not considered more consequently in multi-cloud research. Elastic container platforms provide inherent multi-cloud support that can be easily accessed. We present a solution proposal of a control process which is able to scale (and migrate as a side effect) elastic container platforms across different public and private cloud-service providers. This control loop can be used in an execution phase of self-adaptive auto-scaling MAPE loops (monitoring, analysis, planning, execution). Additionally, we present several lessons learned from our prototype implementation which might be of general interest for researchers and practitioners. For instance, to describe only the intended state of an elastic platform and let a single control process take care to reach this intended state is far less complex than to define plenty of specific and necessary multi-cloud aware workflows to deploy, migrate, terminate, scale up and scale down elastic platforms or applications.

1 INTRODUCTION

Cloud-native applications (CNA) are large scale elastic systems being deployed to public or private cloud infrastructures. The capability to design, develop and operate a CNA can create enormous business growth and value in a very limited amount of time. Companies like Instagram, Netflix, Dropbox, etc. proofed that imposingly. They operate these kind of applications often on large scale elastic container-based clusters with up to thousands of nodes. However, due to slender standardization in cloud computing it can be tricky to operate CNA across differing public or private infrastructures in multi-cloud or hybrid cloud scenarios. CNA – even when written from scratch – are often targeted for a specific cloud only. The effort for porting in a different cloud is usually a one time exercise and can be very time consuming and complex. For instance, Instagram had to analyze their existing services for almost one year to derive a viable migration plan how to transfer their services from *Amazon Web Services* (*AWS*) to Facebook datacenters. This migration worked at last, but it was accompanied by severe outages. This phenomenon is called a vendor lock-in and CNA seem to be extremely vulnerable for it. Almost no recent multi-cloud survey study (see Section 6) considered elastic container platforms (see Table 1) as a viable and pragmatic option to support multi-cloud handling. It is very astonishing that this kind of already existing and open source available technology is not considered more consequently in multi-cloud research (see Section 6). That might have to do with the fact, that *"the emergence of containers, especially container supported microservices and service pods, has raised a new revolution in [...] resource management. However, dedicated auto-scaling solutions that cater for specific characteristics of the container era are still left to be explored."* (Ch. Qu and R. N. Calheiros and R. Buyya, 2016). The acceptance of container technologies and corresponding elastic container platforms has gained substantial momentum in recent years. That resulted in a lot of technological progress driven by companies like Docker, Netflix, Google, Facebook, Twitter who released their solutions very often as Open Source software. So, from the current state of technology existing multi-cloud approaches (often dated before container technologies have been widespread) seem very complex – much

Kratzke, N.
Smuggling Multi-cloud Support into Cloud-native Applications using Elastic Container Platforms.
DOI: 10.5220/0006230700570070
In *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*, pages 29-42
ISBN: 978-989-758-243-1

Table 1: Some popular open source elastic platforms and their major contributing organizations.

Platform	Contributors	URL
Kubernetes	Google	`http://kubernetes.io`
Swarm	Docker	`https://docker.io`
Mesos	Apache	`http://mesos.apache.org/`
Nomad	Hashicorp	`https://nomadproject.io/`

too complex for a lot of use cases of cloud-native applications which have become possible due to the mentioned technological progress of the last three or four years. This paper considers this progress and has mainly two contributions:

- A control loop (see Section 4.2) is presented being able to scale elastic container platforms in multi-cloud scenarios. This single control loop is capable to handle common multi-cloud workflows like to deploy, to migrate/transfer, to terminate, to scale up/down CNAs. The control loop is providing not just scalability but federation and transferability across multiple IaaS cloud infrastructures as a side-effect.
- This scaling control loop is intended to be used in the execution phase of higher-level auto-scaling MAPE loops (monitoring, analysis, planning, execution) as systematized by (Pahl and Jamshidi, 2015; Ch. Qu and R. N. Calheiros and R. Buyya, 2016) and more. To some degree, the proposed control loop makes the necessity for complex and IaaS infrastructure-specific multi-cloud workflows redundant.

The remainder of this paper is **outlined** as follows. Section 2 will investigate how CNAs are being build. This is essential to understand how to avoid vendor lock-in in a pragmatic and often overlooked way. Section 3 will focus some requirements which should be fulfilled by multi-cloud capable CNAs and will show how already existing open source elastic container platforms can contribute pragmatically. The reader will see that these kind of platforms contribute to operate cloud-native applications in a resilient and elastic way. We provide a multi-cloud aware proof-of-concept in Section 4 and derive several lessons learned from the evaluation in Section 5. The presented scaling control loop is related to other work in Section 6. Similarities and differences are summarized.

2 WHAT IS A CNA?

Although the term CNA is vague, there exist similarities of various view points (Kratzke and Quint, 2017b). According to common motivations for CNA architectures are to deliver software-based solutions more quickly (**speed**), in a more fault isolating, fault tolerating, and automatic recovering way (**safety**), to enable horizontal (instead of vertical) application scaling (**scale**), and finally to handle a huge diversity of (mobile) platforms and legacy systems (**client diversity**) (Stine, 2015). (Fehling et al., 2014) propose that a CNA should be IDEAL. It should have an **isolated state**, is **distributed** in its nature, is **elastic** in a horizontal scaling way, operated via an **automated management** system and its components should be **loosely coupled**.

These common motivations and properties are addressed by several application architecture and infrastructure approaches (Balalaie et al., 2015): **Microservices** represent the decomposition of monolithic (business) systems into independently deployable services that do *"one thing well"* (Namiot and Sneps-Sneppe, 2014; Newman, 2015). The main mode of interaction between services in a cloud-native application architecture is via published and versioned APIs (**API-based collaboration**). These APIs often follow the HTTP REST-style with JSON serialization, but other protocols and serialization formats can be used as well. Single deployment units of the architecture are designed and interconnected according to a **collection of cloud-focused patterns** like the *twelve-factor app* collection, the *circuit breaker* pattern and a lot of further cloud computing patterns (Fehling et al., 2014). And finally, **self-service elastic platforms** are used to deploy and operate these microservices via self-contained deployment units (containers). These platforms provide additional operational capabilities on top of IaaS infrastructures like automated and on-demand scaling of application instances, application health management, dynamic routing and load balancing as well as aggregation of logs and metrics. Some open source examples of such kind of elastic platforms are listed in Table 1. So, this paper follows this understanding of a CNA:

> A **cloud-native application** is a distributed, elastic and horizontal scalable system composed of (micro)services which isolates state in a minimum of stateful components. The application and each self-contained deployment unit of that application is designed according to cloud-focused design patterns and operated on a self-service elastic platform. (Kratzke and Quint, 2017b)

It is essential to understand that CNAs are operated on elastic – often container-based – platforms. Therefore, the multi-cloud aware handling of these elastic platforms is focused throughout this paper.

3 MULTI-CLOUD SPECIFICS

Several transferability, awareness and security requirements come along with multi-cloud approaches (Barker et al., 2015; Petcu and Vasilakos, 2014; Toosi et al., 2014; Grozev and Buyya, 2014). We will investigate these requirements in this section and show how already existing elastic container platforms contribute to fulfill these requirements.

3.1 Transferability Requirements

Cloud computing is basically a computing model based on ubiquitous network access to a shared and virtualized pool of computing resources. The problem is, that this conceptual model is implemented by a large number of service providers in different and not necessarily standardized or compatible ways. So, **portability or transferability** has to be requested for CNA by reasons varying from optimal selection regarding utilization, costs or profits, to technology changes, as well as legal issues (Petcu and Vasilakos, 2014).

Elastic container platforms (see Table 1) integrate container hosts (nodes) into one single and higher level logical cluster. These technologies provide **self-service elastic platforms** for cloud-native applications (Stine, 2015) in an obvious but also often overlooked way. Furthermore, some of these platforms are really "bulletproofed". *Apache Mesos* (Hindman et al., 2011) has been successfully operated for years by companies like Twitter or Netflix to consolidate hundreds of thousands of compute nodes. More recent approaches are *Docker Swarm* and Google's *Kubernetes*, the open-source successor of Google's internal *Borg* system (Verma et al., 2015). (Peinl and Holzschuher, 2015) provide an excellent overview for interested readers. From the author's point of view, there are four main benefits using these elastic container platforms, starting with the **integration of single nodes (container hosts) into one logical cluster (1st benefit)**. This integration can be done within an IaaS infrastructure (for example only within *AWS*) and is mainly done for complexity management reasons. However, it is possible to deploy such elastic platforms across public and private cloud infrastructures (for example deploying some hosts of the cluster to *AWS*, some to *Google Compute Engine (GCE)* and some to an on-premise *OpenStack* infrastructure). Even if these **elastic container platforms are deployed across different cloud service providers (2nd benefit)** they can be accessed as one logical single cluster, which is of course a great benefit from a **vendor lock-in avoiding (3rd benefit)** point of view. Last but not least, these kind of platforms are designed for failure, so they have self-healing capabilities: Their auto-placement, auto-restart, auto-replication and auto-scaling features are designed to identfiy lost containers (due to whatever reasons, e.g. process failure or node unavailability). In these cases they restart containers and place them on remaining nodes (without any necessary user interaction). The cluster can be resized simply by adding or removing nodes to the cluster. Affected containers (due to a planned or unplanned node removal) will be rescheduled transparently to other available nodes. If clusters are formed up of hundreds or thousands of nodes, some single nodes are always in an invalid state and have to be replaced almost at any time. So, these features are absolutely necessary to operate large-scale elastic container platforms in a resilient way. However, exactly the same features can be used intentionally to **realize transferability requirements (4th benefit)**. For instance, if we want to migrate from *AWS* to *GCE*, we simply attach additional nodes provisioned by *GCE* to the cluster. In a second step, we shut down all nodes provided by *AWS*. The elastic platform will recognize node failures and will reschedule lost containers accordingly. From an inner point of view of the platform, expectable node failures occur and corresponding rescheduling operations are tasked. **From the outside it is a migration from one provider to another provider at run-time.** We will explain the details in Section 4. At this point it should be sufficient to get the idea. Further multi-cloud options like public cloud exits, cloud migrations, public multi-clouds, hybrid clouds, overflow processing and so on are presented in Figure 1 and can be handled using the same approach.

3.2 Awareness Requirements

Beside portability/transferability requirements, (Grozev and Buyya, 2014) emphasizes that multi-cloud applications need to have several additional **awarenesses**:

1. **Data location awareness**: The persistent data and the processing units of an application should be in the same data center (even on the same rack) and should be connected with a high-speed network. Elastic container platforms like *Kubernetes* introduced the pod concept to ensure such kind of data locality (Verma et al., 2015).
2. **Geo-location awareness:** Requests should be scheduled near the geographical location of their

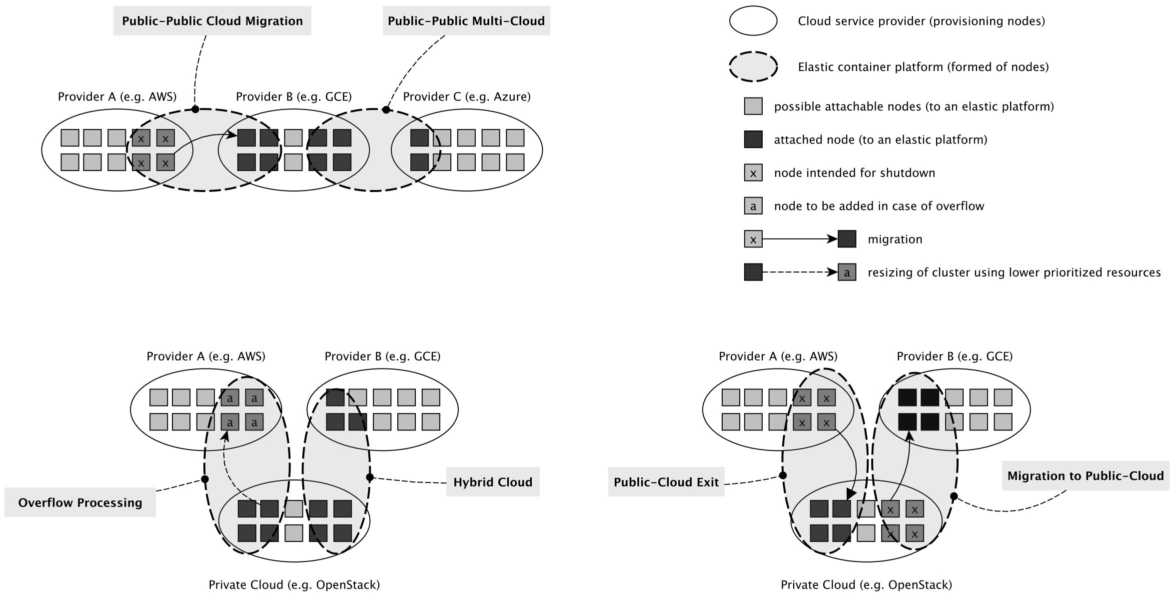


Figure 1: Some deployment options and transferability opportunities of elastic container platforms.

origin to achieve better performance.

3. **Pricing awareness:** An application scheduler needs up to date information about providers' prices to perform fiscally efficient provisioning.
4. **Legislation/policy awareness:** For some applications legislative and political considerations upon provisioning and scheduling must be taken into account. For example, some services could be required to avoid placing data outside a given country.
5. **Local resources awareness:** It is often that the usage of in-house resources should have higher priority than that of external ones (overflow processing into a public cloud).

Platforms like *Kubernetes*, *Mesos*, *Docker Swarm* are able to tag nodes of their clusters with arbitrary key/value pairs. These tags can be used to code geo-locations, prices, policies and preferred local resources (and arbitrary further aspects) and considered in scheduling strategies to place containers accordingly to the above mentioned awareness requirements. For instance, *Docker Swarm* uses constraint filters[1] for that kind of purpose. Arbitrary tags like a location tag "Germany" can be assigned to a node. This tagging can be defined in a cluster definition file and will be assigned to a node in the *install node* step shown in Figure 3(b). This tagging is considered by schedulers of the mentioned platforms. *Docker Swarm* would deploy a database container only on nodes which are tagged as "location=Germany" if a constraint filter is applied like shown.

```
docker run \
  -e constraint:location==Germany \
  couchdb
```

Kubernetes provides similar tag-based concepts called *node selectors* and even more expressive *(anti-)affinities* which are considered by the *Kubernetes* scheduler[2]. The *Marathon* framework for *Mesos* uses *constraints*[3]. Obviously, all of these concepts rely on the same idea, are tagged-based and therefore can be used to cover mentioned awareness requirements in a consistent way by platform drivers (see Figure 2).

3.3 Security Requirements

Furthermore, **security requirements** have to be considered for multi-cloud scenarios because such kind of platforms can span different providers and therefore data is likely to be submitted via the "open and unprotected" internet. Again, platforms like *Docker's Swarm Mode* (since version 1.12) provide an encrypted data and control plane via overlay networks to handle this. *Kubernetes* can be configured to use encryptable overlay network plugins like *Weave*. Accompanying network performance impacts can be contained (Kratzke and Quint, 2015b; Kratzke and Quint, 2017a).

[1] See `https://docs.docker.com/v1.11/swarm/scheduler/filter/` (last access 15th Feb. 2017)

[2] see `https://kubernetes.io/docs/user-guide/node-selection/` (last access 15th Feb. 2017)

[3] see `https://mesosphere.github.io/marathon/docs/constraints.html` (last access 15th Feb. 2017)

3.4 Summary

Existing open source elastic container platforms fulfill common transferability, awareness and security requirements in an elastic and resilient manner. The following Section 4 will explain a proof-of-concept solution how to access these opportunities using a simple control process.

4 PROOF OF CONCEPT

The proposed solution is implemented as a prototypic Ruby command line tool which could be triggered in the execution phase of a MAPE auto-scaling loop (Ch. Qu and R. N. Calheiros and R. Buyya, 2016). Although a MAPE loop needs always on components, this execution loop alone does not need any central server component, no permanent cluster connection, and can be operated on a single machine (outside the cloud). The tool scales elastic container platforms according to a simple control process (see Figure 3(a)). This control process realizes all necessary multi-cloud transferability requirements. The control process evaluates a JSON encoded cluster description format (the *intended state* of the container cluster, see Appendix: Listing 1) and the *current state* of the cluster (attached nodes, existing security groups). If the intended state differs from the current state necessary adaption tasks are deduced (attach/detachment of nodes, creation and termination of security groups). The control process reaches the intended state and terminates if no further adaption tasks can be deduced.

4.1 Description of Elastic Platforms

The description of elastic platforms in multi-cloud scenarios must consider arbitrary IaaS cloud service providers. This is done by the conceptual model shown in Figure 2. Two main approaches can be identified how public cloud service providers organize their IaaS services: project- and region-based service delivery. *GCE* and *OpenStack* infrastructures are examples following the project-based approach. To request IaaS resources like virtual machines one has to create a project and within the project one has access to resources on whatever regions. *AWS* is an example for region-based service provisioning. Resources are requested per region (Europe, US, Asia, ...) and they are not assigned to a particular project. So, one can access easily resources within a region (and across projects) but it gets complicated to access resources outside a region. Both approaches have their advantages and disadvantages as the reader will see. Region-based approaches seem to provide better adaption performances (see Section 5). However, it is not worth discussing – the approaches are given. Multi-cloud solutions simply have to consider that both approaches occur in parallel. They key idea to integrate both approaches is the introduction of a concept called `District` (see Figure 2).

One provider region or project can map to one or more `Districts` and vice versa. A `District` is simply a user defined "datacenter" which is provided by a specific cloud service provider (following the project- or region-based approach). This additional layer provides maximum flexibility in defining multi-cloud deployments of elastic container platforms. A multi-cloud deployed elastic container platform can be defined using two descriptive and JSON encoded definition formats (`cluster.json` and `districts.json`). The definition formats are exemplary explained in the Appendix in Listings 1, 2, and 3 in more details.

A `Cluster` (elastic platform) is defined as a list of `Deployments`. Both concepts are defined in a cluster definition file format (see Appendix, Listing 1). A `Deployment` defines how many nodes of a specific `Flavor` should perform a specific cluster role in a specified `District`. Most elastic container platforms are assuming two roles of nodes in a cluster. A "master" role to perform scheduling, control and management tasks and a "worker" role to execute containers. Our solution can work with arbitrary roles and role(names). However, these roles have to be considered by `Platform` drivers (see Figure 2) in their `install`, `join` and `leave` cluster hooks (see Figure 3(b)). So, role and container platform specifics can be isolated in `Platform` drivers. A typical `Deployment` can be expressed using this JSON snippet.

```
{
  "district": "gce-europe",
  "flavor": "small",
  "role": "master",
  "quantity": 3
  "tags": {
    "location": "Europe",
    "policy": "Safe-Harbour",
    "scheduling-priority": "low",
    "further": "arbitrary tags"
  }
}
```

A complete cluster can be expressed as a list of such `Deployments`. Machine `Flavors` (e.g. small, medium and large machines) and `Districts` are user defined and have to be mapped to concrete cloud service provider specifics. This is done using the `districts.json` definition file (see Appendix, Listing 3). A `District` object is responsible to execute deployments (adding or removing ma-

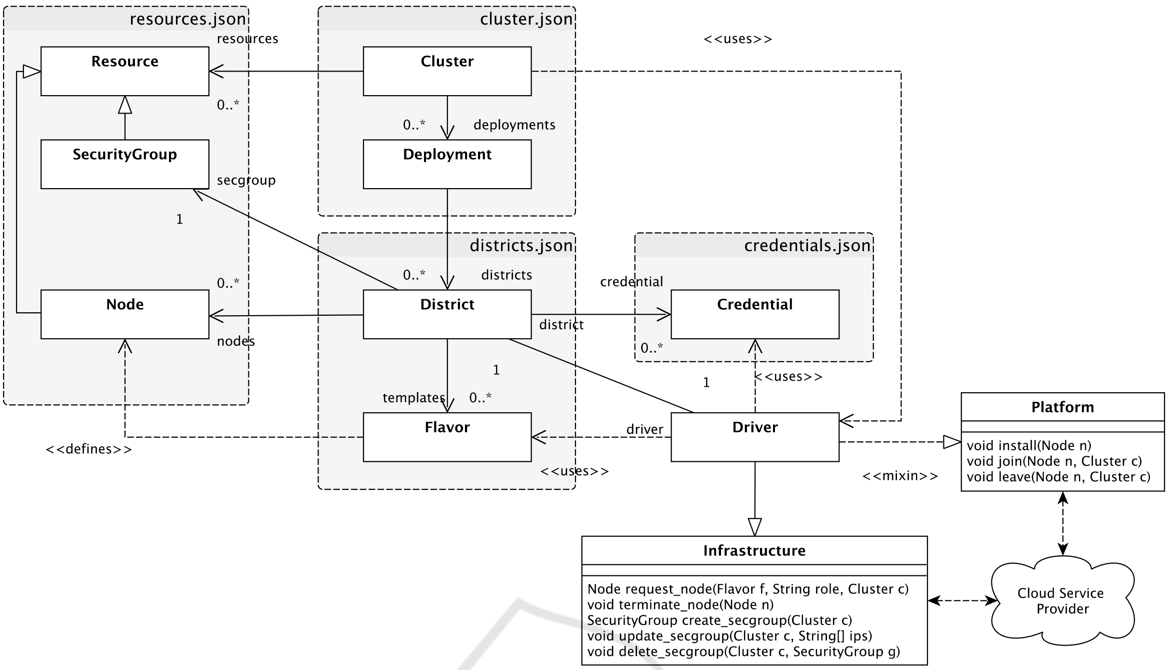


Figure 2: The conceptual model and its relation to descriptive cluster definition formats (please compare with Appendix).

chines to the cluster as well as tagging them to handle the awareness requirements mentioned in Section 3). The execution is delegated to a `Driver` object which encapsules and processes all necessary cloud service provider and elastic container platform specific requests. The driver uses access `Credentials` for authentication (see Appendix, Listing 2). The `Driver` generates `Resource` objects (`Nodes` and `SecurityGroups`) representing resources (current state of the cluster, encoded in a `resources.json` file) provided by the cloud service provider (`District`). `SecurityGroups` are used to allow internal platform communication across IaaS infrastructures. These basic security means are provided by all IaaS infrastructures under different names (firewalls, security groups, access rules, network rules, ...). This resources list is used by the control process to build the delta between the intended state (encoded in `cluster.json`) and the current state (encoded in `resources.json`).

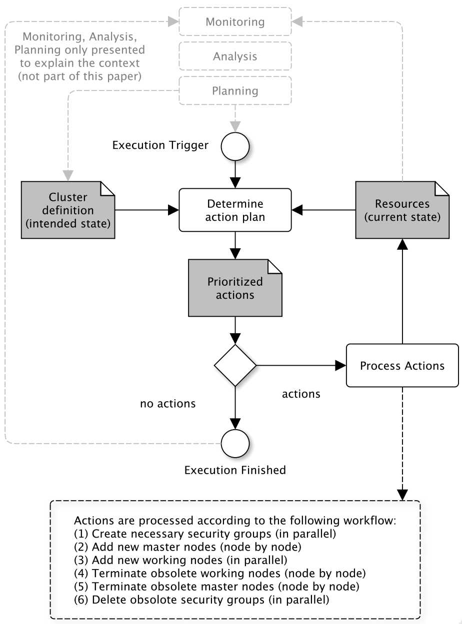
4.2 One Control Process for All

The control process shown in Figure 3(a) is responsible to command and control all necessary actions to reach the intended state (encoded in `cluster.json`). This cybernetic understanding can be used to handle common multi-cloud workflows.

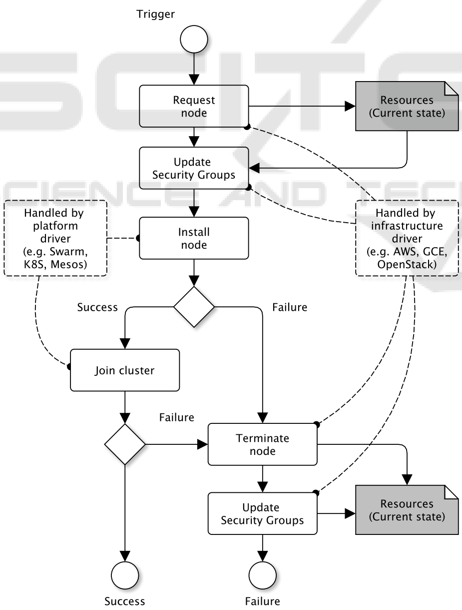
- A fresh *deployment* of a cluster can be understood as executing the control loop on an initially empty resources list.
- A *shutdown* can be expressed by setting all deployment quantities to 0.
- A *migration* from one `District` A to another `District` B can be expressed by setting all `Deployment` quantities of A to 0 and adding the former quantities of A to the quantities of B.

Having this cybernetic understanding it is easy to realize all (and more) multi-cloud deployment options and transferability opportunities shown in Figure 1. The control loop derives a prioritized *action plan* to reach the intended state. The current implementation realizes this according to the workflow shown in Figure 3(a). However, other workflow sequences might work as well and could show faster adaption cycles. The reader should be aware that the workflow must keep the affected cluster in a valid and operational state at all times. The currently implemented strategy considers practitioner experience to reduce "stress" for the affected elastic container platform. Only action steps (2) and (3) of the control loop are explained (shown in Figure 3(b)). The other steps are not presented due to triviality and page limitations.

Whenever a new node attachment is triggered by the control loop, the corresponding `Driver` is called to launch a new `Node` request. The `Node` is added to the list of requested resources (and extends therefore the current state of the cluster). Then all exist-



(a) The execution control loop (monitoring, analysis, planning are only shown to indicate the context of a full elastic setting)



(b) Add master/worker action, steps (2) and (3)

Figure 3: The control loop embedded in a MAPE loop.

ing `SecurityGroups` are updated to allow incoming network traffic from the new `Node`. These steps are handled by an IaaS `Infrastructure` driver. Next, the control is handed over to a `Platform` driver. This driver is performing necessary software installs via SSH-based scripting. Finally, the node is joined to the cluster using platform (and maybe role-)specific joining calls provided by the `Platform` driver. If *install* or *join* operations were not successful, the machine is terminated and removed from the resources list by the `Infrastructure` driver. In these cases the current state could not be extended and a next round of the control loop would do a retry. Therefore, the control-loop is automatically designed for failure and will take care to retry failed *install* and *joining* actions.

4.3 IaaS Infrastructures and Platforms

The workflows shown in Figure 3 are designed to handle arbitrary IaaS `Infrastructure`s and arbitrary elastic `Platform`s. However, the infrastructure and platform specifics must be handled as well. This is done using an extendable driver concept (see Figure 2). The classes `Platform` and `Infrastructure` are two extension points to provide support for **IaaS infrastructures** like *AWS*, *GCE*, *Azure*, *DigitalOcean*, *RackSpace*, ..., and for **elastic container platforms** like *Docker Swarm*, *Kubernetes*, *Mesos/Marathon*, *Nomad*, and so on. Infrastructures and platforms can be integrated simply by extending the `Infrastructure` class (for IaaS infrastructures) or `Platform` class (for additional elastic container platforms). Both concerns can be combined to enable the operation of a `Platform` on an IaaS `Infrastructure`. The current state of implementation provides **platform drivers** for the elastic container platforms *Kubernetes* and *Docker's Swarm-Mode* and **infrastructure drivers** for the public IaaS infrastructures *AWS*, *GCE* and the private IaaS infrastructure *OpenStack*. Due to the mentioned extension points further container `Platform`s and IaaS `Infrastructure`s are easily extendable.

5 EVALUATION

The solution was evaluated using two elastic platforms (Docker's 1.12 *Swarm Mode* and *Kubernetes 1.4*) on three public and private cloud infrastructures (*GCE, eu-west1 region*; *AWS, eu-central-1 region* and *OpenStack, research institutions private datacenter*). The platforms operated two multi-tier web applications (a Redis-based guestbook and a reference "sock-shop" application[4]. Both CNAs are often used by practitioners to demonstrate elastic container platform features.

[4]see `https://github.com/kubernetes/kubernetes/tree/master/examples/guestbook` and `https://github.com/microservices-demo/microservices-demo` (last access 19th Feb. 2017)

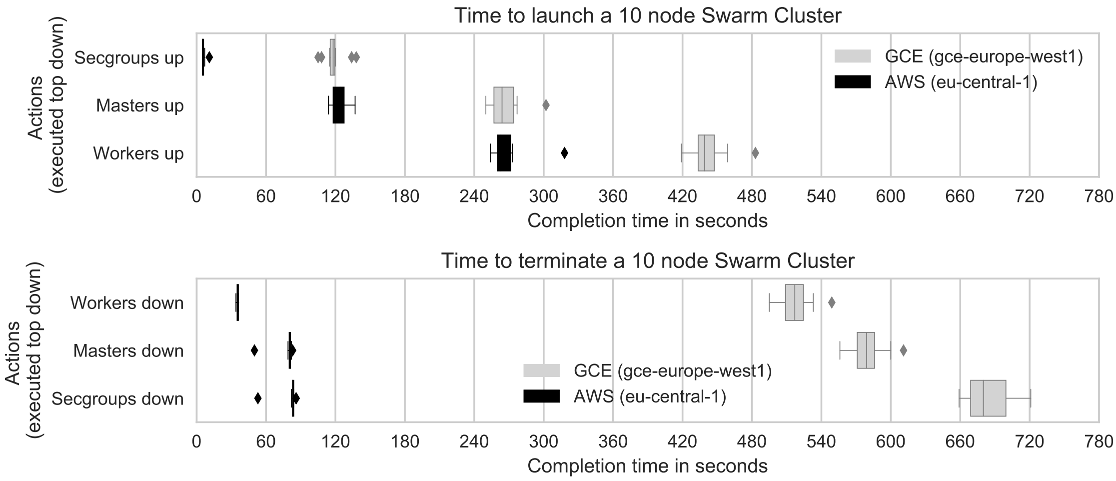


Figure 4: Launching and terminating o a elastic container platform (single-cloud experiments E1 and E2)

5.1 Experiments

The implementation was tested using a 10 node cluster composed of one master node and 9 worker nodes executing the above mentioned reference applications. The following experiments demonstrate elastic container platform deployments, terminations, and complete or partial platform transfers across different cloud service infrastructures. Additionally, the experiments were used to measure runtimes to execute these kind of operations.

- **E1:** Launch a 10 node cluster in *AWS* and *GCE* (single-cloud).
- **E2:** Terminate a 10 node cluster in *AWS* and *GCE* (single-cloud).
- **E3:** Transfer 1 node of a 10 node cluster from *AWS* to *GCE* (multi-cloud) and vice versa.
- **E4:** Transfer 5 nodes of a 10 node cluster from *AWS* to *GCE* (multi-cloud) and vice versa.
- **E5:** Transfer a complete 10 node cluster from *AWS* to *GCE* (multi-cloud) and vice versa.

To compare similar machine types in *AWS* and *GCE* it was decided to use *n1-standard-2* machine types from *GCE* and *m3.large* machine types from *AWS*. These machine types show high similarities regarding processing, networking, I/O and memory performance (Kratzke and Quint, 2015a). Additionally, a machine type on the institutes on-premise *OpenStack* infrastructure has been defined with comparable performance characteristics like the mentioned *"reference machine types"* selected from *GCE* and *AWS*.

Each experiment was repeated at least 10 times. Due to page limitations only data for *Docker Swarm* is presented. It turned out that most of runtimes are due to low level IaaS infrastructure operations and not due to elastic container platform installation/configuration and rescheduling operations. So, the data for *Kubernetes* is quite similar. It is to admit that the measured data is more suited as a performance comparison of *AWS* and *GCE* infrastructure operations than it is suited to be a performance measurement of the proposed control loop. So far, one can say that the proposed control loop is slow on slow infrastructures and fast on fast infrastructures. That is not astonishing. However, some interesting findings especially regarding software defined network aspects in multi-cloud scenarios and reactiveness of public cloud service infrastructures could be derived.

OpenStack performance is highly dependent on the physical infrastructure and detail configuration. So, although it was tested against institutes on-premise *OpenStack* private cloud infrastructure (*OpenStack* is somewhere in between *AWS* and *GCE*), it is likely that this data is not representative. That is why only data for *AWS* and *GCE* is presented.

Figure 4 shows the results of the experiments **E1** and **E2** by boxplotting the completion times when all security groups, master and worker nodes were up/-down. The cluster was in an initial operating mode when the master node was up, and it was fully operational when all worker nodes were up. The reader might be surprised, that *GCE* infrastructure operations are taking much longer than *AWS* infrastructure operations. The cluster was launched on *AWS* in approximately 260 seconds and on *GCE* in 450 seconds (median values). The analysis turned out, that the *AWS* infrastructure is much more reactive regarding adjustments of network settings than *GCE* (see SDN related processing times in Figure 5). The security groups on *AWS* can be created in approximately 10 seconds but it takes almost two minutes on *GCE*. There are similar severe performance differences when adjustments on these security groups are necessary (so when nodes are added or removed). We believe this has to do with network philosophies of both infrastructures. Security groups in *AWS* are region specific. Firewalls and networks in *GCE* are designed to be used in *GCE* projects and *GCE* projects can be deployed across all available *GCE* regions. So, *AWS* has to acivate SDN changes only in one region (that means within one datacenter). But *GCE* has to activate changes in all regions (so in all of their datacenters). From a cloud customer perspective *GCE* networking is much more comfortable but adaptions are slow compared with the *AWS* infrastructure. The runtime effects for deployment visualizes Figure 4.

The termination is even more worse. A cluster can be terminated in approximately 90 seconds on *AWS* but it takes up to 720 seconds on *GCE*. Our analysis turned out that the CLI (command line interface) of *AWS* works mainly asynchronous while the CLI of *GCE* works mainly synchronous. The control loop terminates nodes node by node in order to reduce rescheduling stress for the elastic platform (node requesting is done in parallel because a node adding normally does not involve immediate rescheduling of the workload). So, on *AWS* a node is terminated by deregistering it from its master of the elastic container platform and than its termination is launched. The CLI does not wait until termination is completed and just returns. The effect is, that nodes are deregistered sequentially from the platform (which only takes one or two seconds) and subsequent termination of all nodes is done mainly in parallel (a termination is started almost every 2 seconds but take almost a minute). The reader should be aware, that this results in much more rescheduling stress for the container platform. However, no problems with that higher stress level in the experiments were observable on the *AWS* side. The CLI of *GCE* is synchronous. So, when a node is terminated, the *GCE* CLI waits until the termination is completed (which takes approximately a minute). Additionally, every time a node is removed the *GCE* firewalls have to be adapted and this is a time consuming operation as well in *GCE* (approximately 25 seconds for *GCE* but only 10 seconds for *AWS*). That is why termination durations show these dramatic differences.

Figure 5 shows the results of the experiments **E3, E4** and **E5** (Transfer times between *AWS* and *GCE*). It was simply measured how long it took to transfer 1, 5 or all 10 cluster nodes from *AWS* to *GCE* or vice versa. Transfer speeds are dependent of the origin provider. Figure 5 shows that a transfer from *AWS* to

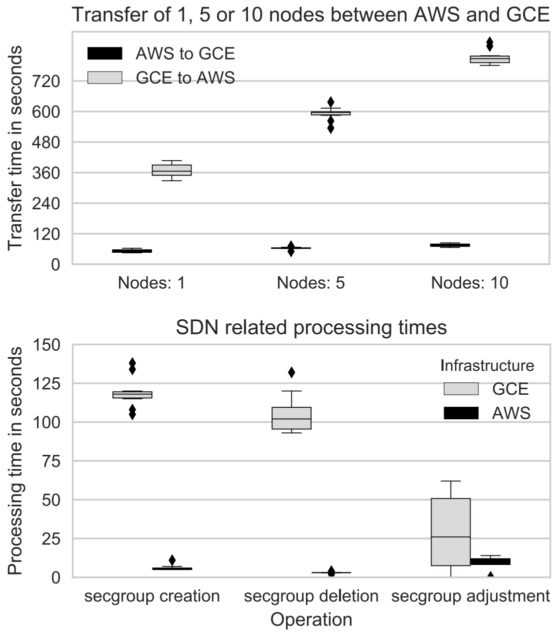


Figure 5: Detail data of multi-cloud experiments E3, E4, E5

GCE (6 minutes) is more than two times faster than from *GCE* to *AWS* (13 minutes). Furthermore, it is astonishing that a transfer of only one single node from *AWS* to *GCE* takes almost as long as a complete single-cloud cluster launch on *AWS* (both operations take approximately 4 minutes). However, a complete cluster transfer (10 nodes) from *AWS* to *GCE* is only slightly slower (6 minutes, but not 40 minutes!). A transfer from one provider to another is obviously a multi-cloud operation and multi-cloud operation always involve operations of the ”slower” provider (in the case of the experiments **E3**, **E4**, **E5** this involved SDN adjustments and node terminations). ***It turned out, that the runtime behaviour of the slowest provider is dominating the overall runtime behaviour of multi-cloud operations.*** In the analyzed case, slower *GCE* SDN related processing times and node termination times were the dominating factor to slow down operations on the *AWS* side. This can result in surprising effects. A complete cluster transfer from a ”faster” provider (*AWS*) to a ”slower” provider (*GCE*) can be done substantially faster than from a ”slower” provider to a ”faster” provider.
Taking all together, IaaS termination operations should be launched asynchronously to improve the overall multi-cloud performance.

5.2 Critical Discussion

Each cloud provider has specific requirements and the proposed control loop is designed to be generic enough to adapt to each one. The presented control loop was even able to handle completely different timing behaviors without being explicitly designed for that purpose. However, this ”genericness” obviously can not be proofed. There might be an IaaS cloud infrastructure not suitable for the proposed approach. But the reader should be aware that – intentionally – only very basic IaaS concepts are used. The control loop should work with every public or private IaaS infrastructure providing concepts like virtual machines and IP-based network access control concepts like security groups (*AWS*) or network/firewalls (*GCE*). These are very basic concepts. An IaaS infrastructure not providing these basic concepts is hardly imaginable and to the best of our knowledge not existing.

The proposed solution tries to keep the cluster in a valid and operational state under all circumstances (whatever it costs). A migration from one infrastructure A to another infrastructure B could be expressed by setting all quantities of A to 0 and all quantities of B to the former quantities of B (that is basically the experiment **E5**). The current implementation of the control loop is not very sophisticated and executes simply a worst case scaling. Node creation steps have a higher priority than node deletion steps. So, a migration increases the cluster to its double size in a first step. In a second step, the cluster will be shrinked down to its intended size in its intended infrastructure. This leaves obviously room for improvement.

Furthermore, the control loop is designed to be just the execution step of a higher order MAPE loop. If the planning step (or the operator) defines an intended state which is not reachable, the execution loop may simply have no effect. Imagine a cluster under high load. If the intended state would be set to half of the nodes (due to whatever reasons), the execution loop would not be able to reach this state. Why? Before a node is terminated by the control loop, the control loop informs the container scheduler to mark this node as unscheduleable with the intent that the container platform will reschedule all load of this node to other nodes (draining the node). For these kind of purposes elastic container platforms have operations to mark nodes as unschedulable (*Kubernetes* has the cordon command, *Docker* has a drain concept and so on). Only in the case that the container platform could successfully drain the node, the node will be deleted. However, in high load scenarios the scheduler of the container platform will simply answer that draining is not possible. The control loop will not terminate the node and will simply try to drain the next node on its list (which will not work as well). In consequence it will finish its cycle without substantially changing the current state. The analyzing step of the MAPE loop will still identify a delta between the intended and the current state and will consequently trigger the execution control loop one more time. That is not perfect but at last the cluster is kept in an operational state.

5.3 Lessons Learned

Finally, several lessons learned can be derived from performed software engineering activities which might be of interest for researchers or practitioners.

1. **Just Use Very Basic Requests to Launch, Terminate and Secure Virtual Machines.** Try to avoid cloud-init. It is not identically supported by all IaaS infrastructures especially in timing relevant public/private IP assignments. Use ssh-based scripting instead.
2. **Consider Secure Networking Across Different Providers.** If you want to bridge different IaaS cloud service providers you have to work with public IPs from the very beginning! However, this is not the default operation for most elastic platforms. Additionally, control and data plane encryption must be supported by the used overlay network of your elastic platform.
3. **Do Never Use IaaS Infrastructure Elasticity Features.** They are not 1:1 portable across providers. The elastic platform has to cover this.
4. **Separate IaaS Support and Elastic Platform Support Concerns from Each Other.** They can be solved independently from each other using two independent extension points.
5. **Describe Intended States of an Elastic Platform and Let a Control Process Take Care to Reach This Intended State.** Do not think in TOSCA-like and IaaS infrastructure specific workflows how to deploy, scale, migrate and terminate an elastic platform. All this can be solved by a single control loop.
6. **Separate Description Concerns of the Intended State.** Try to describe the general cluster as an intended state in a descriptive way. Do not mix the intended state with infrastructure specifics and access credentials.
7. **Consider What Causes Stress to an Elastic Platform.** Adding nodes to a platform is less stressfull than to remove nodes. It seems to be a good and defensive strategy to add nodes in parallel but to shutdown nodes sequentially. However, this increases the runtime of the execution phase of a MAPE loop. To investigate time optimal execution strategies could be a fruitful research direction to make MAPE loops more reactive.

8. **Respect Resilience Limitations of an Elastic Platform.** Never shutdown nodes before you attached compensating nodes (in case of transferability scaling actions) is an obvious solution! But it is likely not ressource efficient. To investigate resilient and resource efficient execution strategies could be a fruitful research direction to optimize MAPE loops for transferability scenarios.
9. **Platform Roles Increase Avoidable Deployment Complexity.** Elastic container platforms should be more P2P-like and composed of homogeneous and equal nodes. This could be a fruitful research direction either.
10. **Asynchronous CLIs or APIs Are Especially Preferable for Terminating Operatings.** Elastic container platforms will show much more reactive behavior (and faster adaption cycles) if operated on IaaS infrastructures providing asynchronous terminating operations (see Figure 4).

6 RELATED WORK

According to (Barker et al., 2015; Petcu and Vasilakos, 2014; Toosi et al., 2014; Grozev and Buyya, 2014) there are several promising approaches dealing with multi-cloud scenarios. However, none of these surveys identified elastic container platforms as a viable option. Just (Petcu and Vasilakos, 2014) identify the need to *"adopt open-source platforms"* and *"mechanisms for real-time migration"* at runtime level but did not identified (nor integrated) concrete and existing platforms or solutions. All surveys identified approaches fitting mainly in the following fields: **Volunteer federations** for groups of *"cloud providers collaborating voluntarily with each other to exchange resources"* (Grozev and Buyya, 2014). **Independent federations** (or multi-clouds) *"when multiple clouds are used in aggregation by an application or its broker. This approach is essentially independent of the cloud provider"* and focus the client-side of cloud computing (Toosi et al., 2014).

This contribution focus independent federations (multi-clouds). We do not propose a broker-based solution (Barker et al., 2015) because cloud-brokers have the tendency just to shift the vendor lock-in problem to a broker. However, the following approaches show some similarities. The following paragraphs briefly explain how the proposed approach is different.

Approaches like **OPTIMIS** (Ferrer et al., 2012), **ConTrail** (Carlini et al., 2012) or **multi-cloud PaaS platforms** (Paraiso et al., 2012) enable dynamic provisioning of cloud services targeting multi-cloud architectures. These solutions have to provide a lot of plugins to support possible implementation languages. (Paraiso et al., 2012) mention at least 19 different plugins (just for a research prototype). This increases the inner complexity of such kind of solutions. Container-based approaches might be better suited to handle this kind of complexity. Approaches like **mOSAIC** (Petcu et al., 2011) or **Cloud4SOA** (Kamateri et al., 2013) assume that an application can be divided into components according to a service oriented application architecture (SOA). These approaches rely that applications are bound to a specific run-time environment. This is true for the proposed approach as well. However, this paper proposes a solution where the run-time environment (elastic container platform) is up to a user decision as well.

The proposed deployment description format is based on JSON. And it is not at all unlike the kind of deployment description languages used by **TOSCA** (Brogi et al., 2014), **CAMEL** (A. Rossini, 2015) or **CloudML** (Lushpenko et al., 2015). In fact, some EC-funded projects like **PaaSage**[5] (Baur and Domaschka, 2016) combine such deployment specification languages with runtime environments. Nonetheless, this contribution is focused on a more container-centric approach. Finally, several **libraries** have been developed in recent years like **JClouds**, **LibCloud**, **DeltaCloud**, **SimpleCloud**, **Nuvem**, **CPIM** (Giove et al., 2013) to name a few. All these libraries unify differences in the management APIs of clouds and provide control over the provisioning of resources across geographical locations. Also **configuration management tools** like **Chef** or **Puppet** address deployments. But, these solutions do not provide any (elastic) runtime environments.

Taking all together, the proposed approach intends to be more "pragmatic", "lightweight" and complexity hiding using existing elastic container platforms. On the downside, it might be only applicable for container-based applications. But to use container platforms gets more and more common in CNA engineering.

7 CONCLUSIONS

As it was emphasized throughout this contribution, elastic container platforms are a viable and pragmatic option to support multi-cloud handling which should be considered more consequently in multi-cloud research. Elastic container platforms provide inherent –

[5]see `http://www.paasage.eu/` (last access 15th Feb. 2017)

but astonishingly often overlooked – multi-cloud support. Multi-cloud workflows to deploy, scale, migrate and terminate elastic container platforms across different public and private IaaS cloud infrastructures can be complex and challenging. Instead of that, this paper proposed to define an intended multi-cloud state of an elastic platform and let a control process take care to reach this state. This paper presented an implementation of such kind of control process being able to migrate and operate elastic container platforms across different cloud-service providers. It was possible to transfer a 10 node cluster from *AWS* to *GCE* in approximately six minutes. This control process can be used as execution phase in auto-scaling MAPE loops (Ch. Qu and R. N. Calheiros and R. Buyya, 2016). The presented cybernetic approach could evaluated successfully using common elastic container platforms (*Docker's Swarm Mode*, *Kubernetes*) and IaaS infrastructures (*AWS*, *GCE*, and *OpenStack*). Furthermore, fruitful lessons learned about runtime behaviors of IaaS operations and promising research directions like more P2P-based and control-loop based designs of elastic container platforms could be derived. The reader should be aware that the presented approach might be not feasible for applications and services outside the scope of CNAs. Nevertheless, it seems that CNA architectures are getting a predominant architectural style how to deploy and operate services in the cloud.

ACKNOWLEDGEMENTS

This research is funded by German Federal Ministry of Education and Research (03FH021PX4). I would like to thank Peter Quint, Christian Stüben, and Arne Salveter for their hard work and their contributions to the Project Cloud TRANSIT. Finally, let me thank all anonymous reviewers for their valuable feedback that improved this paper.

REFERENCES

A. Rossini (2015). Cloud Application Modelling and Execution Language (CAMEL) and the PaaSage Workflow. In *Advances in Service-Oriented and Cloud Computing—Workshops of ESOCC 2015*, volume 567, pages 437–439.

Balalaie, A., Heydarnoori, A., and Jamshidi, P. (2015). Migrating to Cloud-Native Architectures Using Microservices: An Experience Report. In *1st Int. Workshop on Cloud Adoption and Migration (CloudWay)*, Taormina, Italy.

Barker, A., Varghese, B., and Thai, L. (2015). Cloud Services Brokerage: A Survey and Research Roadmap. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 1029–1032. IEEE.

Baur, D. and Domaschka, J. (2016). Experiences from Building a Cross-cloud Orchestration Tool. In *Proc. of the 3rd Workshop on CrossCloud Infrastructures & Platforms*, CrossCloud '16, pages 4:1–4:6, New York, NY, USA. ACM.

Brogi, A., Soldani, J., and Wang, P. (2014). *TOSCA in a Nutshell: Promises and Perspectives*, pages 171–186. Springer Berlin Heidelberg, Berlin, Heidelberg.

Carlini, E., Coppola, M., Dazzi, P., Ricci, L., and Righetti, G. (2012). Cloud Federations in Contrail. pages 159–168. Springer Berlin Heidelberg.

Ch. Qu and R. N. Calheiros and R. Buyya (2016). Autoscaling Web Applications in Clouds: A Taxonomy and Survey. *CoRR*, abs/1609.09224.

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.

Ferrer, A. J., Hernandez, F., Tordsson, J., Elmroth, E., Ali-Eldin, A., Zsigri, C., Sirvent, R., Guitart, J., Badia, R. M., Djemame, K., Ziegler, W., Dimitrakos, T., Nair, S. K., Kousiouris, G., Konstanteli, K., Varvarigou, T., Hudzia, B., Kipp, A., Wesner, S., Corrales, M., Forgo, N., Sharif, T., and Sheridan, C. (2012). OPTIMIS: A holistic approach to cloud service provisioning. *Future Generation Computer Systems*, 28(1):66–77.

Giove, F., Longoni, D., Yancheshmeh, M. S., Ardagna, D., and Di Nitto, E. (2013). An Approach for the Development of Portable Applications on PaaS Clouds. In *Proceedings of the 3rd International Conference on Cloud Computing and Services Science*, pages 591–601. SciTePress - Science and and Technology Publications.

Grozev, N. and Buyya, R. (2014). Inter-Cloud architectures and application brokering: taxonomy and survey. *Software: Practice and Experience*, 44(3):369–390.

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., Shenker, S., and Stoica, I. (2011). Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *8th USENIX Conf. on Networked systems design and implementation (NSDI'11)*, volume 11.

Kamateri, E., Loutas, N., Zeginis, D., Ahtes, J., D'Andria, F., Bocconi, S., Gouvas, P., Ledakis, G., Ravagli, F., Lobunets, O., and Tarabanis, K. A. (2013). Cloud4SOA: A Semantic-Interoperability PaaS Solution for Multi-cloud Platform Management and Portability. pages 64–78. Springer Berlin Heidelberg.

Kratzke, N. and Quint, P.-C. (2015a). About Automatic Benchmarking of IaaS Cloud Service Providers for a World of Container Clusters. *Journal of Cloud Computing Research*, 1(1):16–34.

Kratzke, N. and Quint, P.-C. (2015b). How to Operate Container Clusters more Efficiently? Some Insights Concerning Containers, Software-Defined-Networks, and their sometimes Counterintuitive Impact on Network

Performance. *International Journal On Advances in Networks and Services*, 8(3&4):203–214.

Kratzke, N. and Quint, P.-C. (2017a). Investigation of Impacts on Network Performance in the Advance of a Microservice Design. In Helfert, M., Ferguson, D., Munoz, V. M., and Cardoso, J., editors, *Cloud Computing and Services Science Selected Papers*, Communications in Computer and Information Science (CCIS). Springer.

Kratzke, N. and Quint, P.-C. (2017b). Understanding Cloud-native Applications after 10 Years of Cloud Computing - A Systematic Mapping Study. *Journal of Systems and Software*, 126(April):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 Bencomo, N., Götz, S., and Song, H., editors, *MRT 2015: 10th Int. Workshop on Models@run.time, co-located with MODELS 2015: 18th ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems*, volume 1474 of *CEUR Workshop Proceedings*. CEUR.

Namiot, D. and Sneps-Sneppe, M. (2014). On microservices architecture. *Int. Journal of Open Information Technologies*, 2(9).

Newman, S. (2015). *Building Microservices*. O'Reilly Media, Incorporated.

Pahl, C. and Jamshidi, P. (2015). Software architecture for the cloud – A roadmap towards control-theoretic, model-based cloud architecture. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9278.

Paraiso, F., Haderer, N., Merle, P., Rouvoy, R., and Seinturier, L. (2012). A Federated Multi-cloud PaaS Infrastructure. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 392–399. IEEE.

Peinl, R. and Holzschuher, F. (2015). The Docker Ecosystem Needs Consolidation. In *5th Int. Conf. on Cloud Computing and Services Science (CLOSER 2015)*, pages 535–542.

Petcu, D., Craciun, C., Neagul, M., Lazcanotegui, I., and Rak, M. (2011). Building an interoperability API for Sky computing. In *2011 International Conference on High Performance Computing & Simulation*, pages 405–411. IEEE.

Petcu, D. and Vasilakos, A. V. (2014). Portability in clouds: approaches and research opportunities. *Scalable Computing: Practice and Experience*, 15(3):251–270.

Stine, M. (2015). *Migrating to Cloud-Native Application Architectures*. O'Reilly.

Toosi, A. N., Calheiros, R. N., and Buyya, R. (2014). Interconnected Cloud Computing Environments. *ACM Computing Surveys*, 47(1):1–47.

Verma, A., Pedrosa, L., Korupolu, M. R., Oppenheimer, D., Tune, E., and Wilkes, J. (2015). Large-scale cluster management at Google with Borg. In *10th. Europ. Conf. on Computer Systems (EuroSys '15)*, Bordeaux, France.

APPENDIX

Exemplary Cluster Definition File (JSON)

This **cluster definition** file defines a *Swarm* cluster with the intended state to be deployed in two districts provided by two providers GCE and AWS. It defines three type of user defined node types (flavors): *small*, *med*, and *large*. 3 master and 3 worker nodes should be deployed on *small* virtual machine types in district *gce-europe*. 10 worker nodes should be deployed on *small* virtual machine types in district *aws-europe*. The flavors *small*, *med*, *large* are defined in Listing 3.

```
{ "type": "cluster",
  "platform": "Swarm",
  // [...], Simplified for readability
  "flavors": ["small","med","large"],
  "deployments": [
    { "district": "gce-europe",
      "flavor": "small",
      "role": "master",
      "quantity": 3
    },
    { "district": "gce-europe",
      "flavor": "small",
      "role": "worker",
      "quantity": 3
    },
    { "district": "aws-europe",
      "flavor": "small",
      "role": "worker",
      "quantity": 10
    }
  ]
}
```

Listing 1: Cluster Definition (`cluster.json`).

Exemplary Credentials File (JSON)

The following **credential file** provides access credentials for customer specific GCE and AWS accounts as identified by the district definition file (*gce_default* and *aws_default*).

```
[ { "type": "credential",
    "id": "gce_default",
    "provider": "gce",
    "gce_key_file": "path-to-key.json"
  },
  { "type": "credential",
    "id": "aws_default",
    "provider": "aws",
    "aws_access_key_id": "AKID",
    "aws_secret_access_key": "SECRET"
  }
]
```

Listing 2: Credentials (`credentials.json`).

Exemplary District Definition File (JSON)

The following **district definition** defines provider specific settings and mappings. The user defined district *gce-europe* should be realized using the provider specific GCE zones *europe-west1-b* and *europe-west1-c*. Necessary and provider specific access settings like project identifiers, regions, and credentials are provided as well. User defined flavors (see cluster definition format above) are mapped to concrete provider specific machine types. The same is done for the AWS district *aws-europe*.


```json
[
  {
    "type": "district",
    "id": "gce-europe",
    "provider": "gce",
    "credential_id": "gce_default",
    "gce_project_id": "your-proj-id",
    "gce_region": "europe-west1",
    "gce_zones": [
      "europe-west1-b",
      "europe-west1-c"
    ],
    "flavors": [
      { "flavor": "small",
        "machine_type": "n1-standard-1"
      },
      { "flavor": "med",
        "machine_type": "n1-standard-2"
      },
      { "flavor": "large",
        "machine_type": "n1-standard-4"
      }
    ]
  },
  {
    "type": "district",
    "id": "aws-europe",
    "provider": "aws",
    "credential_id": "aws_default",
    "aws_region": "eu-central-1",
    "flavors": [
      { "flavor": "small",
        "instance_type": "m3.medium"
      },
      { "flavor": "med",
        "instance_type": "m4.large"
      },
      { "flavor": "large",
        "instance_type": "m4.xlarge"
      }
    ]
  }
]
```


Listing 3: District Definitions (`districts.json`).