Self-managing cloud-native applications: design, implementation, and experience

Giovanni Toffetti, Sandro Brunner, Martin Blöchlinger, Josef Spillner, Thomas Michael Bohnert

Zurich University of Applied Sciences, School of Engineering
Service Prototyping Lab (blog.zhaw.ch/icclab/)
8401 Winterthur, Switzerland
{toff,brnr,bloe,spio,bohe}@zhaw.ch

Abstract

Running applications in the cloud efficiently requires much more than deploying software in virtual machines. Cloud applications have to be *continuously managed*: 1) to adjust their resources to the incoming load and 2) to face transient failures replicating and restarting components to provide resiliency on unreliable infrastructure. Continuous management *monitors* application and infrastructural metrics to provide automated and responsive reactions to failures (*health management*) and changing environmental conditions (*auto-scaling*) minimizing human intervention.

In the current practice, management functionalities are provided as infrastructural or third party services. In both cases they are external to the application deployment. We claim that this approach has intrinsic limits, namely that separating management functionalities from the application prevents them from naturally scaling with the application and requires additional management code and human intervention. Moreover, using infrastructure provider services for management functionalities results in vendor lock-in effectively preventing cloud applications to adapt and run on the most effective cloud for the job.

In this paper we discuss the main characteristics of cloud native applications, propose a novel architecture that enables scalable and resilient self-managing applications in the cloud, and relate on our experience in porting a legacy application to the cloud applying cloud-native principles.

Keywords:
Cloud-native applications, CNA, micro services, health-management,

distributed systems, auto-scaling, IaaS, PaaS, container-based applications

1. Introduction

After a phase driven mainly by early adopters, cloud computing is now being embraced by most companies. Not only new applications are developed to be run in the cloud, but legacy workloads are increasingly being adapted and transformed to leverage the dominant cloud computing models. A suitable cloud application design was published previously by the authors [Toffetti et al. (2015)] in the proceedings of the First International Workshop on Automated Incident Management in the Cloud (AIMC'15). With respect to that initial position paper, this article relates on our experience implementing the design we propose with a specific set of technologies and the evaluation of the non-functional behavior of the implementation with respect to scalability and resilience.

There are several advantages in embracing the cloud, but in essence they typically fall into two categories: either *operational* (flexibility/speed) or *economical* (costs) reasons. From the former perspective, cloud computing offers fast self-service provisioning and task automation through application programming interfaces (APIs) which allow to deploy and remove resources instantly, reduce wait time for provisioning development/test/production environments, enabling improved agility and time-to-market facing business changes. The bottom line is *increased productivity*. From the economical perspective, the *pay-per-use* model means that no upfront investment is needed for acquiring IT resources or for maintaining them, as companies pay only for allocated resources and subscribed services. Moreover, by handing off the responsibility of maintaining physical IT infrastructure, companies can avoid capital expenses (*capex*) in favor of usage-aligned operational expenses (*opex*) and can focus on development rather than operations support.

An extensive set of architectural patterns and best practices for cloud application development have been distilled, see for instance Wilder (2012); Fehling et al. (2014); Homer et al. (2014).

However, day-to-day cloud application development is still far from fully embracing these patterns. Most companies have just reached the point of adopting hardware virtualization (i.e., VMs). Innovation leaders have already moved on to successfully deploying newer, more productive patterns, like microservices, based on light-weight virtualization (i.e., containers).

On one hand, a pay-per-use model only brings cost savings with respect to a dedicated (statically sized) system solution if 1) an application has varying load over time and 2) the application provider is able to allocate the "right" amount of resources to it, avoiding both over-provisioning (paying for unneeded resources) and under-provisioning resulting in QoS degradation. On the other hand, years of cloud development experience have taught practitioners that commodity server hardware and network switches break often. Failure domains help isolate problems, but one should "plan for failure", striving to produce resilient applications on unreliable infrastructure, without compromising their elastic scalability.

In this article we relate on our experience in porting a legacy Web application to the cloud, adopting a novel design pattern for self-managing cloud native applications. This enables vendor independence and reduced costs with respect to relying on IaaS/PaaS and third party vendor services.

The main contributions of this article are: 1) a definition of cloud-native applications and their desired characteristics, 2) a distributed architecture for self-managing (micro) services, and 3) a report on our experiences and lessons learnt applying the proposed architecture to a legacy application brought to the cloud.

2. Cloud-native applications

Any application that runs on a cloud infrastructure is a "cloud application", but a "cloud-native application" (CNA from here on) is an application that has been *specifically designed* to run in a cloud environment.

2.1. CNA: definitions and requirements

We can derive the salient characteristics of CNA from the main aspects of the cloud computing paradigm. As defined in Mell and Grance (2011), there are five essential characteristics of cloud computing: *on-demand self service*, *broad network access*, *resource pooling*, *rapid elasticity* and *measured service*. In actual practice the *cloud infrastructure* is the *enabler* of these essential characteristics. Due to the economy of scale, infrastructure installations are large and typically built of *commodity hardware* so that failures are the norm rather than the exception [Verma et al. (2015)]. Finally, cloud applications often rely on third-party services, as part of the application functionality, support (e.g., monitoring) or both. Third-party services might also fail or offer insufficient quality of service.

Given the considerations above, we can define the main requirements of CNA as:

- **Resilience**: CNA have to anticipate failures and fluctuation in quality of both cloud resources and third-party services needed to implement an application to remain available during outages. *Resource pooling* in the cloud implies that unexpected fluctuations of the infrastructure performance (e.g., noisy neighbor problem in multi-tenant systems) need to be expected and managed accordingly.

- **Elasticity**: CNA need to support adjusting their capacity by adding or removing resources to provide the required QoS in face of load variation avoiding over- and under-provisioning. In other terms, cloud-native applications should take full advantage of the cloud being a *measured service* offering *on-demand self-service* and *rapid elasticity*.

It should be clear that resilience is the first goal to be attained to achieve a functioning and available application in the cloud, while scalability deals with load variation and operational cost reduction. Resilience in the cloud is typically addressed using redundant resources. Formulating the trade-off between redundancy and operational cost reduction is a business decision.

The principles identified in the "12 factor app" methodology [Adam (2012)] focus not only on several aspects that impact on resiliency and scalability (e.g., dependencies, configuration in environment, backing services as attached resources, stateless processes, port-binding, concurrency via process model, disposability) of Web applications, but also the more general development and operations process (e.g., one codebase, build-release-run, dev/prod parity, administrative processes). Many of the best practices in current cloud development stem from these principles.

2.2. Current state of cloud development practice

Cloud computing is novel and economically more viable with respect to traditional enterprise-grade systems also because it relies on self-managed software automation (restarting components) rather than more expensive hardware redundancy to provide resilience and availability on top of commodity hardware [Wardley (2011)]. However, many applications deployed in the cloud today are simply legacy applications that have been placed in VMs without changes of architecture or assumptions on the underlying infrastructure. Failing to adjust cost, performance and complexity expectations, and

assuming the same reliability of resources and services in a traditional data center as in a public cloud can cost dearly, both in terms of technical failure and economical loss.

In order to achieve resilience and scalability, cloud applications have to be continuously *monitored*, analyzing their application-specific and infrastructural metrics to provide automated and responsive reactions to failures (*health management functionality*) and changing environmental conditions (*auto-scaling functionality*), minimizing human intervention.

The current state of the art in monitoring, health management, and scaling consists of one of the following options: a) using services from the infrastructure provider (e.g., Amazon CloudWatch[1] and Auto Scaling[2] or Google Instance Group Manager[3]) with a default or a custom provided policy, b) leveraging a third-party service (e.g., Rightscale[4], New Relic[5]), c) building an ad-hoc solution using available components (e.g., Netflix Scryer[6], logstash[7]). Both infrastructure providers and third-party services are footsteps on a path leading to vendor lock-in, are paid services, and moreover they may themselves suffer from outages. Ad-hoc solutions can be hard to engineer, especially because they have to scale seamlessly with the application they monitor and manage, in other terms *they have to be themselves resilient and scalable*.

All the management approaches proposed above have one common characteristic: their logic is run in isolation from the managed application, as an external service/process. In this article we claim that this approach has intrinsic limits and we argue that one possible solution is to *build management functionalities within the managed application itself*, resulting in monitoring, health management, and scaling functionalities that naturally adapt to the managed application and its dynamic nature. Moreover, self-managing applications are fundamental enablers of vendor-independent multi-cloud applications. We propose a decomposition of the application into stateful and stateless containers, following the microservices paradigm.

---

[1] https://aws.amazon.com/cloudwatch

[2] https://aws.amazon.com/autoscaling

[3] https://cloud.google.com/compute/docs/autoscaler

[4] http://www.rightscale.com

[5] https://newrelic.com

[6] http://techblog.netflix.com/2013/11/scryer-netflixs-predictive-auto-scaling.html

[7] https://www.elastic.co/products/logstash

3. Self-managing (micro) services

The main contributions of this article is a high-level distributed architecture that can be used to implement self-managing cloud native applications.

The idea is that just as there are best practices to build reliable services on the cloud by leveraging distributed algorithms and components, so can *management functionalities* (e.g., health-management, auto-scaling, adaptive service placement) be implemented as resilient distributed applications.

More in detail, the idea is to leverage modern distributed in-memory key-value store solutions (KV-store; e.g. Consul[8], Zookeeper[9], Etcd[10], Amazon Dynamo [DeCandia et al. (2007)], Pahoehoe [Anderson et al. (2010)]) with strong or eventual consistency guarantees. They are used both to store the "state" of each management functionality and to facilitate the internal consensus algorithm for leader election and assignment of management functionalities to cluster nodes. In this way, management functionalities become stateless and, if any of the management nodes were to fail, the corresponding logic can be restarted on another one with the same state. More concretely, any management functionality (e.g., the autoscaling logic) can be deployed within an atomic service as a stateless application component to make the service self-managing in that aspect. If the autoscaling logic or the machine hosting it were to fail, the health management functionality would restart it, and the distributed key-value store would still hold its latest state.

With the same approach, hierarchies of configuration clusters can be used to delegate atomic service scaling to the components, and atomic service composition and lifecycle to service elected leaders. What we propose integrates naturally with the common best practices of cloud orchestration and distributed configuration that we will discuss in the following sections.

Self-managing microservice compositions. By generalization, and building on the concept of service composability, the same architecture can be employed to deploy self-managing service compositions or applications using the microservice architectural pattern [Lewis and Fowler (2014)].

A microservice-oriented application can be represented with a type graph of microservices that invoke each other, and an instance graph representing

[8] https://www.consul.io

[9] https://zookeeper.apache.org/

[10] https://github.com/coreos/etcd

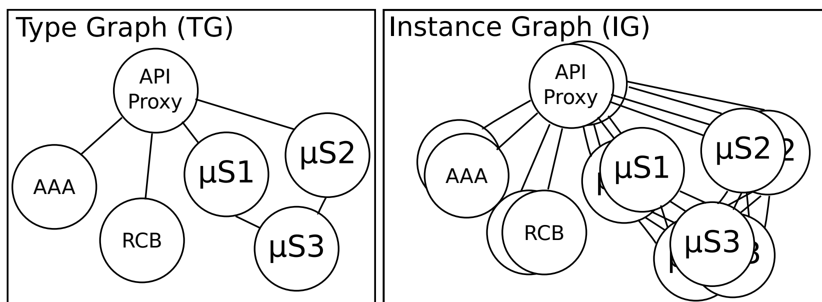

Figure 1: Type graph and instance graph of an example microservices based application

the multiple instances of microservices that are running to provide resilience and performance guarantees (e.g., as in Figure 1).

In microservice architectures, several patterns are used to guarantee resilient, fail-fast behavior. For instance, the circuit-breaker pattern [Nygard (2007)] or client-side load balancing such as in the Netflix Ribbon library[11]. The typical deployment has multiple instances of the same microservice running at the same time, possibly with underlying data synchronization mechanisms for stateful services. The rationale behind this choice is to be able to deploy microservice instances across data centers and infrastructure service providers and letting each microservice quickly adjust to failures by providing alternative endpoints for each service type.

In Figure 2, we provide an intuitive representation of how multiple KV-store clusters can be used to implement self-managing microservice applications across cloud providers. Each microservice is deployed with its own KV-store cluster for internal configuration management and discovery among components. Local management functionalities (e.g., component health management, scaling components) are delegated to nodes in the local cluster.

Another KV-store cluster is used at “global” (application) level. This “composition cluster” is used both for endpoint discovery across microservices and leader election to start monitoring, auto-scaling, and health management functionalities at service composition level. Other application-level decisions like for instance micro-service placement across clouds depending on latencies and costs, or traffic routing across microservices can be implemented as management logic in the composition cluster.

[11] http://techblog.netflix.com/2013/01/announcing-ribbon-tying-netflix-mid.html

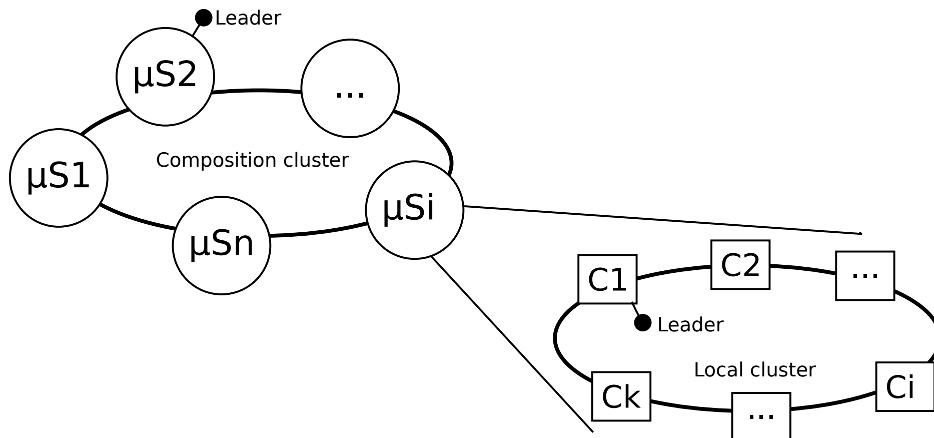

Figure 2: Hierarchical KV-store clusters for microservices management

Combined with placement across failure domains, the proposed architecture enables distributed hierarchical self management, akin to an organism (i.e., the composed service) that is able to recreate its cells to maintain its morphology while each cell (i.e., each microservice) is a living self-managing element.

3.1. Atomic service example

In this subsection we provide an example of how to apply the concept of self-managing services to a monolithic Web application acting as a single atomic service which is gradually converted to a CNA.

The functionalities needed in our example are going to be component discovery and configuration, health-management, monitoring and auto-scaling. In order to introduce them, we are also going to introduce the concepts of orchestration, distributed configuration, and service endpoint discovery in the following paragraphs.

3.1.1. Cloud service orchestration

Infrastructure as a service offers APIs to deploy and dimiss compute, network, and storage resources. However, the advantages of on-demand resource deployment would be limited if it could not be *automated*. Services and applications typically use a set of interconnected compute, storage, and network resources to achieve their specific functionality. In order to automate their deployment and configuration in a consistent and reusable manner, deployment automation tools and languages (e.g., Amazon Cloud

Formation[12], OpenStack Heat[13], TOSCA[14]) have been introduced. Generalizing, they typically consist of a language for the declarative description of the needed resources and their interdependencies (service template) combined with an engine that from the template builds a dependency graph and manages the ordered deployment of resources.

Cloud orchestration [Karagiannis et al. (2014)] is an abstraction of deployment automation. Services are defined by a *type graph* representing the needed resources and connection topology. Each time a service of a given type needs to be instantiated, an orchestrator is started with the aim of deploying and configuring the needed resources (possibly using deployment automation tools as actuators). The abstraction w.r.t. deployment automation comes from the fact that cloud orchestration is *arbitrarily composable*: the orchestration logic of a composed service triggers the orchestration of its (atomic or composed) service components creating and running as many orchestrator instances as needed.

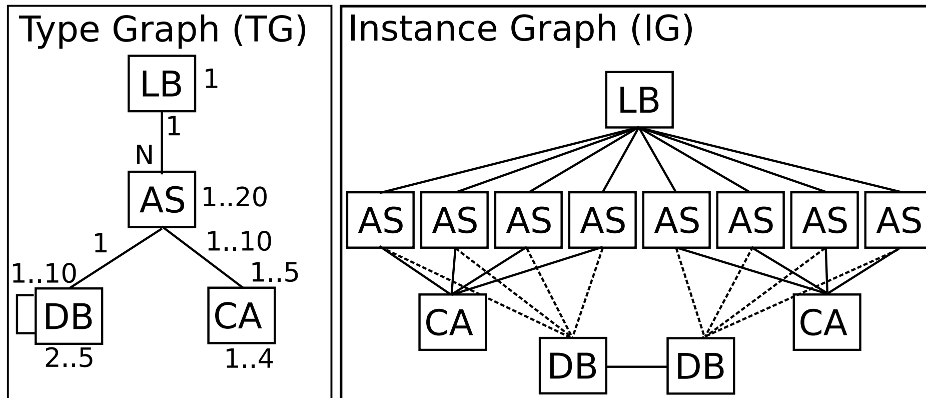

Figure 3: Type graph of a simple Web application with caching (left), example of instance graph of the same application (right)

Each orchestrator has its own representation of the deployed service topology and its components in the *instance graph*. In Figure 3 we provide an example of a type graph (TG) for a simple Web application with caching (left). The application topology allows a load balancer (LB) forwarding requests to

[12]https://aws.amazon.com/cloudformation/

[13]https://wiki.openstack.org/wiki/Heat

[14]http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html

up to 20 application servers (AS) that are connected to respectively maximum 5 and 4 database (DB) and caching (CA) instances. Cardinalities on edges represent the minimum and maximum number of allowed connections among instances. For example, a CA node can serve up to 5 AS nodes. These cardinalities are typically derived from engineering experience [Gandhi et al. (2012)]. The right graph in Figure 3 is the instance graph (IG) for the same application. In algebraic graph terminology we can say that the instance graph "is typed" over the type graph, with the semantics that the topology respects the type graphs' topology and cardinalities [Ehrig et al. (2006)].

3.1.2. Distributed configuration and service endpoint discovery

A common problem in cloud computing development is the configuration of service components and their dependencies. The main reason it is challenging is due to the dynamic nature of cloud applications. Virtual machines (or containers) are dynamically provisioned and their endpoints (the IP addresses and ports at which services can be reached) are only known after resources have been provisioned and components started. This is what is commonly known as service endpoint discovery. Different solutions for distributed cloud configurations have been proposed both in academic literature and open source communities, most of them sharing common characteristics such as a consensus mechanism and a distributed KV-store API, as presented in the self-managing microservices concept. In this work we will consider Etcd which is our preferred choice due to its simplicity to deploy and its extensive documentation.

According to its self-description, Etcd is a "distributed, consistent key value store for shared configuration and service discovery". The high-level function is simple: multiple nodes run an Etcd server, are connected with each other forming a cluster, and a consensus algorithm (Raft[15] in this case) is used for fault tolerance and consistency of the KV-store.

Nodes that form part of the same cluster share a common token and can discover each other by using a global public node registry service. Alternatively, a dedicated private node registry service can be run anywhere[16].

In Etcd the key space is hierarchical and organized in directories. Both keys and directories are generally referred as "nodes". Node values can be

[15] http://raftconsensus.github.io/

[16] For example see http://blog.zhaw.ch/icclab/setup-a-kubernetes-cluster-on-openstack-with-heat "dedicated Etcd host"

set and retrieved by Etcd clients over a REST interface. Node values can also be "watched" by clients which receive a notification whenever the value of a node changes.

The typical usage of Etcd for service configuration is to automatically update component configurations whenever there is a relevant change in the system. For instance, referring to our example application in Figure 3, the load balancer component can use Etcd to watch a directory listing all the application servers and reconfigure its forwarding policy as soon as a new application server is started. The "contract" for this reconfiguration simply requires application servers to know where to register when they are started.

The consensus algorithm underneath Etcd also provides leader election functionality, so that one node of the cluster is recognized as the sole "coordinator" by all other nodes. We will extensively use this functionality in the self-managing architecture we propose in the next section.

3.2. Component deployment and discovery

The initial deployment of a self-managing atomic service is achieved through cloud orchestration as described in Karagiannis et al. (2014). All deployed software components (be it VMs or containers) know their role in the type graph (e.g., if they are an LB, AS, DB, or CA in our example). Each component is assigned a universally unique identifier (UUID). All components can access the Etcd cluster and discover each other.

The Etcd directory structure can be used both to represent the service type graph as well as the instance graph of the deployed components and their interconnections as in Figure 4. When a new component is deployed and started, it 1) joins the Etcd cluster and 2) advertises its availability by registering a new directory under its component type and saving relevant connection information there. For instance, in our example in Figure 4, a new CA instance adds a new directory with its UUID (uuid1) and saves a key with its endpoint to be used by the application server components. Edges in the instance graph are used to keep track of component connections in order to enforce the cardinalities on connections as specified in the type graph. The auto-scaling manager (described in the following subsections) is responsible for deciding how many components per type are needed, while the health manager will make sure that exactly as many instances as indicated by the auto-scaling logic are running and that their interconnections match the type graph. Component information (e.g., endpoints) is published by each component in Etcd periodically with a period of 5 seconds and a time

```
/TG
   /nodes
         /LB/{...}
         /AS/image=xxx
            /card
                 /min=1
                 /max=20
                 /req=4
         /CA/{...}
         /DB/{...}
   /edges
        /AS2CA
              /src=AS
              /dest=CA
              /src_card
                       /min=1
                       /max=10
              /dest_card
                        /min=1
                        /max=5
```

```
/IG
   /nodes
         /LB/{...}
         /AS/{...}
         /CA
            /uuid1
                    /endpoint=xxx
         /DB/{...}
   /edges
        /AS2CA
              /src=uuid1
              /dest=uuid7
```

Figure 4: An example snippet of the representation of a type graph (left) and instance graph (right) using Etcd directory structure

to live (TTL) of 10 seconds. Whenever a component fails or is removed, its access information is automatically removed from Etcd, and the health manager and all dependent components can be notified of the change.

Once the orchestrator has deployed the initial set of required components for the service, it sets the status of the service on Etcd to "active". Once this happens, the component which was elected leader of the Etcd cluster will start the self-managing functionality with the auto-scaling and health management logic.

3.2.1. Monitoring

Before discussing the auto-scaling functionality, we will describe how Etcd can also be used to store a partial and aggregated subset of monitoring information in order to allow auto-scaling decisions to be taken. The rationale behind storing monitoring information in Etcd is to allow resilience of the auto-scaling logic by making it stateless. Even if the VM or container where the auto-scaling logic has been running fails, a new component can be started to take over the auto-scaling logic and knowledge base from where it was left.

The common practice in cloud monitoring is to gather both low-level metrics from the virtual systems such as CPU, I/O, RAM usage as well as higher-level and application-specific metrics such as response times and throughputs [Aceto et al. (2013)]. Considering the latter metrics, full response time dis-

tributions are typically relevant in system performance evaluation, but for the sake of QoS management high percentiles (e.g, 95th, 99th) over time windows of few seconds are in general adequate to assess the system behavior. We assume that each relevant component runs internal monitoring logic that performs metrics aggregation and publishes aggregated metrics to Etcd. The actual directory structure and format in which to save key performance indicators (KPIs) is dependent on the auto-scaling logic to be used and is beyond the scope of this work. For instance, in our example the load balancer can use its own internal metrics in combination with the logstash aggregator[17] to provide the average request rate, response time, and queue length in the last 5, 10, 30 seconds and 1, 5, 10 minutes. These metrics are typically enough for an auto-scaling logic to take decisions on the number of needed application servers.

3.2.2. Auto-scaling

The auto-scaling component uses a performance model to control horizontal scalability of the components. The main function is to decide how many instances of each component need to be running to grant the desired QoS. Auto-scaling is started by the leader node. Its logic collects the monitoring information from Etcd, the current system configuration, and outputs the number of required components for each component type. This information is stored in the type graph for each node under the cardinality folder with the key "req" (required) as in Figure 4.

3.2.3. Health management

The node that is assigned health management functionalities compares the instance graph with the desired state of the system (as specified by the auto-scaling logic) and takes care of 1) terminating and restarting unresponsive components, 2) instantiating new components, 3) destroying no longer needed components, 4) configuring the connections among components in the instance graph so that cardinalities are enforced.

3.2.4. Full Life-cycle

Figure 5 depicts a simplified sequence diagram putting all the pieces together. The orchestrator sets up the initial deployment of the service components. They register to Etcd and watch relevant Etcd directories to perform

[17]http://logstash.net/

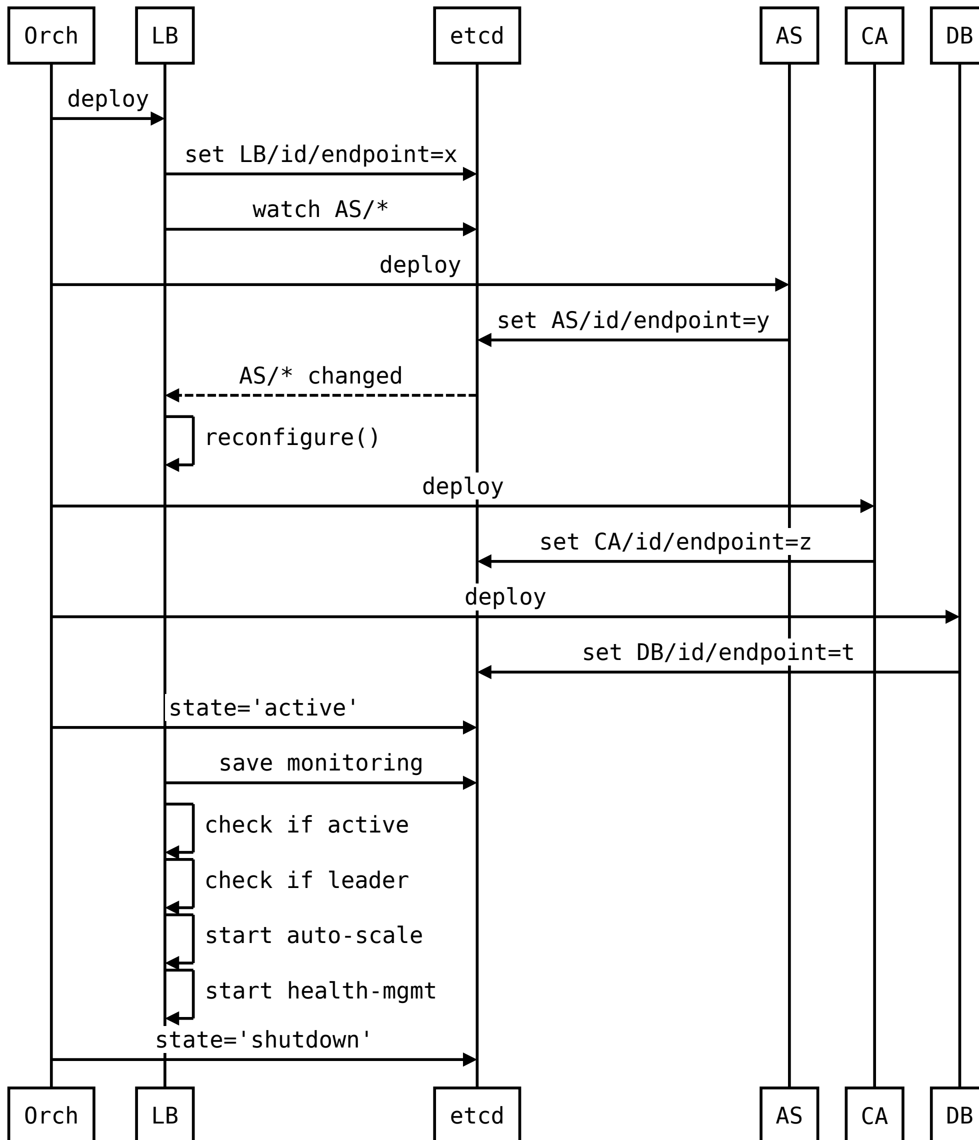

Figure 5: Sequence diagram depicting a simplified service instantiation and deinstantiation. For simplicity we represent Etcd as a single process.

configuration updates (reconfiguration parts for AS and CA components are omitted). Once all initial components are deployed, the orchestrator sets the service state to 'active'. Components generating monitoring information

save it periodically in Etcd.

Each component runs a periodic check on the service state. If the service is active and a node detects to be the Etcd cluster leader, it starts the auto-scale and health management processes. Alternatively, auto-scale and health management components can be started on other nodes depending on their utilization. A watch mechanism can be implemented from the cluster leader to indicate to a component that it should start a management functionality.

3.2.5. Self-healing properties

By placing components across different failure domains (e.g., availability zones in the same data center, or different data centers), the architecture described above is resilient to failure and is able to guarantee that failed components will be restarted within seconds. The fact that any remaining node can be elected leader, and that the desired application state and monitoring data is shared across an Etcd cluster, makes the health management and auto-scaling components stateless, and allows the atomic service to be correctly managed as long as the cluster is composed of the minimum required number of nodes for consensus which is three.

4. Experience

In the context of the Cloud-Native Applications (CNA) research initiative at the Service Prototyping Lab at Zurich University of Applied Sciences[18], we designed and evaluated various different forms of CNA applications. Amongst the practical results are CNA guidelines with the most common problems and pitfalls of application development specifically for the cloud. Here, we report on our experiences with a specific focus on applying the self-managing principles exposed in the previous sections to an existing application with one specific CNA support stack.

4.1. Implementation

Step 1: Use case identification. The goal of our experiments was on one hand to gather hands-on experience with the latest technologies supporting the design patterns for cloud-based applications, and on the other hand to successfully apply these patterns to a traditional business application which was not designed to run in the cloud. Rather than starting from scratch

[18]Service Prototyping Lab: http://blog.zhaw.ch/icclab/

with an application designed from inception for the cloud, we wanted to show that decomposition in smaller components (even by component functionality rather than application feature) often allows to achieve resilience and elasticity even in legacy applications.

For the evaluation of a suitable application, we decided to uphold the following criteria. The application should be:

- available as open source, to guarantee the reproducibility of our experiments
- a "business" application, to promote adoption of the CNA methods also for legacy applications
- a commonly used type of application, to achieve representative results

We took some time evaluating several well known and positively reviewed open source business applications and came up a list of about ten applications such as Customer Relationship Management (CRM), Enterprise Resource Planning (ERP), Document Management Systems (DMS). At the very end of our evaluation we were left with two choices: SuiteCRM[19] and Zurmo[20].

In the end we decided to go with Zurmo. The reasons behind this choice were that Zurmo:

- Is developed by a team with extensive experience with CRMs (formerly offering a modified version of SugarCRM @ Intelestream)
- Follows test-driven development (TDD) practices in its development
- Has all core functionality of a CRM without offering an overwhelming amount of features
- Has a modern look and feel to it

The first two reasons have given us confidence that the code is of high quality and that our changes will not just break the system in an unforeseen or hidden way. While evaluating CRMs, we repeatedly encountered statements saying one of the main problems of CRMs is that people are not

[19]https://suitecrm.com/
[20]http://zurmo.org/

using them. The last two reasons for choosing Zurmo address exactly this issue. After evaluating alternative products, we think Zurmo could be a CRM solution which would actually be used by its end-users.

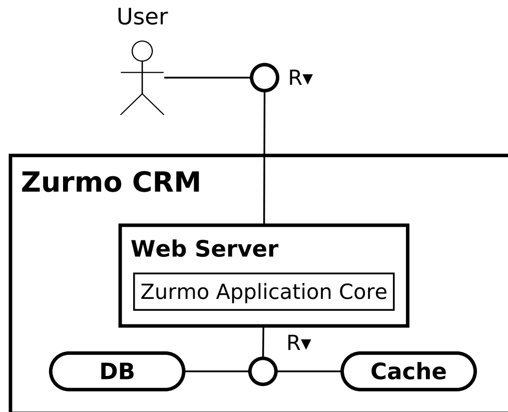

Figure 6: Zurmo initial architecture

Zurmo CRM (see Figure 6) is a PHP application employing the MVC pattern (plus a front-controller which is responsible for handling/processing the incoming HTTP requests) based on the Yii web framework. Apache is the recommended web server, MySQL is used as the backend datastore and Memcached for caching. It is pretty much a typical monolithic 3-tier application with an additional caching layer. The recommended way of running Zurmo is via Apache's PHP module. Thus, the logic to handle the HTTP requests and the actual application logic are somewhat tightly coupled.

Step 2: Platform. The subsequent step consisted of choosing a platform upon which to run the application. We wanted to address both private and public cloud scenarios. Given the availability of an OpenStack deployment at our lab, we chose to use both our internal private cloud and Amazon Web Services (AWS).

We used *CoreOS*[21] as basic VM image and *Fleet*[22] as a basic container / health-management component. Fleet is a distributed systemd (boot manager) cluster combined with a distributed *Etcd* (key-value store). After using

[21] https://coreos.com/
[22] https://coreos.com/fleet

it for some time, we can definitely confirm what CoreOS states about Fleet in their documentation: it is very low-level and other tools (e.g., Kubernetes[23]) are more appropriate for managing application-level containers. Our experiences with the CoreOS + Fleet stack were not always positive and we encountered some known bugs that made the system more unstable than we expected (e.g., failing to correctly pull containers concurrently from Docker hub[24]). Also, it is sometimes pretty hard to find out why a container is not scheduled for execution in Fleet. A more verbose output of commands and logging would be much more helpful to developers approaching Fleet for the first time.

Step 3: Architectural changes. We need to make every part of the application scalable and resilient. The first thing we did was to split the application using different Docker[25] containers to run the basic components (e.g., Apache Web server, Memcached, Mysql RDBMS).

We decided to first scale out the web server. Since the application core is in its original configuration tightly coupled to the web server, every Apache process comes with an embedded PHP interpreter. When we scale the webserver, we automatically also scale the application core. To achieve this, all we need is a load balancer which forwards incoming HTTP requests to the web servers. In the original implementation, Zurmo saved session-based information locally in the web server.

We modified the session handling so that it saves the session state in the cache as well as in the database. We can now access it in a quick manner from the cache, or should we encounter a cache miss we could still recover it from the database. After this change the architecture looks exactly the same but the overall application is scalable and resilient. After this modification there is no more need to use sticky sessions in Web servers. In other terms we made the web server tier stateless so that users can be evenly distributed among the existing web servers and, if one of the web servers or the caching system should crash, users won't be impacted by it.

Memcached already allows horizontally scaling its service by adding additional servers to a cluster. We then replaced the single server MySQL setup with a MySQL Galera Percona cluster to CNA-ify more parts of the

[23] http://kubernetes.io
[24] https://hub.docker.com
[25] https://www.docker.com

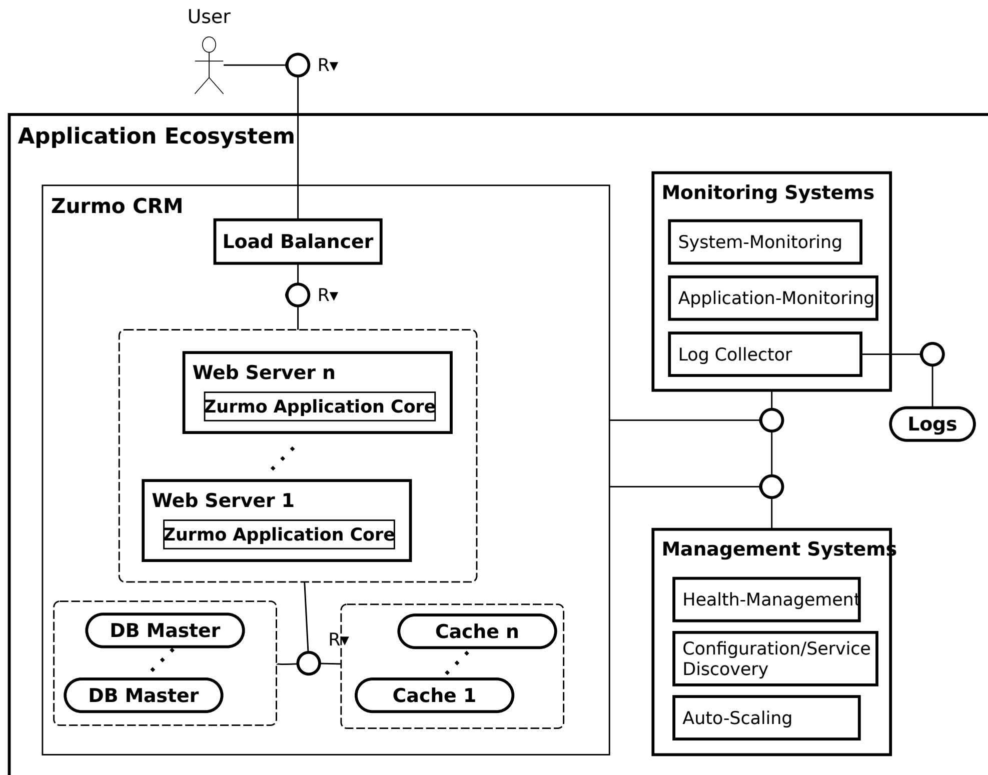

Figure 7: Zurmo CNA Architecture

application.

Step 4: Monitoring. We implemented a generic monitoring system that can be easily reused for any CNA application. It consists of the so-called *ELK stack*[26], *log-courier* and *collectd*. The ELK stack in turn consists of *Elasticsearch*, *Logstash* and *Kibana*. Logstash collects log lines, transforms them into a unified format and sends them to a pre-defined output. Collectd collects system metrics and stores them in a file. We use Log-Courier to send the application and system metric log-files from the container in which a service runs to Logstash. The output lines of Logstash are transmitted to Elasticsearch which is a full-text search server. Kibana is a dashboard and

[26]https://www.elastic.co/webinars/introduction-elk-stack

visualization web application which gets its input data from Elasticsearch. It is able to display the gathered metrics in a meaningful way for human administrators. To provide the generated metrics for the scaling engine, we developed a new output adapter for Logstash which enables to send the processed data directly to Etcd. The overall implementation is depicted in Fig. 8. The monitoring component is essential to our experimental evaluation.

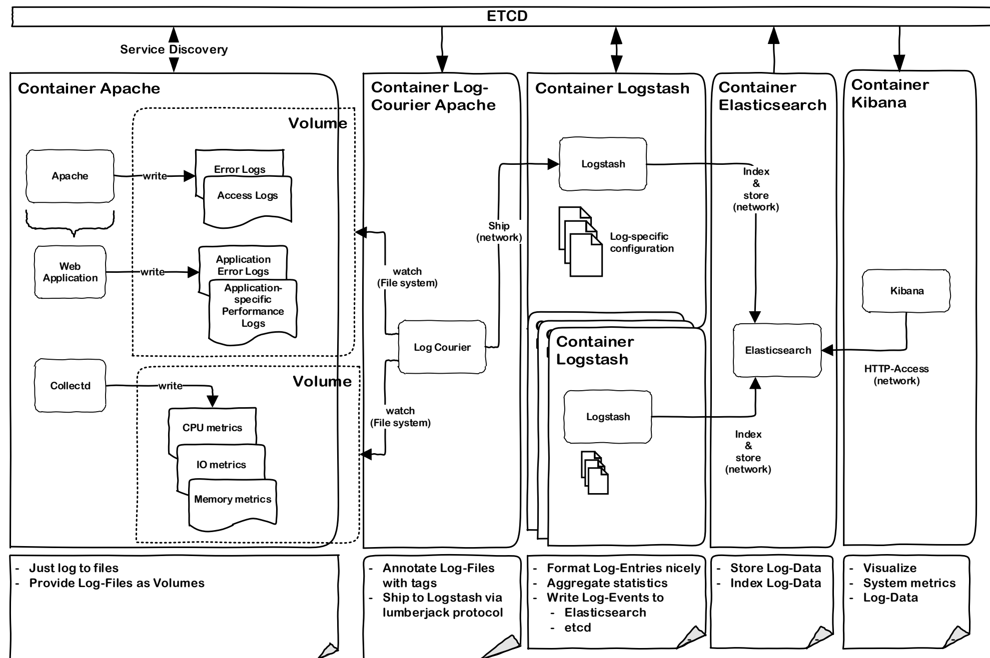

Figure 8: Monitoring and Logging

Step 5: Autoscaling. We also implemented our own scaling engine for container-based applications: *Dynamite*. Dynamite is an open-source Python application leveraging Etcd and Fleet. It takes care of automatic horizontal scaling, but also of the initial deployment of an application orchestrating the instantiation of a set of components (Fleet units) specified in a YAML configuration file. This configuration strategy allows to use Dynamite to recursively instantiate service compositions by having a top level YAML configuration specifying a list of Dynamite instances each with its own configuration file. Deploying the top Dynamite instance enables the "orchestration of a set of

orchestrators" each responsible for the independent scaling and management of a microservice. Dynamite uses system metrics and application-related information to decide whether a group of containers should be scaled out or in. If a service should be scaled out, Dynamite creates a new component instance (i.e., a "unit" in Fleet parlance) and submits it to Fleet. Otherwise, if a scale-in is requested, it instructs Fleet to destroy a specific unit.

Dynamite is itself *designed according to CNA principles*. If it crashes, it is restarted and re-initialized using the information stored in Etcd. This way, Dynamite can be run in a CoreOS cluster resiliently. Even if the entire node Dynamite is running on were to crash, Fleet would re-schedule the service to another machine and start Dynamite there where it could still restore the state from Etcd. For more details, we refer the reader to the documentation of the Dynamite implementation[27] as well as our work previously published in [Brunner et al. (2015)].

5. Experimental results

In this section we report on our resilience and scalability experiments with our cloud-native Zurmo implementation. We provide the complete source code of the application on our github repository[28].

All the experiment we discuss here have been executed on Amazon AWS (eu-central) using 12 t2.medium-sized virtual machines. We also ran the same experiments on our local OpenStack installation. The OpenStack results are in line with AWS and we avoid reporting them here because they don't provide any additional insight. Instead, in the spirit of enabling verification and repeatability, we decided to focus on the AWS experiments. They can be easily repeated and independently verified by deploying the CloudFormation template we provide in the "aws" directory of the implementation.

The experiments are aimed at demonstrating that the proposed self-managing architecture and our prototypical implementation correctly address the requirements we identified for cloud-native applications: elasticity and resilience. In other terms we pose ourselves the following questions:

- Does the application scale (out and in) according to load variations?
- Is the application resilient to failures?

[27] Dynamite scaling engine: https://github.com/icclab/dynamite/blob/master/readme.md
[28] https://github.com/icclab/cna-seed-project

In order to demonstrate resilience we emulate IaaS failures by respectively killing containers and VMs. Scaling of the application is induced by a load generator whose intensity varies over time.

The load generation tool we used is called Tsung[29]. We created a Zurmo navigation scenario by capturing it through a browser extension, then generalized and randomized it. You can also find this in our repository, in the "zurmo_tsung" component. In our experiments the load was generated from our laptops running Tsung locally. We simulated a gradually increasing number of users (from 10 up to 100) with a random think time between requests of 5 seconds in average. This yields a request rate of 0.2 requests per second per user, and a theoretical maximum expected rate of 20 requests per second with 100 concurrent users. The load is mainly composed of read (HTTP GET) operations, around 200, and roughly 30 write (HTTP POST) requests involving database writes. It is important to notice that, due to our choice of avoiding to use sticky HTTP sessions, also any request saving data in the HTTP session object results in database writes.

5.1. Scaling

In order to address the first question we configured Dynamite to scale out the service creating a new Apache container instance every time the 95th percentile of the application response time (RT) continuously exceeds 1000 milliseconds in a 15 seconds window.

The scale in logic instead will shut down any Apache container whose CPU utilization has been lower than 10% for a period of at least 30 seconds. Given that we are managing containers, scaling in and out is a very quick operation, and we can afford to react upon short term signals (e.g., RT over few seconds).

Since we used Fleet and CoreOS for the experiments, and not directly an IaaS solution billing per container usage, we also needed to manage our own virtual machines. We used 10 VMs that are pre-started before initiating the load and that are not part of the scaling exercise. The assumption is that future container-native applications will be only billed per container usage in seconds, and developers will only scale applications through containers. The actual distribution of containers upon virtual machines is decided by the Fleet scheduler, and in general results in uniform distribution across VMs.

[29]Tsung tool: http://tsung.erlang-projects.org

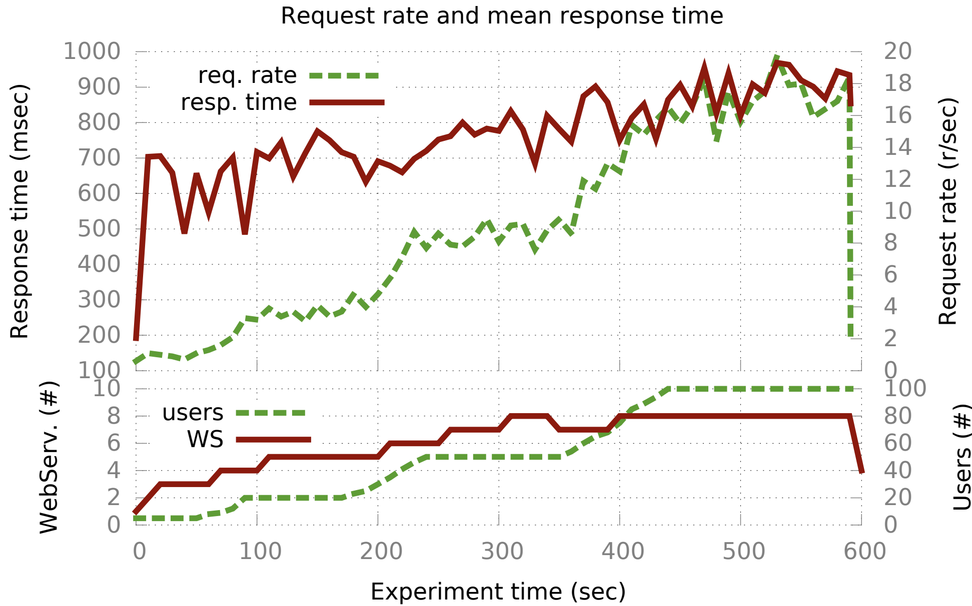

Figure 9: Response time, request rate, number of users and apache servers running the system without externally induced failures

Using our own internal monitoring system allows the application to scale on high level performance metrics (e.g. 95th percentiles) that are computed at timely intervals by the logstash component and saved to Etcd to be accessed by Dynamite.

Figure 9 shows one example run using the scaling engine to withstand a load of 10 concurrent users growing to 100. In the upper graph we plot the application response time in milliseconds (red continuous line, left axis) and the request rate in requests per second (green dashed line, right axis). The request rate grows from roughly 2 requests per second up to 20, while the response time is kept at bay by adaptively increasing the number of running Apache containers. The bottom part of the graph shows the number of running Apache containers at any point in time (red continuous line) as well as the number of simulated users. As soon as the generated traffic ends, the number of Apache containers is reduced.

This simple experiment shows the feasibility of an auto-scaling mechanism according to our self-managing cloud-native applications principles. For this example we only implemented a simple rule-based solution and we make no claims concerning its optimality with respect to minimizing operational

costs. More advanced adaptive model-based solutions (for instance the one in Gambi et al. (2015)) could be easily integrated using the same framework.

5.2. Resilience to container failures

In order to emulate container failures, we extended the Multi-Cloud Simulation and Emulation Tool (MC-EMU)[30]. MC-EMU is an extensible open-source tool for the dynamic selection of multiple resource services according to their availability, price and capacity. We have extended MC-EMU with an additional unavailability model and hooks for enforcing container service unavailability.

The container service hook connects to a Docker interface per VM to retrieve available container images and running instances. Following the model's determination of unavailability, the respective containers are forcibly stopped remotely. It is the task of the CNA framework to ensure that in such cases, the desired number of instances per image is only shortly underbid and that replacement instances are launched quickly. Therefore, the overall application's availability should be close to 100% even if the container instances are emulated with 90% estimated availability.

Figure 10 depicts the results of an example run in which we forced containers to fail with a 10% probability every minute. With respect to the previous example one can clearly notice the oscillating number of Apache Web servers in the bottom of the figure, and the effect this has on the application response time. Figures 11 and 12 show a glimpse of the monitoring metrics we were able to track and visualize through Kibana while running the experiment. We plot the average and percentile response times, response time per Apache container, request rate, HTTP response codes, number of running Apache containers and the CPU, memory, and disk utilization for each.

5.3. Resilience to VM failures

We also emulated VM failures, although without the automation models of MC-EMU or similar tools like ChaosMonkey[31]. Instead, we simply used the AWS console to manually kill one or more VMs at a given point in time to pinpoint critical moments.

[30]MC-EMU tool: https://github.com/serviceprototypinglab/mcemu
[31]https://github.com/Netflix/SimianArmy

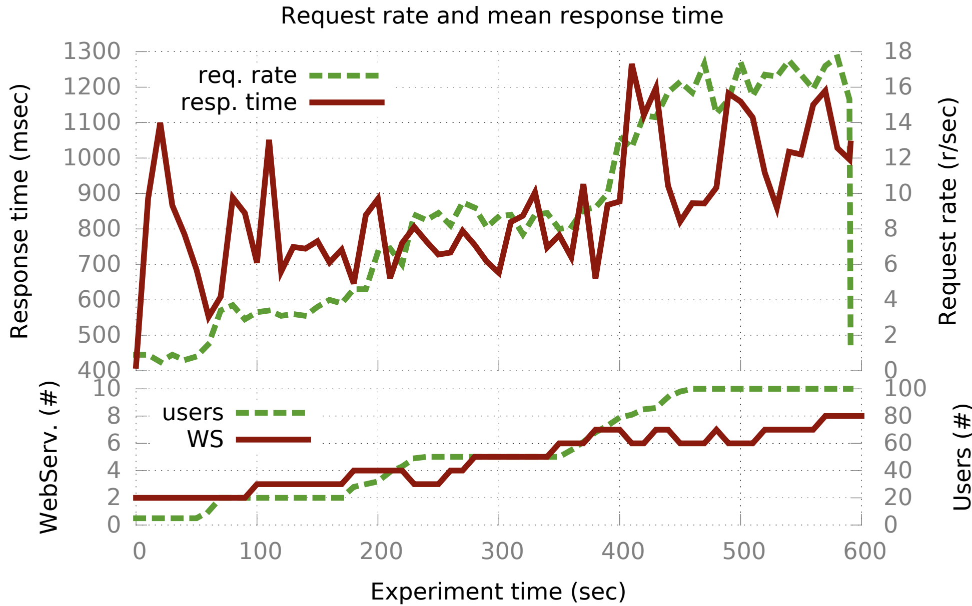

Figure 10: Response time, request rate, number of users and apache servers running the system inducing probabilistic container failures

The effects of killing entire VMs in our prototype implementation vary a lot depending on the role of the VM in the Etcd cluster as well as the type of containers it is running. As one could expect, killing VMs only hosting "stateless" (or almost stateless) containers (e.g., Apache, Memcached) only has small and transitory effects on the application quality of service. However, terminating a VM running stateful components (e.g., the database) has much more noticeable effects.

There are 2 types of VMs which we explicitly did not target for termination:

- the VM running logstash;
- the VMs acting as "members" of the Etcd cluster

The reason for the former exclusion is trivial and easily amendable: we simply did not have time to implement logstash as a load-balanced service with multiple containers. Killing the logstash container results in a period of few seconds without visible metrics in Kibana which would have defeated the goals of our experiment. The solution to this shortcoming is straightforward engineering.

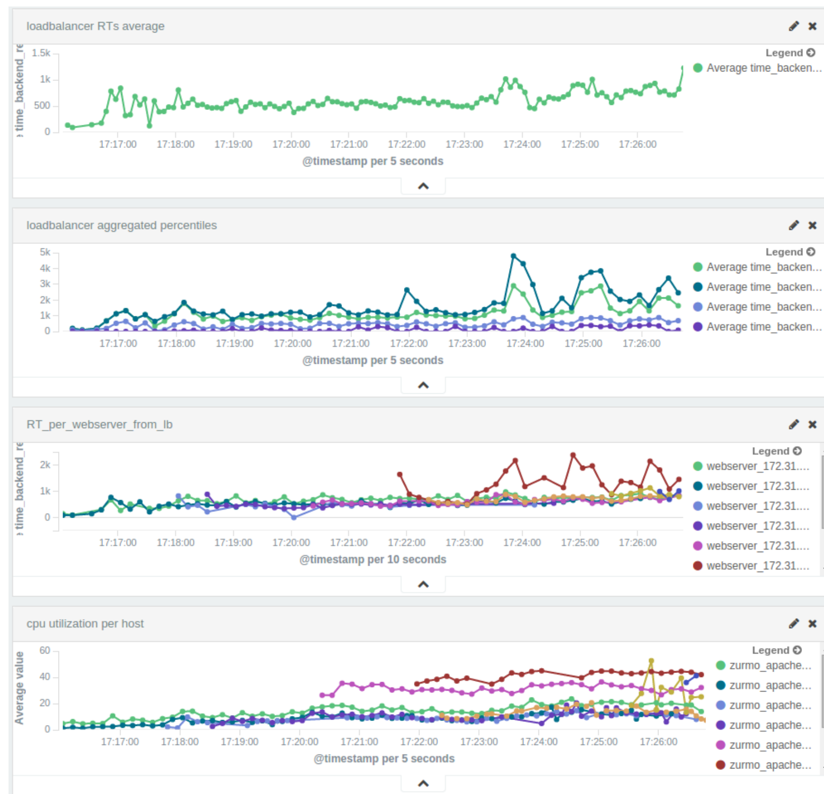

Figure 11: The real-time monitoring metrics while running the experiment depicted in Figure 10

Concerning the Etcd cluster member VMs, the issue is that the discovery token mechanism used for Etcd cluster initialization works only for cluster bootstrap. In order to keep the consensus quorum small, the default cluster is only composed of three members while other nodes join as "proxies" (they just read cluster state). Any VM termination of one of the 3 member nodes in AWS would restart the VM that would try to use Etcd discovery again to rejoin the cluster, but this would fail. In other failure scenarios, the machine might even change its IP address, requiring manual deletion and addition of the new endpoint. This problem is fairly common for Etcd in AWS, so much that we found an implementation of a containerized solution for it[32]. However, we did not yet integrate it into our stack and will leave a comparison

[32] http://engineering.monsanto.com/2015/06/12/etcd-clustering/

Figure 12: The real-time monitoring metrics while running the experiment depicted in Figure 10

to future work.

In order to show in practice how different the effects of killing VMs can be, we report here a run in which we target VMs running different types of components. Figure 13 depicts a run in which we killed 2 of the VMs running half of the 4 MySQL Galera cluster nodes roughly 3 minutes into the run (manually induced failures of two VMs each time are marked with blue vertical dashed lines). Together with the database containers, one can see that also some Apache containers were terminated. Moreover, having only two Galera nodes left, one of which was acting as a replication source for the Galera nodes newly (re)spawned by Fleet, means that the database response time became really high for a period, with a clearly visible effect on the Web application response time. Other two VMs at a time were killed respectively 6 and 9 minutes into the run, but since no database components

were hit, apart from the graph of the number of Apache instances, no major effects are perceived in the application response time.

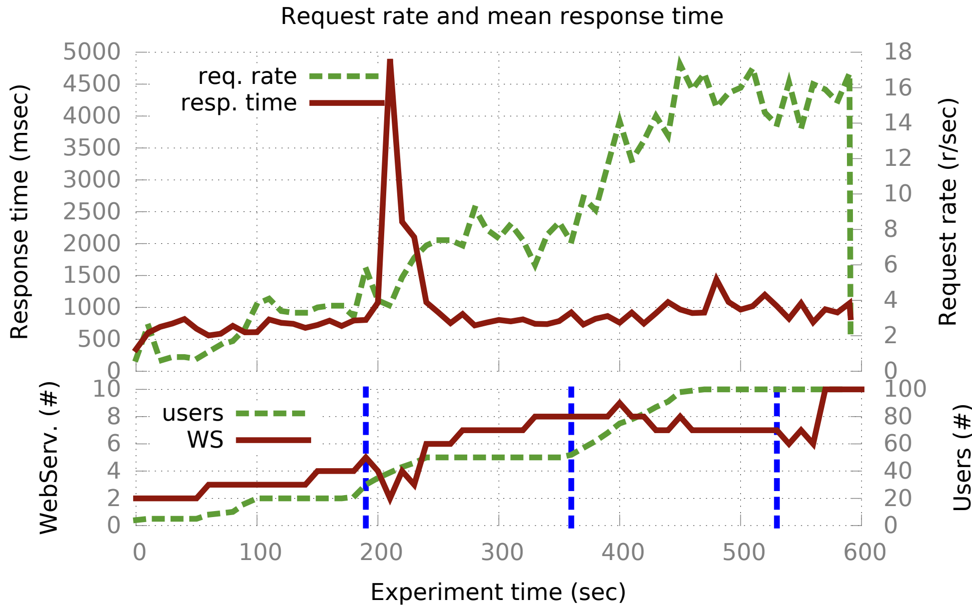

Figure 13: Response time, request rate, number of users and apache servers running the system inducing VM failures for stateful components

5.4. Lessons learnt

Implementing our self-manging cloud-native application design and applying it to an existing legacy Web application have proven to be valuable exercises in assessing the feasibility of our approach through a proof of concept implementation and identifying its weaknesses.

As it is mostly the case when realizing a theoretical design in practice, we were faced with several issues that hindered our progress. Some of them were a consequence of adopting fairly new technologies lacking mature and battle-tested implementations. Here we report in a bottom-up fashion the main problems we encountered with the technological stack we used for our implementation.

CoreOS. During about one year of research on CNA we used different releases of CoreOS stable. The peskiest issue we had with it took us quite some time to figure out. The symptoms were that the same containers deployed on the

same OS would randomly refuse to start. This caused Fleet/systemd to give up trying to bring up units after too many failed restarts. Fleet was failing in bringing up replicas of components we needed to be redundant, which made it extremely hard for to hope in achieving a reliable system in those conditions. These failures in starting containers happened sporadically and we could not reproduce them at will. This is not the behavior one is expecting with containers: one of the key points of them is to offer consistency between development and production environments.

It took us a while to understand that the random failures were due to a bug[33] in the Docker version included in CoreOS 766.3.0. In very few cases concurrently pulling multiple containers resulted in some container layers to be only partially downloaded, but docker would consider them complete and would refuse to pull again. The problem was aggravated by the fact we used unit dependencies in Fleet, requiring some units to start together on the same machine. In this case a failing container would cause multiple units to be disabled by Fleet.

It is hence always worth repeating: tight coupling is bad, especially if it implies cascading failures while building a resilient system.

Etcd. The biggest issue we had with Etcd was already mentioned in the previous section. We use Etcd as the foundation of the whole distributed solution both as a distributed key value store and for leader election. We expected that after a machine failure (when the machine gets rebooted or another machine takes its place) rejoining a cluster would be automatic, however this is not the case in AWS. Artificially causing machine failures like we did to test the reliability of our implementation often caused the Etcd cluster to become unhealthy and unresponsive.

Another issue we experienced is that Etcd stops responding to requests (also read requests!) if the machine disk is full. In this case the cluster might again fail, and Fleet would stop behaving correctly across all VM instances.

Fleet. Fleet is admittedly not a tool to be used directly for container management. Our lesson learnt here is that using managed approaches like Kubernetes should be preferred. Apart from this, we had often issues with failed units not being correctly removed in systemd on some nodes and in general misalignment between the systemd state of some hosts and the units Fleet

[33] https://github.com/coreos/bugs/issues/471

was aware of. Some command line interface mistakes which can easily happen (e.g., trying to run a unit template without giving it an identifier) result in units failing to be removed in systemd and Fleet hanging on the loop requesting their removal preventing any other command from being executed.

Another unexpected behavior we managed to trigger while developing is due to the interplay of Fleet and Docker. Fleet is expected to restart automatically failed containers, however Docker volumes are not removed by default (the rationale is that they might be remounted by some other containers). The net effect is that after a while machines with problematic containers run out of disk space, Etcd would stop working, the cluster would become unhealthy, and the whole application would be on its own running without Fleet. The CoreOS bug we mentioned above also caused this on long running applications effectively bringing down the service.

These are all minor issues due to the fact that most of the tools we use are in development themselves. However, any of these problems might become a blocker for developers using these tools for the first time.

Self-managing Zurmo. Finally some considerations concerning our own design. The first thing to discuss is that we did not go all the way and implement Zurmo as a set of self-managing microservices each with its own specific application-level functionality.

The main reason is that we did not want to get into Zurmo's code base to split its functionality into different services. This would have meant investing a large amount of time to understand the code and the database (which has more than 150 tables). Instead, we preserved the monolithic structure of the application core written in PHP. What we did was replicating the components and put a load balancer in front of them (e.g., for Apache or MySQL Galera cluster). So, in a way, we created a microservice for each type of component, with a layer of load balancers in front. This is not the "functional" microservice decomposition advocated by Lewis and Fowler [Lewis and Fowler (2014)], however we showed experimentally that for all the purposes of resilience and elasticity it still works. Where it would not work is in fostering and simplifying development by multiple development teams (each catering for one or more microservices as a product) in parallel. This for us means that the microservices idea is actually more a way to scale the development process itself rather than the running application.

We used Etcd for component discovery, for instance for the Galera nodes to find each other, the loadbalancers to find backends, and Apache to find the

Galera cluster endpoint and Memcached instances. Breaking the application at least in microservices based on component types would in hindsight have been a cleaner option.

One of the negative effects of having automatic component reconfigurations upon changes of the component endpoints registered in Etcd is that circular dependencies would cause ripple effects propagating through most components. This for instance happened when we initially replaced a single MySQL instance with a set of MySQL Galera nodes that needed to self-discover. A much more elegant solution is to put one or more load balancers in front of every microservice and register them as the endpoint for a service. An even better solution is using the concept of services and an internal DNS to sort out service dependencies as done in Kubernetes. This solution does not even require reconfigurations upon failures.

A very positive aspects of our implementation is that we have a self-managing solution that now works seamlessly in OpenStack, AWS, and Vagrant. The internal monitoring stack can be easily reused for other applications, and the decomposition in docker containers allowed us to hit the ground running when starting our porting of the solution to Kubernetes which is our ongoing work.

Another aspect to notice is that when we started our implementation work, container managers were in their infancy and we had to build a solution based on IaaS (managing VMs and their clustering) rather than directly using APIs to manage sets of containers. Already now, the availability of container managers has improved, and we expect the commercial market to grow fast in this segment. If one is being charged per VM in a IaaS model, then only auto-scaling containers does not mean reducing costs. In practice what can be done is using, for example in AWS, AWS AutoscalingGroups for VMs and custom metrics generated from within the application to trigger the instantiation and removal of a VM. The work is conceptually simple, but we did not implement it yet and are not aware of existing re-usable solutions.

Although our own experience using Fleet to implement the proposed solution was somehow difficult, we can already relate on the excellent results we are having by porting the entire architecture to Kubernetes. It is still work in progress, but the whole work basically amounts to converting Fleet unit files into replicaiton controller and service descriptors for Kubernetes, no need for component discovery since "services" are framework primitives. All in all, the availability of more mature container management solutions will only simplify the adoption of microservices architectures.

6. Related work

To the best of our knowledge, the work in [Toffetti et al. (2015)] we extended here was the first attempt to bring management functionalities *within* cloud-based applications leveraging on orchestration and the consensus algorithm offered by distributed service configuration and discovery tools to achieve *stateless* and *resilient* behavior of management functionalities according to cloud-native design patterns. The idea builds on the results and can benefit from a number of research areas, namely cloud orchestration, distributed configuration management, health management, auto-scaling, and cloud development patterns.

We already discussed the main orchestration approaches in literature, as this work reuses much of the ideas from [Karagiannis et al. (2014)]. With respect to the practical orchestration aspects of microservices management, a very recent implementation[34] adopts a similar solution to what we proposed in our original position paper. We had some exchanges of views with the authors, but are not aware whether our work had zero or even minimal influence on the Autopilot[35] cloud-native design pattern recently promoted by Joyent. Either way, we consider the fact that other independent researchers came up with a very similar idea an encouraging sign for our work.

Several tools provide distributed configuration management and discovery (e.g., Etcd, ZooKeeper, Consul). From the research perspective, what is more relevant to this work is the possibility of counting on a reliable implementation of the consensus algorithm. Much of the health management functionality described in the paper is inspired from Kubernetes [Bernstein (2014)], although to the best of our knowledge Kubernetes was originally "not intended to span multiple availability zones"[36]. Ubernetes[37], is a project aiming to overcome this limit by federation.

A set of common principles concerning automated management of applications are making their way in container management and orchestration

[34] https://www.joyent.com/blog/app-centric-micro-orchestration [retrieved on 2016.06.10]

[35] http://autopilotpattern.io/

[36] https://github.com/GoogleCloudPlatform/kubernetes/blob/master/DESIGN.md retrieved 03/03/2015

[37] https://github.com/kubernetes/kubernetes/blob/master/docs/proposals/federation.md

approaches (e.g., Kubernetes, Mesos[38], Fleet, Docker-compose[39]) with the identification, conceptualization, and instantiation of management control loops as primitives of the underlying management API. To give a concrete example, "replication controllers" in Kubernetes are a good representative of this: "A replication controller ensures the existence of the desired number of pods for a given role (e.g., "front end"). The autoscaler, in turn, relies on this capability and simply adjusts the desired number of pods, without worrying about how those pods are created or deleted. The autoscaler implementation can focus on demand and usage predictions, and ignore the details of how to implement its decisions"[Burns et al. (2016)]. Our proposed approach leverages on basic management functionalities where present, but proposes a way to achieve them as a part of the application itself when deployed on a framework or infrastructure that does not support it. Moreover, we target not only the atomic service level managing components (akin to what Kubernetes does for containers) but also service composition level managing multiple microservice instances. In [Burns et al. (2016)], the authors also advocate control of multiple microservices through *choreography* rather than "centralized orchestration" to achieve *emergent behavior*. In our minds, once applications are deployed across different cloud vendors, orchestration (albeit with distributed state as we propose) is still the only way to achieve a coherent coordinated behavior of the distributed system.

Horizontal scaling and the more general problem of quality of service (QoS) of applications in the cloud have been addressed by a multitude of works. We reported extensively on the self-adaptive approaches in [Gambi et al. (2013)] and here give only a brief intuition of the most relevant ones. We can cite the contributions from Nguyen et al. [Nguyen et al. (2013)] and Gandhi et al. [Gandhi et al. (2012)] which use respectively a resource pressure model and a model of the non-linear relationship between server load and number of requests in the system together with the maximum load sustainable by a single server to allocate new VMs. A survey dealing in particular with the modelling techniques used to control QoS in cloud computing is available in [Ardagna et al. (2014)]. With respect to the whole area of auto-scaling and elasticity in cloud computing, including the works referenced from the surveys cited above, this work does not directly address

[38] http://mesos.apache.org
[39] https://docs.docker.com/compose

the problem of *how* to scale a cloud application to achieve a specific quality of service. Works in current and past elasticity / auto-scaling literature focus either on the models used or on the actual control logic applied to achieve some performance guarantees. In [Toffetti et al. (2015)] we propose an approach that deploys the management (e.g., auto-scaling) functionalities *within the managed application*. This not only falls in the category of self-* / autonomic systems applied to auto-scaling surveyed in [Gambi et al. (2013)] (the application becomes self-managing), but with respect to the state of the art, brings the additional (and cloud-specific) contribution of making the managing functionalities stateless and resilient according to cloud-native design principles. In this respect, the works listed above are related just in desired functionality, but not relevant to the actual contribution we claim as any of the scaling mechanisms proposed in literature can be used to perform the actual scaling decision.

Finally, considering cloud patterns and work on porting legacy applications to the cloud, the work of Ellison et al. (2014) is worth considering when addressing the thorny problem of re-engineering the database layer of existing applications to achieve scalable cloud deployment. With this respect, in our implementation work we just migrated a single MySQL node into a multi-master cluster whose scalability is still limited in the end.

7. Conclusion

In this experience report article, we have introduced an architecture that leverages on the concepts of cloud orchestration and distributed configuration management with consensus algorithms to enable self-management of cloud-based applications. More in detail, we build on the distributed storage and leader election functionalities that are commonly available tools in current cloud application development practice to devise a resilient and scalable managing mechanism that provides health management and auto-scaling functionality for atomic and composed services alike. The key design choice enabling resilience is for both functionalities to be stateless so that in case of failure they can be restarted on any node collecting shared state information through the configuration management system.

Concerning future work, we plan to extend the idea to incorporate the choice of geolocation and multiple cloud providers in the management functionality. Another aspect we look forward to tackle is that of continuous deployment management, including adaptive load routing.

Acknowledgements

This work has been partially funded by an internal seed project at ICCLab[40] and the MCN project under grant nr. [318109] of the EU 7th Framework Programme. It has also been supported by an AWS in Education Research Grant award, which helped us to run our experiments on a public cloud.

Finally we'd like to acknowledge the help and feedback from our colleagues Andy Edmonds, Florian Dudouet, Michael Erne, and Christof Marti in setting up the ideas and implementation. A big hand for Özgür Özsu who ran most of the experiments and collected all the data during his internship at the lab.

References

G. Aceto, A. Botta, W. De Donato, and A. Pescapè. Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115, 2013.

W. Adam. The twelve-factor app. `http://12factor.net/`, January 2012. Retrieved: 2016.06.10.

E. Anderson, X. Li, M. A. Shah, J. Tucek, and J. J. Wylie. What consistency does your key-value store actually provide? In *Sixth USENIX Workshop on Hot Topics in System Dependability (HotDep)*, OCtober 2010.

D. Ardagna, G. Casale, M. Ciavotta, J. F. Pérez, and W. Wang. Quality-of-service in cloud computing: modeling techniques and their applications. *Journal of Internet Services and Applications*, 5(1):1–17, 2014.

D. Bernstein. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, (3):81–84, 2014.

S. Brunner, M. Blöchlinger, G. Toffetti, J. Spillner, and T. M. Bohnert. Experimental Evaluation of the Cloud-Native Application Design. In *4th International Workshop on Clouds and (eScience) Application Management (CloudAM)*, December 2015.

[40] http://blog.zhaw.ch/icclab/

B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, Omega, and Kubernetes. *Communications of the ACM*, 59(5):50–57, 2016.

G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41 (6):205–220, Oct. 2007. ISSN 0163-5980. doi: 10.1145/1323293.1294281. URL `http://doi.acm.org/10.1145/1323293.1294281`.

H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 3540311874.

M. Ellison, R. Calinescu, and R. Paige. Re-engineering the database layer of legacy applications for scalable cloud deployment. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, UCC '14, pages 976–979, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-7881-6. doi: 10.1109/UCC.2014.160. URL `http://dx.doi.org/10.1109/UCC.2014.160`.

C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter. *Cloud Computing Patterns*. Springer, 2014.

A. Gambi, G. Toffetti, and M. Pezzè. Assurance of self-adaptive controllers for the cloud. In J. Cámara, R. de Lemos, C. Ghezzi, and A. Lopes, editors, *Assurances for Self-Adaptive Systems - Principles, Models, and Techniques*, volume 7740 of *Lecture Notes in Computer Science*, pages 311–339. Springer, 2013. ISBN 978-3-642-36248-4. doi: 10.1007/978-3-642-36249-1_12. URL `http://dx.doi.org/10.1007/978-3-642-36249-1_12`.

A. Gambi, M. Pezze, and G. Toffetti. Kriging-based self-adaptive cloud controllers. *Services Computing, IEEE Transactions on*, PP(99):1–1, 2015. ISSN 1939-1374. doi: 10.1109/TSC.2015.2389236.

A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems*, pages 1–33, 2012.

A. Homer, J. Sharp, L. Brader, N. Masashi, and T. Swanson. *Cloud Design Patterns - Prescriptive Architecture Guidance for Cloud Applications*. Microsoft, 2014.

G. Karagiannis, A. Jamakovic, A. Edmonds, C. Parada, T. Metsch, D. Pichon, M. Corici, S. Ruffino, A. Gomes, P. S. Crosta, et al. Mobile cloud networking: Virtualisation of cellular networks. In *Telecommunications (ICT), 2014 21st International Conference on*, pages 410–415. IEEE, 2014.

J. Lewis and M. Fowler. Microservices. `http://martinfowler.com/articles/microservices.html`, March 2014. Retrieved: 2016.06.10.

P. Mell and T. Grance. The NIST Definition of Cloud Computing, September 2011.

H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service. In *Proc. of the International Conference on Autonomic Computing and Communications*, pages 69–82, 2013. ISBN 978-1-931971-02-7.

M. T. Nygard. *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2007.

G. Toffetti, S. Brunner, M. Blöchlinger, F. Dudouet, and A. Edmonds. An architecture for self-managing microservices. In V. I. Munteanu and T. Fortis, editors, *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud, AIMC@EuroSys 2015, Bordeaux, France, April 21, 2015*, pages 19–24. ACM, 2015. ISBN 978-1-4503-3476-1. doi: 10.1145/2747470.2747474. URL `http://doi.acm.org/10.1145/2747470.2747474`.

A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. 2015.

S. Wardley. Private vs Enterprise Clouds. `http://blog.gardeviance.org/2011/02/private-vs-enterprise-clouds.html`, February 2011. Retrieved: 2015.03.16.

B. Wilder. *Cloud Architecture Patterns*. O'Reilly, 2012.