Experimental Evaluation of the Cloud-Native Application Design

Sandro Brunner, Martin Blöchlinger, Giovanni Toffetti, Josef Spillner, Thomas Michael Bohnert
Zurich University of Applied Sciences, School of Engineering
Service Prototyping Lab (blog.zhaw.ch/icclab/)
8401 Winterthur, Switzerland
Email: {brnr,bloe,toff,josef.spillner,thomas.bohnert}@zhaw.ch

***Abstract*—Cloud-Native Applications (CNA) are designed to run on top of cloud computing infrastructure services with inherent support for self-management, scalability and resilience across clustered units of application logic. Their systematic design is promising especially for recent hybrid virtual machine and container environments for which no dominant application development model exists. In this paper, we present a case study on a business application running as CNA and demonstrate the advantages of the design experimentally. We also present Dynamite, an application auto-scaler designed for containerised CNA. Our experiments on a Vagrant host, on a private OpenStack installation and on a public Amazon EC2 testbed show that CNA require little additional engineering.**

I. Background

Delivering software applications from the cloud requires a new way of thinking about software architectures and software engineering processes [1]. Typical cloud service environments with raw infrastructure and featureful platform services (IaaS and PaaS, respectively) offer multiple benefits over other software hosting and delivery paradigms. Their full exploitation is however only possible if the software is made – to a certain extent – aware of the cloud hosting characteristics, including elastic horizontal scalability, on-demand payment and orchestrated application lifecycle. Often, applications are migrated into the cloud as monolithic virtual machines, which makes them vulnerable to *availability failures* and *demand spikes*. This paper reports on how to mitigate these two risks.

There are a number of approaches to tackle this problem. Model-based software deployment in the cloud matches software capabilities with the hosting environment [2]. CodeCloud is a concrete multi-IaaS platform for executing scientific applications with specified infrastructure requirements, including SLAs, minimum replication factors and elasticity [3]. Dispersed Computing is an availability-increasing concept for storing and processing fragments of data across cloud providers, either in the clear or with encryption for higher confidentiality, resulting in Stealth Computing [4].

Cloud Native Applications (CNA) [5] are our approach of de-composing software functionality into smaller service units, connecting them with an orchestration authority and a scaling engine, and running them with high resilience and scalability. So far, CNA has been a mostly theoretic design with a distinct lack of an evaluation study. This paper therefore does not attempt to argue for CNA, for which we refer to our previous publication [5]. It is also not a step-by-step guide to CNA, for which we refer to our detailed posts on this topic [1]. Instead, it briefly repeats the CNA characteristics and then focuses on a thorough evaluation.

In the next sections, we will first recapitulate design principles for CNA. Then, we will introduce an evaluation scenario, and subsequently evaluate the scenario application's scalability and resilience in a private and in a public cloud environment. Finally, we will wrap up our findings and contributions which includes a novel scaling engine called Dynamite.

II. Cloud Native Applications

The essential properties of CNA are *scalability* and *resilience*. For the scalability, a CNA has to be capable of taking advantage of the cloud characteristics *on-demand self-service*, *rapid elasticity* and *measured service* and to adjust its capacity by adding or removing resources. For the resilience, a CNA has to tolerate failures of commodity hardware, virtualised resources or services. In order to achieve both properties, CNA are structured into core functionality and supporting functionality. The core is subdivided into fine-grained microservices so that each service can be scaled and governed individually depending on the load within the corresponding part of the application. The support encompasses monitoring and management systems [5].

On the implementation level, cloud containers are an appropriate mechanism to realise CNA. Containers, as opposed to virtual machines, can be spawned and terminated quickly similar to native system processes. Furthermore, in order to make use of existing infrastructure management tooling, the handling of containers can be harmonised with the handling of virtual machines by having groups of containers across nodes.

III. Evaluation Scenario

We present the CNA evaluation scenario in four steps. First, we select a suitable target application. Second, we analyse the application and plan the conversion to a CNA. Third, we conduct the cloud deployment to achieve a running CNA.

[1] http://blog.zhaw.ch/icclab/category/research-approach/themes/cloud-native-applications/

A. Selection of a Software Application

Our aim is to show and evaluate how to migrate an application as CNA into the cloud – and by doing so, making it resilient and scalable. This implies that the application is self-managing so that once the application has been deployed, it should automatically recover from failures and also automatically scale up or down should the need arise. For the evaluation, we had to settle on one particular existing software application to show the applicability of the research. One of the main requirements the application to migrate needed to have was to be open-source. This allows for changes to some of the core functionality of the application at some point. After filtering popular business domains on the cloud and a subsequent comparative evaluation involving five customer-relationship management (CRM) business applications, we selected Zurmo, an open-source CRM, as target to be migrated into the cloud. There are two reasons Zurmo is a suitable example. First, its code-base size is moderate which would allow us to make changes to the source-code without too much effort. Second, the application is being developed using a test-driven process which ensures that the code has a certain quality which would allow for a safer change of the application.

Next, we will describe how the original architecture of the application looked like and which changes we introduced to adapt it for a cloud environment. We explain how the resulting CNA looks like and how it is technically implemented.

B. Analysis and Planning

From an architectural point of view, Zurmo is very straightforward and representative for many SaaS offerings. It is a web application connected to a database with an optional cache. An Apache web server is used to run the PHP code, MySQL is the standard database used for the backend and Memcached is used for the caching system. Session information is saved locally on the web server by the application process. Fig. 1 shows Zurmo's original architecture. In contrast, Fig. 2 shows the architecture after the process of migrating it to a proper CNA architecture. The CNA support functions are prominently visible.

The application is scaled by placing a (possibly distributed) load balancer in front of the web servers. Memcached already allows horizontally scaling its service by adding additional servers to a cluster. The database runs in a master/slave setup. It can handle a bigger write load than the usage of a CRM will typically require, and therefore, we do not consider the single master a bottleneck in practice.

To enable the application's ability of being self-managing, a monitoring as well as a management system has been added. The monitoring system monitors the application and the systems the application runs on. This information is needed to know the status of the system and to provide input for the auto-scaling decision. All information gathered from those systems are collected and saved in an external database for analysis. The management system is responsible for the discovery, the reliability and the automatic scaling of the application. The discovery allows services to find other

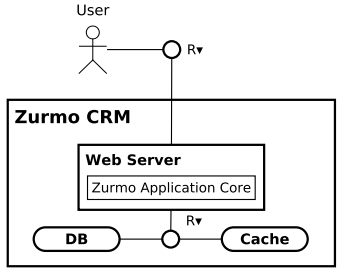

Fig. 1. Original monolithic architecture of Zurmo

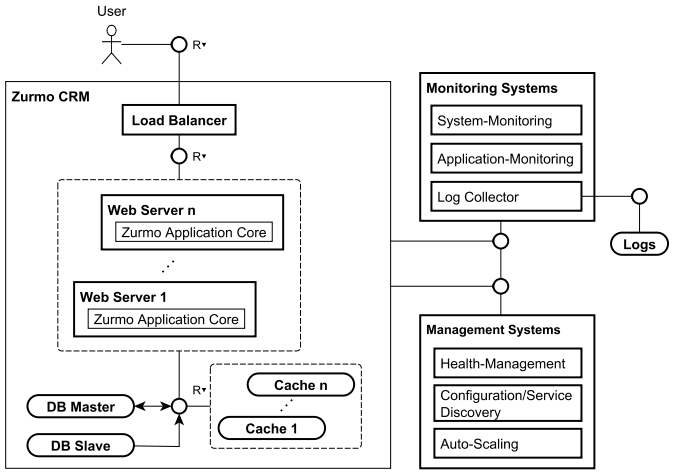

Fig. 2. Evolved architecture of Zurmo with CNA support functions

services. The health management component keeps track of the health of the various parts of the system and restarts failed components. The configuration/service discovery is used for the various parts of the application to register themselves and store information about the current status of the system. The auto-scaling system takes scaling decisions based on input it gets from the monitoring systems.

C. Conversion to CNA Architecture

Zurmo originally saved the session state from clients locally on the web server nodes. Scaling the application horizontally by running multiple web server instances behind a load balancer would work but present some significant drawbacks. Because of the local sessions, clients would always have to be forwarded to the web server they originally connected to and a crash of that particular web server would have resulted in all clients connected to it losing their session information. The former issue could have easily been resolved by using sticky sessions. The latter issue could only be resolved by changing the application to save session state remotely instead of locally. In the migrated version of the application, Zurmo saves the client session state to both the cache and the database, making the apache layer *stateless* and the application both scalable and more resilient. The failure of a single web server will not negatively influence the clients anymore.

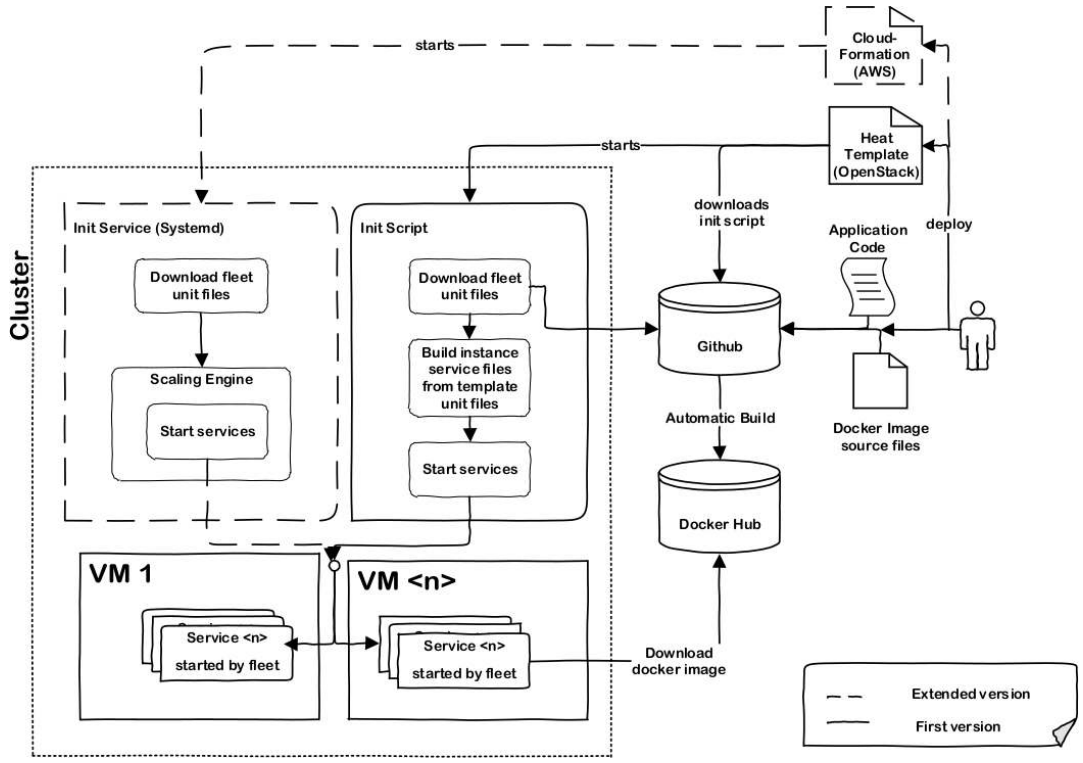

Fig. 3. Deployment scheme for containerised applications

D. Implementation and Migration Details

1) Virtual Machines and Containers: We chose the operating system image CoreOS as underlying virtual machine because it is designed for clustered deployments and comes with system services for health management (*Fleet*[2]) and service/configuration management (*etcd*). Its software processes typically run inside *Docker* containers. Different components of an application are put into dedicated containers which are then network-connected or directory-connected with each other. In a first step, we containerized the components of Zurmo. Thus, we created one container for the web server, one for the load balancer, one for the database and one for the caching system. All the CNA support components added later – monitoring systems, scaling-engine – were also containerized. Fleet is actually a distributed *init* system responsible for system boot sequences. It runs on top of *systemd*, a recent parallelized implementation and conceptual enhancement of *init*. Fleet allows to describe services and to deploy them on machines in a cluster, ensuring they are kept in a certain state, and re-deploys them in case of machine failure. In the migrated application, the services typically start containers. By containerizing the application and running it in a CoreOS cluster with Fleet, we can ensure a resilient system as shown in Fig. 3.

2) Service Autodiscovery: In order for the application to be truly self-managing, its components need to be aware of one another and be able to adapt to a changing environment, e.g. components being added or removed. In a self-managing system, the web server should announce itself and the load-balancer should reconfigure itself accordingly, for instance. In order to achieve this, we use Etcd, essentially a distributed key-value store, as service-discovery component in combination with *confd*. Confd watches certain keys in Etcd, gets notified when an update occurs, and updates configuration files based on the values of those keys (Fig. 4). To announce services, we use the *sidekick* pattern. With every service which needs to announce itself (e.g. web server), a complementary (lifecycle-bound sidekick) service whose only job it is to announce the former service in Etcd is deployed. Confd is only installed in containers of services which need to update themselves in case components are added or removed.

3) Monitoring: The monitoring system, re-usable across CNA applications, consists of the so-called *ELK stack*, *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 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

[2]For more details on Fleet the interested reader can refer to https://github.com/coreos/fleet

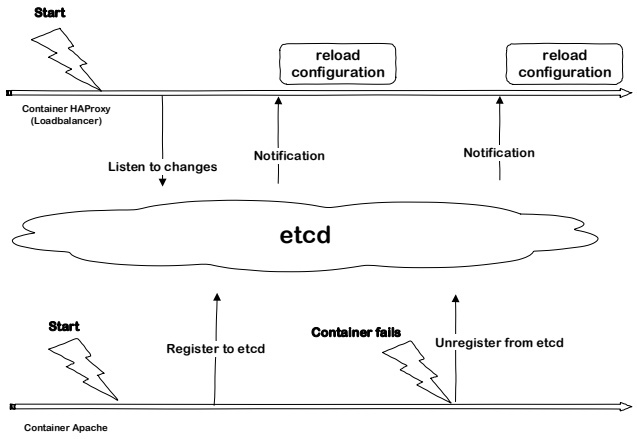

Fig. 4. Dynamic reconfiguration of services through the auto-discovery mechanism of etcd

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. 5. The monitoring component is essential to our experimental evaluation.

4) Auto-Scaling Engine: The new scaling-engine *Dynamite* is an open-source Python application which uses Etcd and Fleet and therefore integrates natively with CNA. Dynamite takes care of automatic horizontal scaling, but also of the initial deployment of the application. It consists of several components shown in Fig. 6. An auto-scaled application is assumed to be composed of Fleet services. Dynamite uses system metrics and application-related information to take decisions when a service should be scaled out or in. If a service should be scaled out, Dynamite creates a new service instance and submits it to Fleet. Otherwise, if a scale-in is requested, it instructs Fleet to destroy a specific service instance.

To calculate scaling decisions, Dynamite uses at the moment a rule-based approach, but it can be easily extended to support more advanced scaling logic (e.g., model based). A rule consists of the name of the metric to be observed and a threshold value, the type of the service the metric belongs to and a comparative operator which tells Fleet how to compare reported monitoring values with the threshold. Additionally, a period can be defined in which the value should always be over respectively under the threshold before executing a scaling action. This avoids single peaks generating useless actions. A cool-down period can also be configured per rule. Dynamite will wait for the defined amount of time after the scaling rule was executed before issuing the next scaling action from the same rule. The following example shows an excerpt of the configuration file as used in the Zurmo scenario application.

```
Service:
  apache:
    name_of_unit_file: apache@.service
    type: webserver
    min_instance: 2
    max_instance: 5
```

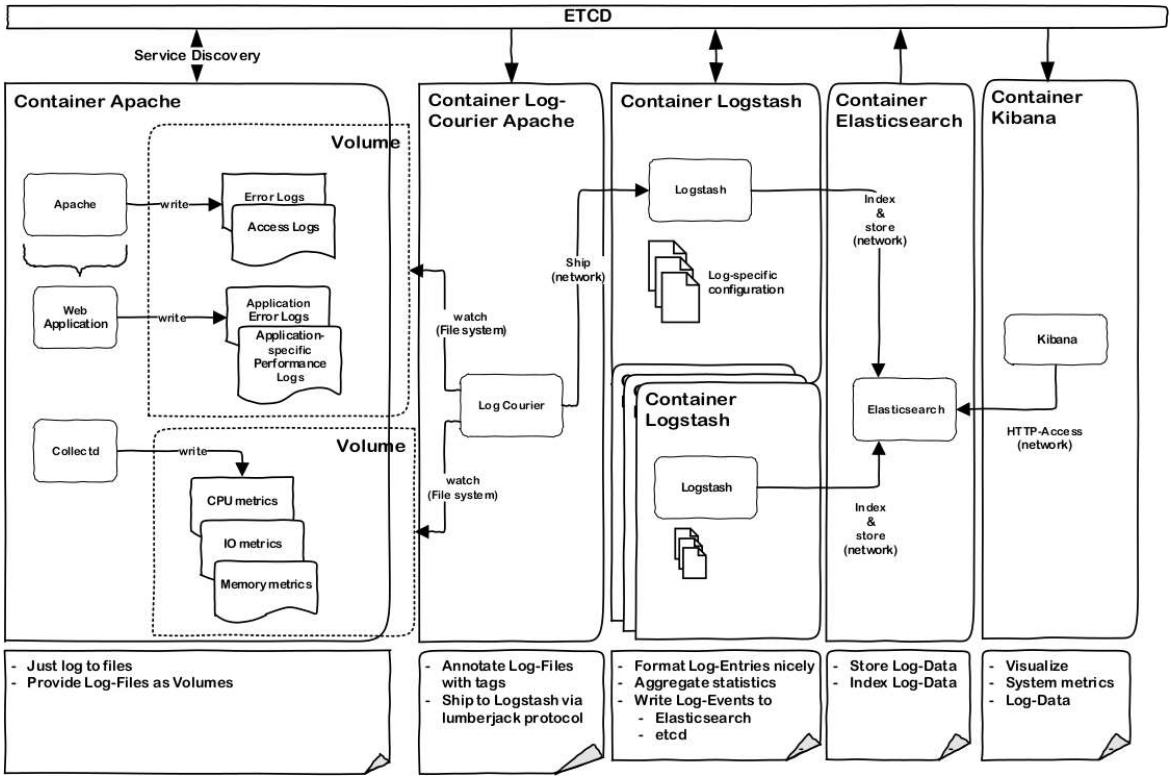

Fig. 5. Monitoring and Logging

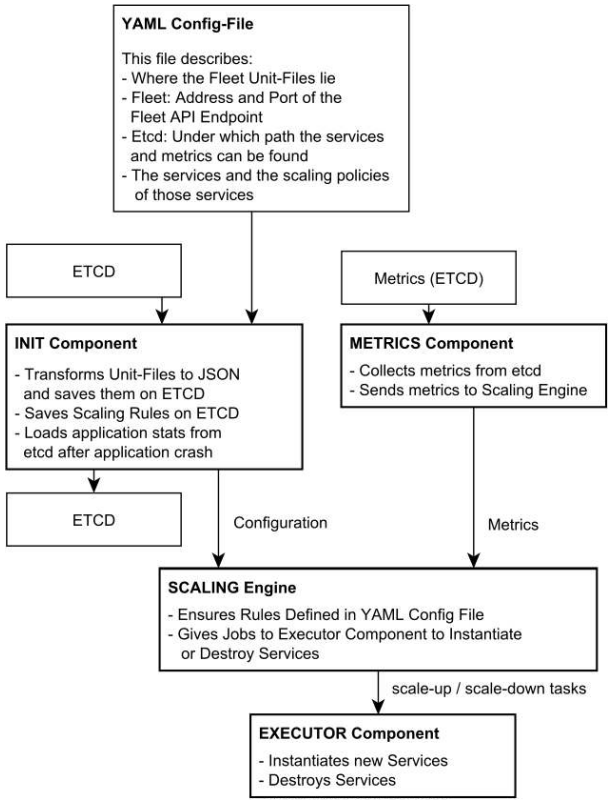

Fig. 6. Dynamite scaling engine components

```
      scale_down_policy:
        ScalingPolicy: apache_scale_down
      scale_up_policy:
        ScalingPolicy: apache_scale_up
[...]
  ScalingPolicy:
    apache_scale_down:
      service_type: webserver
      metric: cpu_utilization
      comparative_operator: lt
      threshold: 15
      threshold_unit: percent
      period: 30
      period_unit: second
      cooldown_period: 1
      cooldown_period_unit: minute
```

The file declares an Apache web service which has at least two running instances. If the scale-out (instance up) policy would be triggered, at most five instances would be running. The policy also defines under what circumstances the service should be scaled-in (instance down). If the CPU utilization of a service of the type `webserver` falls under 15% for at least 30 seconds, the instance which triggered the scaling policy is removed. Figure 6 depicts the components of Dynamite and the workflow between them. The configuration file is read by the *INIT* component which initializes Dynamite and writes the information from the configuration file to Etcd. 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. If the node Dynamite is running on crashes, Fleet will re-schedule the service to another machine and start Dynamite there where it can restore the state from Etcd. The INIT component also takes care of starting the other components of Dynamite. The *METRICS* component is used to collect the metrics stored in Etcd. It regularly requests them and forwards them to the *SCALING* component. This component compares the received metric values with the scaling policy constraints and manages the state of the scaling policies. If a scaling policy triggers a scaling action, it will be sent to the *EXECUTOR* component. This component creates or destroys service instances with the help of Fleet. For more details, we refer to the documentation of the Dynamite implementation[3].

IV. Evaluation Experiments and Results

The evaluation of the CNA design involves stress-testing the scenario application Zurmo in several cloud infrastructure environments. The goal is to confirm the scalability and resilience properties under the influence of availability failures and demand spikes. Emulation within the actual system as opposed to pure simulation has been chosen as scientific technique for validating the application behaviour due to the ability to cover all side effects, including service dependencies [6]. We describe our approach of emulating cloud failures and provoking demand spikes, followed by a presentation of the results.

A. Resilience: Failure Emulation

For the emulation experiments, we have chosen the Multi-Cloud Simulation and Emulation Tool (MCS-SIM)[4] which is an extensible open-source tool for the dynamic selection of multiple resource services according to their availability, price and capacity. Subsequently, we have extended MCS-SIM with an additional unavailability model and hooks for enforcing container service unavailability. Fig. 7 compares the two models. By executing them, the state transition from *unavailable* to *available* causes the corresponding container service to be killed immediately.

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.

B. Scalability: Demand Spike Emulation

The simulation of heavy user influx on the web application requires a configurable HTTP stress testing tool. We have

[3]Dynamite scaling engine: https://github.com/icclab/dynamite/blob/master/readme.md

[4]MCS-SIM Tool: http://nubisave.org/cgi-bin/gitweb.cgi?p=mcssimulation

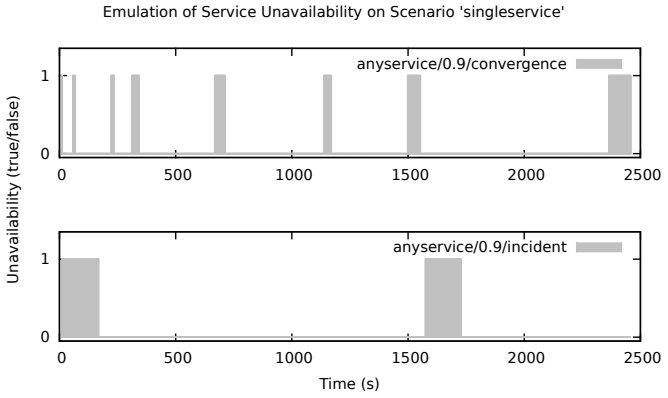

Fig. 7. Comparison of two unavailability models for a single service with 90% average availability

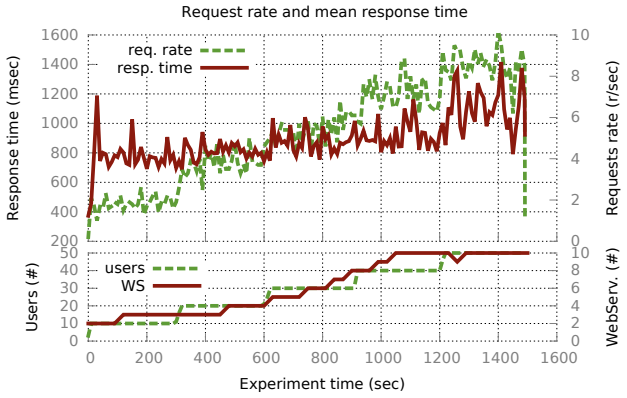

Fig. 8. Sustained request rate and measured response time

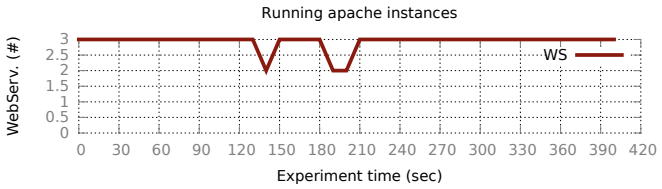

Fig. 9. Resulting service availability

chosen Tsung, an open-source multi-protocol tool which runs distributed across multiple nodes[5]. In our experiment, the tool first records the interaction with Zurmo from a single web browser as intermediate proxy. Then, it replays the interaction from multiple nodes in a cluster to achieve a high load. For our experiment we used a simple step model increasing the demand from 10 to 50 users over 25 minutes.

C. Resilience and Scalability Results

We have run the CNA scenario on Vagrant during development, on OpenStack with Heat orchestration in a private cloud and on Amazon AWS with CloudFormation orchestration on a public cloud. The experiments reported here were performed on the AWS deployment. The input models for demand spikes and service unavailability have been executed in configurations ranging from 3 to 10 virtual machines[6] over which the application containers were deployed.

Fig. 8 shows the response time for the configuration of 10 VMs as a result of the increasing number of users. The bottom graph shows the number of active users over time. The trace-driven workload run by Tsung for each of our simulated users is configured to adopt and average think time of 5 seconds between requests. As a result, the expected request range is supposed to vary between 2 and 10 requests per second. In practice, Tsung waits for a response before starting the think time for submitting the next requests, so the request rate obtained experimentally is slightly lower.

As it can be seen in the upper graph of Fig. 8, while the request rate increases five-fold, the application continues to handle the load while keeping the response time constrained. As a reference, running the same experiment using only 4 VMs results in average response times of over 4 seconds.

Fig. 9 shows the effect of the health management component which ensures a rapid recovery (within seconds) of the Apache containers after failures are induced in the system through our extended version of MCS-SIM.

[5]Tsung Tool: http://tsung.erlang-projects.org/

[6]We used AWS t2.small machines to test a larger distributed scenario with constrained resources requiring autoscaling

V. Discussion and Future Work

Our work has shown that with moderate effort, resulting in mostly re-usable components, applications can be migrated into cloud-native, resilient and scalable services.

We continue to work on CNA and evaluate conceptual extensions such as clustered database backends, new tools such as *fluentd* and additional scenario applications.

References

[1] R. Bahsoon, I. Mistrík, N. Ali, T. S. Mohan, and N. Medvidovic, "The future of software engineering IN and FOR the cloud," *Journal of Systems and Software*, vol. 86, no. 9, pp. 2221–2224, September 2013, Editorial.

[2] F. M. R. Junior and T. da Rocha, "Model-based Approach to Automatic Software Deployment in Cloud," in *Proceedings of the 4th International Conference on Cloud Computing and Services Science (CLOSER)*, Barcelona, Spain, April 2014, pp. 151–157.

[3] M. Caballer, C. de Alfonso, G. Moltó, E. Romero, I. Blanquer, and A. García, "CodeCloud: A platform to enable execution of programming models on the Clouds," *Journal of Systems and Software*, vol. 93, pp. 187–198, July 2014.

[4] J. Spillner, "Secure Distributed Data Stream Analytics in Stealth Applications," in *3rd IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)*, Constanța, Romania, May 2015.

[5] G. Toffetti, S. Brunner, M. Blöchlinger, F. Dudouet, and A. Edmonds, "An architecture for self-managing microservices," in *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud (AIMC)*, Bordeaux, France, April 2015, pp. 19–24.

[6] R. Lübke, D. Schuster, and A. Schill, "Experiences Virtualizing a Large-Scale Test Platform for Multimedia Applications," in *10th International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities (TridentCom)*, Vancouver, Canada, June 2015.