Self-Healing Microservices with Kubernetes

Self-Adaptation in Micro-Service Architectures with Kubernetes – SoSe 2019

Sebastian Schmidl[1]

Abstract: One essential part of a self-adaptive system is its self-healing capabilities. Self-healing systems monitor the running application and try to keep the system in a healthy state to increase availability and adapt to unexpected changes. Therefore, they have to be fault tolerant, mask temporary failures, maintain essential services, and recover from the failures in a finite amount of time to reach the healthy system state again. In cloud environments, self-healing techniques are already used in form of tools that try to achieve continuous availability for cloud services.
In this paper, we take a look at the self-healing capabilities of Kubernetes for microservice architectures in the cloud and compare it to the approaches in self-healing literature. We find that Kubernetes' approach is a form of architecture-based self-healing and that Kubernetes implements all important aspects of self-healing systems. However, Kubernetes depends on the underlying infrastructure to provide fault-tolerant persistent volumes, must run in an highly available setup to be itself resilient in case of Kubernetes master component failures.

Keywords: Self-Healing; Microservices; Cloud Computing; Kubernetes; Distribution

1 Introduction

Cloud Computing has become the de-facto standard of deploying new scalable applications. Companies chose cloud over on-premise or self-hosted environments, because they can deploy their applications more flexible, with higher and dynamically scalable performance, and because prices are very competitive [To15]. However, present cloud environments have to deal with heterogeneous resources and an ever-increasing scale. With this growing complexity failures are more likely to occur and software engineers have to design applications with that in mind. This can be achieved via replication, containment, isolation, and monitoring paired with responsive actions to failures [Bo14].

Microservice architectures are a way to realize containment and isolation of software components in a scalable way. In this approach, the software application is decomposed along business domain boundaries into small, lightweight, and autonomous services. Each service runs as its own application decoupled from the other services and acts as a scaling unit. Services communicate through lightweight REST APIs or asynchronous message

[1] Hasso Plattner Institut, University of Potsdam, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, sebastian.schmidl@student.hpi.de

queues. Microservice architectures embrace failure. If a service relies on other services, it is aware that the other service may not be available or the connection may be slow and the service can deal with the failures [Ne15].

For those complex distributed software systems consisting of hundreds of microservices, deployment and management gets more complex as well. The service management can be simplified by executing the microservices isolated from each other in containers, such as Docker containers [Do19]. Container orchestration tools, such as Kubernetes [Ku19a], can further be used to deploy, scale, and manage containerized microservice applications.

The increasing complexity of modern software systems motivated the development of self-adaptive systems. Those systems introduce an autonomous behavior that takes decisions at runtime and manages the complex underlying software system. This allows the software systems to adapt to unpredictable changes in their environments. Self-adaptive systems combine four self properties, as defined by Ganek; Corbi [GC03]:

self-configuring Systems adapt automatically to dynamically changing environments ("on-the-fly").
self-healing Systems discover, diagnose, and react to failures reducing disruptions and enabling continuous availability.
self-optimization Systems efficiently maximize resource utilization.
self-protection Systems anticipate, detect, identify, and protect themselves from attacks.

This list has been continuously extended and the extended properties are now referred to as self-* properties [PD11].

Essential parts of self-adaptive systems are its self-healing capabilities, which is our focus in this paper. Self-healing systems monitor the running application and try to keep it in a healthy state to increase availability and maintain the essential functionality. Therefore, they have to be fault tolerant, mask temporary failures, and recover from them in a finite amount of time to reach the healthy system state again. In cloud environments, self-healing techniques are already used in form of tools that try to achieve continuous availability for cloud services. Those tools detect disruptions, diagnose failures and recover from them by applying an appropriate strategy [PD11]. As an example, consider Apache Mesos [Th19] and Mesosphere [Me19]. They perform periodic health checks to ensure a leader is available in the cluster at all times. This means they can detect failing nodes and replace them automatically. Other tools are located in the infrastructure layer: Amazon Auto Scaling [Am19] and Google Cloud Managed Instance Groups [Go19] monitor virtual machine (VM) instances and automatically recreate them when they crash or stop. In addition, both tools allow the user to specify application health checks. However, we will focus on Kubernetes [Ku19a].

In this paper, we compare the solutions and approaches proposed in self-healing research literature with the way Kubernetes implements self-healing capabilities for microservices in the cloud. We show which self-healing concepts are already implemented by Kubernetes and where there are still open issues and limitations. The rest of this paper is structured as

follows: Sect. 2 introduces the concept of self-healing systems, summarizes recent research literature in this area, and discusses self-healing challenges in cloud environments. In Sect. 3, we quickly present base concepts of Kubernetes and its architecture. Using this information, we can then explain how Kubernetes implements self-healing capabilities in Sect. 4. The discussion of benefits and limitations of Kubernetes' self-healing capabilities follows in Sect. 5, before concluding this paper in Sect. 6.

2 Self-Healing

Self-healing is an integral part of self-adaptive systems. It combines properties of (i) fault-tolerant systems, which handle transient failures and mask permanent ones to ensure system availability, (ii) self-stabilizing systems, which are non-fault masking and converge to the legal state in a finite amount of time, and (iii) survivable systems, which maintain essential services and recover non-essential ones after intrusions have been dealt with [PD11]. A widely-used definition for self-healing systems is from Ghosh et al. [Gh07]:

> "The key focus [...] is that a self-healing system should recover from the abnormal (or 'unhealthy') state and return to the normative ('healthy') state, and function as it was prior to disruption."

This definition is very broad, but one can argue that the key aspects of self-healing systems are recovery oriented functionalities that bring the system back to the healthy state, which neither sole fault-tolerant systems nor sole survivable systems encompass [PD11].

Similar to an autonomous system, the main component in a self-healing system is the self-healing manager. It runs a control loop with three stages that is a reduced version of the autonomic control loop, also referred to as MAPE-K loop [05]. Fig. 1 shows the mapping of the three stages to the MAPE-K control loop [PD11]. The stages are:

Detect The self-healing manager filters the status information about the running system and reports suspicious events and detected degradations.

Diagnose The diagnosis stage performs root cause analysis on the received reports from the previous stage and calculates a recovery strategy.

Recover In the recovery phase the manager applies the strategies to the system while it avoids any unpredictable side effects.

There are two different ways how a software system can be equipped with the above self-healing capabilities:

The first approach is a software application with *built-in self-healing logic*. This means that the self-healing manager is within the application code and is able to access internal state and mechanisms. This can be an advantage as the application is seen as white box

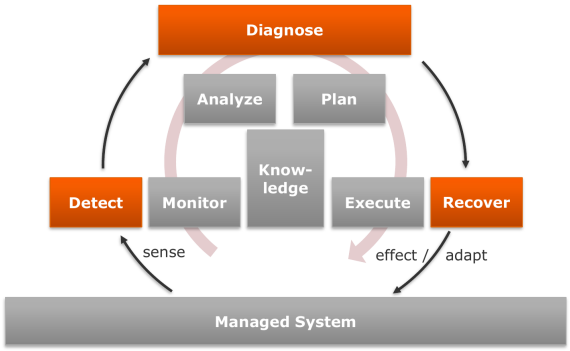

Fig. 1: Condensed autonomic control loop as the self-healing loop

and the self-healing logic can use detailed status information and even domain knowledge for detection, analysis and recovery. At the other hand, this also means that the healing logic is tied to the application increasing coupling and violating the separation of concerns principle. If the application starves the self-healing manager may starve as well.

The second approach is an *external self-healing manager* provided as an infrastructural component or as third-party service. The self-healing logic runs in isolation from the application code and can therefore treat the application only as a black box and has to use external metrics to judge the application's health. It's the current state of the art for monitoring, health management and scaling logic in the cloud [To15]. The external management logic has to be itself resilient, fault-tolerant, and scalable for being able to heal the application. Using third party services or services provided by the infrastructure provider could lead to vendor lock-in. There are also open-source alternatives that include self-healing capabilities and can be used as middleware between cloud infrastructure and the application, such as Kubernetes or Mesosphere.

2.1 Architecture-based self-healing

In the past, most self-healing capabilities were included in the application code itself [SG02]. This has the downside that the self-healing adaption engines and rules are tightly coupled to the application code and can't be reused. Researchers have therefore developed ways to generalize self-healing mechanisms. Architecture-based self-healing systems are one approach, in which external self-healing managers perform repairs on the level of software components and connectors of the monitored systems [DHT02]. The assumption of those approaches is that the component boundaries are the most loosely coupled points in the software system and can be reconfigured to allow changes in the architecture during runtime.

Dashofy et al. present the infrastructure to develop and run an architecture-based self-healing system for event-based applications in [DHT02]. They chose event-based architectures,

because all included components show a significant degree of autonomy. This reduces coupling of the components and makes external changes to the running system possible.

Microservice architectures take isolation of software components to the next level and improve the autonomy of the components (called microservices). This makes applying architecture-based self-healing easier, because services are decoupled, autonomous, and scalable and service boundaries are well-defined. External self-healing managers can monitor the microservices of the application and apply repairs to the failing ones by replacing the service with another one and rerouting the messages. If microservice application are deployed in the cloud, the aforementioned cloud tools (Sect. 1) and Kubernetes can be used to provide self-healing capabilities.

In contrast to external self-healing managers, Toffetti et al. propose a self-healing system with the self-healing logic built into a microservice application. They leverage standard methods from distributed systems (i. e. consensus algorithms) to assign self-healing functionality to some nodes of the distributed application. The selected nodes form a hierarchy and perform different parts of the self-management logic. This means that the self-management logic is distributed in the cluster to overcome the connectedness to the application code [To15].

2.2 Self-healing in cloud environments

As more applications are deployed in cloud environments, developers must consider the implications of running their software in the cloud. Past software was deployed on bare metal or virtual machines, but in the cloud, multiple layers of abstractions and virtualizations are used. Most of the time this is transparent to the application as Platform as a Service (PaaS) solutions are used. This means that compute units are much easier to scale, exchange, and expand making recovery strategies simpler. On the other hand, self-healing strategies in cloud environments differ significantly from traditional ones, because the self-healing manager has no control over the underlying infrastructure and must work on higher abstraction levels.

While systems and software components can fail in various ways and research has come up with general failure classifications and resolutions [PD11, Tab. 1] for any software system, failures in cloud environments can be reduced to a single failure class: a component is detected unreachable. Although an unreachable component can have different root causes on the infrastructure level, the impact on the system is the same. Because application developers have no control about the infrastructure in a cloud deployment, they have to find other ways besides repairing the infrastructure to heal from failures in the cloud. Component failures can be detected via heartbeat messages or latency metrics and a common recovery strategy is the redeployment of the failed component, possibly on a different node.

Research approaches that deal with these challenges for self-healing systems in the cloud are Stack et al. [St17], who developed an hierarchical approach using a master-slave architecture to provide flexible and high available self-healing functionality in the cloud, and Florio;

Nitto [FN16], who use a similar approach than Kubernetes. They developed their own self-healing container orchestration tool, called Gru, for microservice architectures.

3 Kubernetes

Kubernetes is an open-source platform for automating and managing distributed software in the cloud. It heavily relies on container technology and supports declarative configuration of the managed containers. Kubernetes was developed by Google and is open-source software since 2014 [Ku19a].

Kubernetes is built on top of existing PaaS solutions and consists of a master-slave architecture forming a cluster as depicted in Fig. 2. Slave nodes, Kubernetes calls them only "nodes", are responsible for executing and maintaining the actual application via an underlying container runtime. Docker [Do19] is most common. The nodes also run Kubernetes components that manage cluster networking (`kube-proxy`) and interact with the container runtime and the master (`kubelet`). The `kubelet` monitors and manages pods of the local node. Pods are the smallest deployment unit in Kubernetes and consist of the application container (or multiple), connected resources, such as storages or network addresses, and configuration options. Container runtime, `kube-proxy` and `kubelet` form the Kubernetes runtime environment [Ku19b].

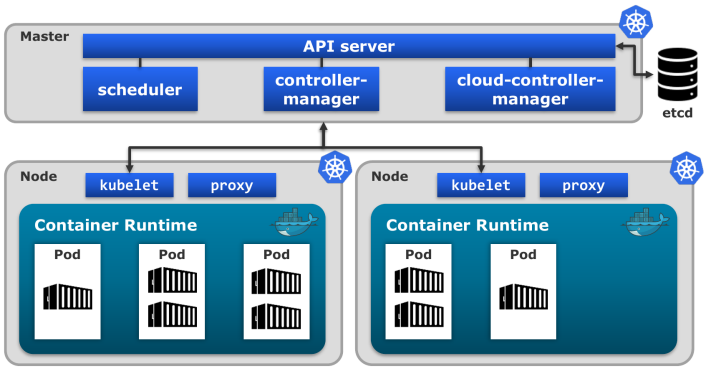

Fig. 2: Kubernetes architecture

The Kubernetes master components provide the cluster's control plane. As shown in the top of Fig. 2, they typically run only on one node that does not run application pods. However, the components can be executed on any node in the cluster. The master components include (i) the `kube-apiserver` that exposes the control options to the user and other software, (ii) an `etcd` [et19] instance as store for all cluster data, (iii) the `kube-scheduler`, which schedules newly created pods to the nodes in the cluster, (iv) the `kube-controller-manager`, which runs Node and Pod Controllers, and (v) the `cloud-controller-manager` to interact with the underlying cloud providers [Ku19b].

For the declarative configuration and management of the application, Kubernetes employs the Kubernetes object model. All entities of the Kubernetes runtime are represented as description objects. The entirety of those objects represents the cluster state. Each object consists of three parts: (i) the object's metadata, such as name, version or labels, (ii) the object `spec`, which describes the desired state for the object, and (iii) the object `status`, which describes the actual state of the object. Users of Kubernetes therefore only provide the object's metadata and `spec` to declare the desired deployment of their containerized applications and policies around how the application should behave. Kubernetes will constantly update the `state` of the objects according to the observed cluster state and takes corrective actions in the cluster to ensure that the cluster state matches the desired one declared in the objects' `spec` [Ku19b].

4 Self-healing capabilities of Kubernetes

In this section we will discuss the self-healing capabilities of Kubernetes. In Sect. 4.1, we start with a short overview how Kubernetes approaches self-healing. We then explain, how Kubernetes implements the three properties of self-healing systems: fault-tolerant in Sect. 4.2, self-stabilizing in Sect. 4.3 and survivable in Sect. 4.4.

4.1 Overview

Kubernetes' self-healing capabilities are spread across various components and functionalities. However, they also perform the three-staged self-healing loop introduced in Sect. 2. Kubernetes' approach to self-healing is comparable to architecture-based self-healing [DHT02; To15]. Its declarative object configuration model resembles the concept of a desired and an actual runtime architecture of the managed application. The desired runtime architecture is defined by the user as deployment configuration using the `spec` part of the objects. Kubernetes then internally creates the actual runtime architecture by continuously monitoring the nodes and pods in the cluster and recording the system state in the `state` part of the objects. This allows Kubernetes to detect failures in the cluster. To recover from those failures, Kubernetes calculates corrective measures form the state deviations, which are applied to the cluster fully automatically. The actual state of the system thereby eventually converges to the desired one.

There are two levels of disruptions that can occur in a Kubernetes deployment: Container failure and pod disruptions. containers are runtime artifacts defined by users and can therefore fail or crash during execution. Those container failures are captured by the restart policy of their pod objects. When the restart policy is set to *Always* or *OnFailure*, failing containers are automatically restarted by the local `kubelet` component with an exponential back-off strategy. In the special case of container failure the self-healing loop is performed by the local `kubelet` component on each node.

In contrast to containers, pods do not disappear until the user or a Kubernetes component destroys them or there is an unavoidable system error. Kubernetes considers the following involuntary disruptions for pods [Ku19b]:

- a hardware failure of the physical machine backing the node
- cloud provider or hypervisor failure makes VM disappear
- administrator deletes VM by mistake
- a kernel panic of the operating system
- a cluster network partition removes the node from the cluster

All those cases deal with failures of nodes or their communication. Therefore, Kubernetes employs *node-controllers* run by the *controller-manager* in the control plane (see Fig. 2) to monitor nodes. They detect those disruptions through a heartbeat-based failure detector and set the phase of all pods that where running on the failed node to *Failed*. This summarizes those failures into one category and propagates it to all the pods running on the node. This means the self-healing logic in Kubernetes only has to consider pod failures, thus the following three sections explain how Kubernetes implements the three properties of self-healing systems introduced in Sect. 2 to heal from those pod failures.

4.2 Fault-tolerant

Kubernetes uses pods and containers to isolate different parts of the managed application. This fits well for microservice architectures and creates failure domains limiting the failure impact. Fault-tolerant applications can be created with Kubernetes by making use of pod replication and `Services`.

Pod replication allows us to run multiple equivalent pods of a microservice at the same time. The `kube scheduler` automatically spreads the pod replicas across the available nodes to increase the failure tolerance [Ku19b]. This allows other replicas to take over the workload if one replica fails due to an involuntary disruption.

To allow external services to communicate with our replicas, we must be able to reroute the traffic from failing pods to the remaining ones. This is done by specifying a `Service`, which exposes the set of replicated pods to the cluster internal and external network [Ku19b]. It performs the load balancing and rerouting of the network traffic to the changing pod instances. This decouples the service access from the actual placement of the pods.

4.3 Self-stabilizing

Pods can be created manually, but this means the user has to take care of recovering failed pods. A better solution is to use Pod Controllers [Ku19b]. They take pod description objects and manage their pods automatically. Pod Controllers execute the Detect – Analyze

– Recover loop and run in the `kube-controller-manager` as a master component. Pod Controllers monitor the health of their managed pods via heartbeats and user-defined liveness probes [Ku19b]. They continuously update the `state` part of the descriptor object to reflect the actual system state. Based on the desired state, the current system state, and the policies in the object, the Pod Controllers derive corrective actions to transition the system from the current state into the desired state. In the case of a failed pod, a controller would for example instruct the deletion of the failed pod and the creation of a new one with the same properties. The Pod Controllers not only contain the self-healing logic for pods, but also handle pod replication and rollout. Using Pod Controllers, Kubernetes is able to recover failed pods and to eventually let the faulty state converge to the desired one. This means applications managed by Kubernetes can be self-stabilizing.

There are four different types of pod payloads, which are unique to their failure handling: Stateless services, stateful services, daemons, and jobs. This is reflected in Kubernetes by different controller types, one for each payload type:

Stateless services as payloads for Kubernetes pods are easy to manage, because they can be destroyed and recreated on all nodes without any special resource dependencies. A stateless service can be defined via a `Deployment` or a `ReplicaSet`. In the case of a pod failure, those Pod Controllers instruct the deletion of the failed pod and the creation of a new one according to the supplied `spec`. The placement of the pods is transparent and done by the `kube-scheduler`. The controllers always try to match the desired number of replicas.

Stateful services can be defined via a `StatefulSet`. Pods managed by this type of Pod Controller have a unique identity including their name, network ID and configuration. They can be connected to `PersistentVolumes`, which provide persistent storage for pods. Once connected to a pod, they are also tied to the pod's identity. If a pod fails, the controller reschedules it, potentially on another node, with the same identity. Therefore, the pod will reuse the assigned `PersistentVolumes` and network IDs. Routing of network traffic must be taken care of by creating a headless `Service` [Ku19b]. Stateful services using `PersistentVolumes` rely on the availability and fault-tolerance of the volumes provided by the underlying infrastructure, such as a PaaS solutions.

Daemons are application components that should run only once on all or selected nodes of the cluster, such as cluster storage, node monitoring, or a log collection component. Daemons can be defined via `DaemonSets`. They ensure that a copy of the daemon runs on all the selected nodes. If a node with a daemon fails, no action is taken, as the desired state is still met, just with a reduced number of nodes. If a new node is added back to the cluster, the `DaemonSet` controller takes care of scheduling the creation of a new copy of the daemon on the newly added node. Recovering failed nodes is not part of Kubernetes.

Jobs run only once. They can be defined using a `Job`. The job Pod Controller ensures that the job is run to completion. This means if the job consists of one pod, this pod must terminate successfully, otherwise it will be restarted. Jobs can be started with a degree of parallelism.

In this case, the user can either specify a number of completions that must succeed or the controller waits for the first pod to successfully terminate. If those conditions are not yet met, the controller will recover the failed pods.

4.4 Survivable

Survivable systems maintain the essential services of the managed application in the case of failure and resource pressure, while non-essential services may experience disruptions and are recovered after the failure has been dealt with. Kubernetes provides two mechanisms to achieve survivability: `PriorityClasses` and `PodDisruptionBudgets` (PDBs) [Ku19b]. They can be used to compile a set of policies that help the scheduler to decide, which pods are essential and how many disruptions a set of replicated pods can handle.

`PriorityClasses` map to integer values, where a higher value indicates higher priority. The user can assign them to pods. Pod priority will affect the scheduling order: Higher priority pods will be scheduled first. In addition, under resource pressure, higher priority pods in the scheduler queue will lead to the eviction of lower priority pods. This is called preemption. If the user assigns high priority `PriorityClasses` to the essential services, the scheduler will do its best to maintain all instances of the essential services. In the case of failures, it will even evict lower priority pods to make room for the essential services if necessary.

To prevent the scheduler from evicting all instances of a lower priority service, the user can specify PDBs. They limit the number of replicated pods that are down due to voluntary disruptions, such as draining or preemption. Unfortunately, the scheduler only considers PDBs on best effort basis during preemption, which limits the applicability of PDBs for self-healing.

5 Discussion

As described in Sect. 4, Kubernetes provides all features required to setup a self-healing system. Nevertheless, it requires the managed application to make use of the provided features and the user to configure the deployment correctly. This includes packing the application parts in containers, writing an application that supports replication and distribution, using `Services`, Pod Controllers, `PriorityClasses`, and PDBs, and setting up external services for persistent storage of stateful services. Compared to implementing new recovery methods or specifying imperative recovery policies, Kubernetes is configured via the declarative definition of the desired system state. This provides a high abstraction and eases the usage of Kubernetes for equipping a system with self-healing capabilities in a cloud environment. The recovery and healing logic is provided by Kubernetes and thoroughly tested in big production systems. Kubernetes also provides a rich API to retrieve the current system state and update the desired state. This can be used to extend Kubernetes with further self-healing logic. However, using Kubernetes for self-healing also comes with some limitations:

- Kubernetes is an external self-healing manager and Toffetti et al. argue that "this approach has intrinsic limits" and requires additional management effort and human intervention and lead to vendor lock-in [To15].
- Kubernetes does not automatically repair failing infrastructure, such as nodes, external load balancers, or `PersistentVolumes`. It relies on the availability and fault tolerance of the underlying infrastructure.
- As Kubernetes is external to the application, it must itself be resilient and fault tolerant. By default, the Kubernetes' master components are only deployed on one node. This does not provide any fault-tolerance. To achieve this, Kubernetes must be deployed in a highly available setup with multiple master nodes, potentially across availability zones. This setup is considerably complex and not yet automated.

6 Conclusion

In this work we showed how Kubernetes can be used to equip a microservice application with self-healing capabilities. Kubernetes monitors the cluster state and takes corrective actions to let the cluster converge to the user-defined desired state. Kubernetes implements fault-tolerance through the replication and isolation of application parts encapsulated in pods and load balancing the network accesses to the pods using `Service`s. On pod failures, other replicas can simply take over the failed node's tasks. After the failures have been dealt with, the cluster converges back to the desired state. Pod Controllers detect pod or node failures, diagnose them, and recover the failed pods by terminating the failed pod and scheduling a new identical one. This reflects the self-stabilizing aspect of self-healing systems. `PriorityClasses` and `PodDisruptionBudgets` (PDBs) help the Kubernetes scheduler on resource pressure and occurring failures to maintain the essential services, while recovering the non-essential services when enough resources are available again. This allows the system to survive extreme stress situations. Kubernetes depends on the underlying infrastructure to provide fault-tolerant `PersistentVolumes` and must be run in an high-availability setup to be itself resilient in the case of Kubernetes master component failures.

References

[05] An Architectural Blueprint for Autonomic Computing, tech. rep., IBM, 2005.

[Am19] Amazon Web Services, Inc.: AWS Auto Scaling, 2019, URL: `https://aws.amazon.com/de/autoscaling/`, visited on: 06/30/2019.

[Bo14] Bonér, J.; Farley, D.; Kuhn, R.; Thompson, M.: The Reactive Manifesto, 2014, URL: `https://www.reactivemanifesto.org`, visited on: 06/04/2019.

[DHT02] Dashofy, E. M.; van der Hoek, A.; Taylor, R. N.: Towards Architecture-based Self-healing Systems. In: Proceedings of the First Workshop on Self-healing Systems (WOSS). Pp. 21–26, 2002.

[Do19] Docker, Inc.: Docker – Enterprise Container Platform for High-Velocity Innovation, 2019, URL: https://www.docker.com/, visited on: 06/20/2019.

[et19] etcd Authors: etcd – A distributed, reliable key-value store for the most critical data of a distributed system, 2019, URL: https://etcd.io/, visited on: 06/22/2019.

[FN16] Florio, L.; Nitto, E. D.: Gru: An Approach to Introduce Decentralized Autonomic Behavior in Microservices Architectures. In: IEEE International Conference on Autonomic Computing (ICAC). Pp. 357–362, 2016.

[GC03] Ganek, A. G.; Corbi, T. A.: The Dawning of the Autonomic Computing Era. IBM Systems Journal 42/1, pp. 5–18, 2003.

[Gh07] Ghosh, D.; Sharman, R.; Raghav Rao, H.; Upadhyaya, S.: Self-healing Systems - Survey and Synthesis. Decision Support Systems 42/4, pp. 2164–2185, 2007.

[Go19] Google, Inc.: Google Cloud Compute Engine Documentation – Instance Groups, 2019, URL: https://cloud.google.com/compute/docs/instance-groups/, visited on: 06/30/2019.

[Ku19a] Kubernetes Authors: Kubernetes – Production-Grade Container Orchestration, 2019, URL: https://kubernetes.io/, visited on: 06/17/2019.

[Ku19b] Kubernetes Authors: Kubernetes Documentation, 2019, URL: https://kubernetes.io/docs/home/, visited on: 06/17/2019.

[Me19] Mesosphere, Inc.: Mesosphere – Focus on building apps, not inrastructure, 2019, URL: https://mesosphere.com/, visited on: 06/30/2019.

[Ne15] Newman, S.: Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, 2015.

[PD11] Psaier, H.; Dustdar, S.: A survey on self-healing systems: approaches and systems. Computing 91/1, pp. 43–73, 2011.

[SG02] Schmerl, B.; Garlan, D.: Exploiting Architectural Design Knowledge to Support Self-Repairing Systems. In: Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE). Pp. 241–248, 2002.

[St17] Stack, P.; Xiong, H.; Mersel, D.; Makhloufi, M.; Terpend, G.; Dong, D.: Self-Healing in a Decentralised Cloud Management System. In: Proceedings of the 1st International Workshop on Next Generation of Cloud Architectures. 3:1–3:6, 2017.

[Th19] The Apache Software Foundation: Apache Mesos – Program against your datacenter like it's a single pool of resources, 2019, URL: http://mesos.apache.org/, visited on: 06/30/2019.

[To15] Toffetti, G.; Brunner, S.; Blöchlinger, M.; Dudouet, F.; Edmonds, A.: An architecture for self-managing microservices. In: Proceedings of the 1st International Workshop on Automated Incident Management in Cloud. Pp. 19–24, 2015.