Seminar Web Engineering in Winter semester 2020/21

Kubernetes-111

Meshu Deb Nath, Diwakar Dangal

Professorship Distributed and Self-Organizing Computer System
Technische Universität Chemnitz
Chemnitz, Deutschland

1. Kubernetes

Edited by Meshu Deb Nath, Diwakar Dangal

In the last 5 years, Kubernetes has become the most popular orchestration platform. It provides high availability and scalability through diverse self-healing and scaling mechanisms. Therefore, a huge community rise of this open-source project makes it the most popular orchestration platform right now. Kubernetes makes it easy to deploy and automatically manage a microservice-based complex application.

In this report, we are going to discuss Kubernetes and its architecture. we will demonstrate a microservice deployment in the Kubernetes cluster and discuss the horizontal pod autoscaling (HPA) **[1]**, the most important features of Kubernetes.

2. Container Technology

Edited by Diwakar Dangal

Containers are the one which shares the properties of the operating system. They are Loosely coupled, distributed and independent to each other, therefore, which can be deployed and managed dynamically by the smaller teams which result in flexibility and effectiveness in every aspect of the entire process **[1, 13, 14]**.

In a nutshell container technology is treated as one of the best practices for medium and large-scale industries because of its ability to make maximum use of the resources available **[1]**.

2.1. Container Orchestration

Container orchestration is a set of actions undertaken by application owners for selecting, deploying, monitoring and controlling the different configuration of operating resources to guarantee in the quality of services **[14]**. Container technology gives rise to the birth of a new methodology called container orchestration. While handling different applications, continuous monitoring can be harsh. Also, the allocation of containers and solving pods could be a hectic job. So, Automation technology provides self-healing process, distribution of cut-off resources and auto-generating of the services.

Container orchestration not only helps in the first step of development of the application, but it also manages the life cycles of containers throughout the process especially in large, dynamic environments **[25]**. Processes that orchestrations generally control and automate, for example, scaling, allocating the resources, load balancing and traffic routing, monitoring the container health, configuring and scheduling of the process, making the interaction between containers secure.

2.2. When to use a plain docker, when Kubernetes, when none of them?

Edited by Meshu Deb Nath, Diwakar Dangal

Docker **[22]** and Kubernetes **[1]** are not alternative technology to the other. Docker is a software that is used for containerized applications **[22]**. Containerization is a technique of running an application so that the application can be isolated from the system. It can create an illusion for the application that it has working on its OS instances. Hence, Docker prepares to run, create and manage containers on a single OS.

Kubernetes automates the process of scaling, updating, maintaining, and sorting containers. While Docker enables us to have containers in the first place. On the other side, Docker or Kubernetes will be an inadequate choice when the hardware resources are limited **[1, 22]**.

2.3 What are alternatives to Kubernetes?

Edited by Diwakar Dangal

Kubernetes **[1],** Docker swarm **[22]**, and Mesos **[23]** all of these technologies belong to the class of Dev Ops 3 layers: resource management, scheduling, and service management **[18]**. Based on resource management availability such as memory, CPU, GPU, disk space, etcMesos has a greater functionality **[18]**.

Based on the schedule layer comparison Kubernetes performs better in all of this resources like placement, replication, scaling, readiness checking, resurrection, rescheduling, rolling deployment, and co-location **[18]**. Based on the service management layer comparison Kubernetes and Mesos are in equivalent labels while here, Kubernetes have partial levels of dependencies and Mesos has less effectiveness in load balancing **[18]**.

Apart from the comparison between these technologies application owner, Docker swarm **[22]** is easy to organize and set up, with a flexible API but has limited customization.

Mesos are best for systems that have large and complex architecture and are also designed for maximum redundancy. It is a stable platform, however overly complex for small-scale systems that operate between 10-20 nodes.

So, in a nutshell, it depends on the company structure and working methodology for choosing the orchestration technique for them.

3. Kubernetes

Edited by Diwakar Dangal

Containers bundle up the application of user choice in the real-time environment, there could be thousands of containers. It can be a hectic job to monitor them and perform designated solutions accordingly. Now, here Kubernetes comes to the rescue, it provides an automated environment and handles a framework to run distributed systems silently in the background. Kubernetes can detect the failed application and provides deployment patterns accordingly **[1]**. For example, if some container goes down it will start a new container and can easily manage a canary deployment throughout the process **[1]**.

What's the specialities of Kubernetes and why user needs to use it.

- Containers have their DNS name and IP address. Here in the condition if traffic to the container gets high, it balances the load and distributes the network traffic which makes deployment stable **[1]**. So, service discovery and load balancing can be achieved.
- Kubernetes provides storage orchestration which automatically allows mounting storage of user choice, for example, local storage, public clouds, and many more **[1]**.
- Automated rollout and rollbacks are possible because it gives a flexible environment to create new containers, remove existing or modify them in the desired state of the application owner**[1]**.
- Automatic bin packing can be done by giving instruction to Kubernetes on how much CPU and memory(RAM) could be used according to process and container needs for utilizing the resources**[1]**.
- Restarting the container which fails, replacing or killing non-functioning containers according to user-defined instruction and cannot participate until they are ready to serve **[1]**. Hence, there is a self-healing in Kubernetes.
- It can store and manage confidential data i.e password, OAuth token, and SSH keys **[1]**.

In a nutshell, secret and configuration management is possible and because of these many features, Kubernetes becomes the most popular orchestration tool for the enterprise-scale industry in these current circumstances.

3.1 Kubernetes Components

Edited by Diwakar Dangal

This section described the main components of a Kubernetes cluster and their interconnection between them.

3.1.1 Pod

The most basic execution and element in Kubernetes is a pod, which consists of multiple containers and deployment instructions for their execution **[2]**. Each pod represents a single instance for the process and necessary execution guidelines to be carried out throughout the process and always belongs to a specific namespace **[2]**. In this sense, it is referred to as a replica, hence during the process of deployment of an application, the desired number of replicas and amount of requested resources need to be specified in the YAML file of the application **[2]**.

Moreover, each pod is assigned with a unique IP address which allows an application to scale up for the instance. When the application requires more instances instead of adjusting new one can be created in the available pod to share the load of applications **[2]**.

3.1.2 Service

It has a permanent IP address in the cluster, it can be attached to every node, if the pod dies in the process it remembers the current state of the pod when the new pod is formed, the operation begins from the previous state where it had died **[1, 2]**. Hence, when the pod dies service remains alive.

3.1.3 Ingress

It provides load-balancing, SSL-termination and virtual hosting for the Kubernetes cluster **[1]**.

3.1.4 Config Map

It is an external configuration of the application, it contains configuration data like URLs of a database or some other service that the application owner used. So in Kubernetes users need to connect it to the pod so that pod gets the data that the config map contains **[24]**.

3.1.5 Secret

It can be used to store secret credentials, for example, base 64 encoded format, so secrets would contain things like credentials **[24]**. Hence, it store's confidential information like passwords and certificates.

3.1.6 Volumes

A Volume is a directory that can be accessed by a pod for containers. Kubernetes uses its own abstraction of volumes, allowing all containers to share data and remain available until the pod is dismantled **[1]**.

3.1.7 Node

Node is the place of the machine where Kubernetes is installed either it can be a physical or virtual machine **[1]**. Node is the part of a machine where Kubernetes launches its container inside of pods for the execution of processes.

3.1.8 Containers

Containers are self-contained, standardized execution enclosures that are composed of dependencies along with the binaries and libraries required to get similar behaviour wherever the user runs it **[1]**.

3.2 Architecture of Kubernetes

Edited by Diwakar Dangal

This section described the main architecture of Kubernetes.

3.2.1 Kubernetes Cluster

Kubernetes **[1]** cluster at least consists of one master or multiple and several worker nodes. In production environments, there could be multiple masters and worker nodes to ensure the high availability of the cluster.

3.2.2 Master Node

In master nodes, we have completely different processes running inside, and these four processes that run on every master node that controls the cluster state and the worker nodes.

3.2.2.1 Kube-controller Manager

This is responsible for ensuring that the cluster is running in the desired state**[2]**. For instance, 3 pods are running if one pod gets collapsed, now it's Kube-controller manager job is to create new replicas and carry on process execution **[2, 24, 1]**.

3.2.2.2 Kube-scheduler

It decides how events and jobs will be scheduled across the cluster considering several factors including nodes, resources availability, and the policy set by the application owner **[2, 4]**.

3.2.2.3 Kube-API Server

Kube-API server is like a cluster gateway that gets the initial request of any updates into the cluster or even the queries from the cluster. It makes sure it is an authorized request and if everything is fine then it will forward our request to other processes in order to schedule the pod **[2]**.

3.3 System Requirement

Edited by Meshu Deb Nath

This section described the hardware and software requirements of Kubernetes.

3.3.1 Hardware Requirements

Kubernetes cluster can be set up on a Linux based operating system such as Ubuntu, Debian, CentOS, Red Hat Enterprise Linux, Fedora, HypriotOS, Flatcar Container Linux. This operating system should have at least 2GB or more RAM per machine with a minimum of 2 CPU or more. Also, Unique hostname, MAC address, and product_uuid are required for every node. It also requires certain ports for the operation **[1]**.

3.3.2 Typical Node and Application Sizes

While creating a Kubernetes cluster choosing how many worker nodes and what type of nodes is challenging. The total computational capacity of the cluster (in terms of CPU and memory) is dependent on the sum of all the nodes of the cluster **[19]**. There are several ways to find out the desired target capacity of a cluster. For example, if our application needed a cluster setup with a total capacity of 8 CPU cores and 32 GB of RAM. Therefore, we can design it in two possible ways such as a cluster with few large nodes which is 2 nodes (4 CPU and 16 GB of RAM for each). Another way of design is a cluster with many small nodes which is 4 nodes (2 CPU and 8 GB of RAM for each node) **[19]**.

The best design depends on application requirements. For example, if the application requires 10 GB of memory then the cluster with a few large nodes would be the best choice. Because it can be designed with fewer machines than many smaller nodes hence it is cost-effective. It can easily fit the resource-hungry application into the worker nodes. But it has some issues such as a large number of pods per node thus it makes the system slow. There are some reports of nodes being reported as non-ready **[20]**. Therefore, Kubernetes recommend a maximum number of 110 pods per nodes **[1]**.

Alternatively, if the application requires high availability with 10-fold replication therefore, designing the cluster with many smaller nodes would be the best design **[19]**. Because it can reduce the blast radius for example, if we have 100 pods in 10 nodes then each node will be responsible for on average 10 pods. Thus, if one of the nodes fails, it barely impacts the total workload. Also, many smaller nodes allow highly replication that means during the nodes failure scenario, there is a possibility of another replica

maintaining the app's stability all the time. It has also some drawbacks such as pods limitation, lower resource utilization, more system overhead etc **[19]**. So, achieving a target capacity depends on the specific application requirements.

3.4 Deployment Strategy

Edited by Meshu Deb Nath

One of the biggest challenges in cloud technology is how to deploy applications more instantly and cost-effectively. It is important to choose the right strategy to make a reliable transition during an application update procedure. System availability measured by a given time limit which is a nonfunctional requirement **[15]**. High availability is only achieved when the system is 99.999% available at the time **[16]**. We are going to discuss four types of deployment strategies here.

- Rolling Deployment: It is the standard default deployment strategy which gradually updates the pods, one by one and replaces pods with the newer version pods without cluster downtime **[7]**.
- Recreate Deployment: This deployment killed all the existing pods and replaced them with new instances from the latest updates **[17]**.
- Blue-Green Deployment: In this deployment strategy an old version referred to Green and newer version referred to blue get deployed simultaneously but the users only have access to the Green version **[17]**.
- Canary Deployment: It is more similar to Blue-Green deployment but more controlled and used a progressive delivery phase for the deployment. It is convenient for error and performance monitoring but slow rollout and sometimes needs additional tools for shifting traffic, for example, Istio or Linkerd **[17]**.

4. Demonstration

Edited by Meshu Deb Nath

We set up a Kubernetes cluster of 4 nodes, consisting of 1 master node and 4 worker nodes which has 1 core for each machine with 4 GB of Ram. (we ignored the minimum core recommendation due to resource limitation). Linux Debian Kernel version 4.19.152-1 (2020-10-18) is the OS on all VMs and Docker 18.09 is running as the container engine with Kubernetes 1.15.2 is running on all nodes. For the demonstration, we use an application developed using microservice patterns. The pod deployment YAML provided to the deployment controller contains the container image of a Backend service, a Frontend Service and a MongoDB. We also define three services for the three pods with one as a load balancer type to make an endpoint for the outside of the cluster.

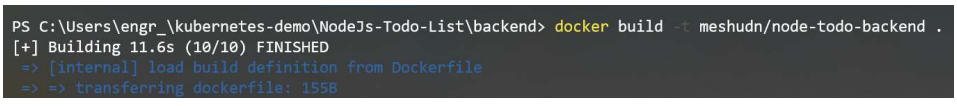

Figure 1. Command for building containerize image for backend microservice.

```
PS C:\Users\engr_\kubernetes-demo\NodeJs-Todo-List\backend> docker push meshudn/node-todo-backend:latest
The push refers to repository [docker.io/meshudn/node-todo-backend]
71cce4405f47: Pushed
fa206263343c: Pushed
```

Figure 2. Command for pushing docker image to docker hub.

We created our backend microservice using express js, front-end microservice using Vue js and containerized these microservices using docker build **[21]**(see **figure 1**). Then, we pushed and stored them in the Docker Hub **[20]** (see **figure 2**).

4.1 Components Overview of Our Cluster

Edited by Meshu Deb Nath

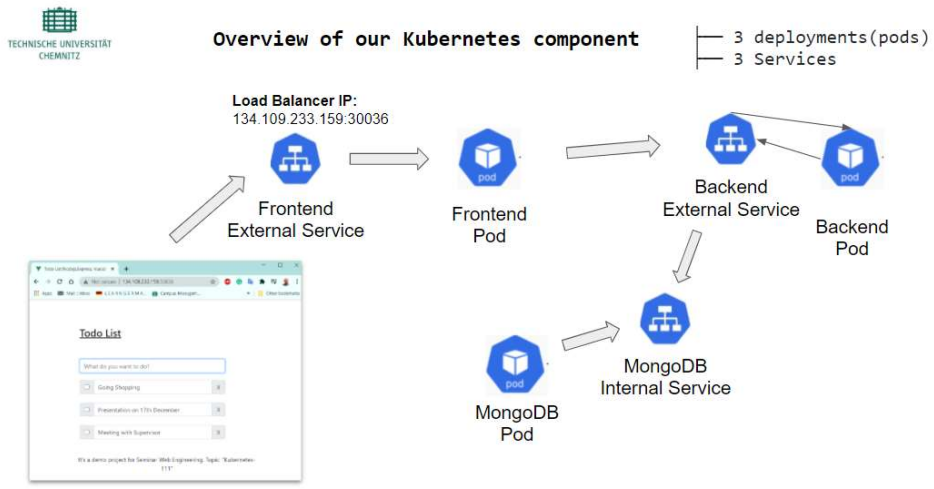

Figure 3. Overview of Pods and Services deployed in our cluster.

According to **figure 3**, a user can access our Frontend external service via a load balancer IP address. Therefore, the external service of Frontend pods calls the Backend external service which can get the data from the MongoDB through an internal service.

4.2 Deployment Templates

Edited by Meshu Deb Nath

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mongodb-deployment
  labels:
    app: mongodb
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mongodb
  template:
    metadata:
      labels:
        app: mongodb
    spec:
      containers:
      - name: mongodb
        image: mongo
        ports:
        - containerPort: 27017
```

Figure 4. Deployment yaml template to create pods of MongoDB.

```
apiVersion: v1
kind: Service
metadata:
  name: mongodb-service
spec:
  selector:
    app: mongodb
  ports:
    - protocol: TCP
      port: 27017
      targetPort: 27017
```

Figure 5. Deployment yaml template to create service of MongoDB.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend-deployment
  labels:
    app: backend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: backend
```

```
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
      - name: backend
        image: meshudn/node-todo-backend:latest
        env:
        - name: MONGO_URL
          value: "mongodb://mongodb-service:27017/todos"
        - name: APP_PORT
          value: "80"
        ports:
        - containerPort: 80
```

Figure 6. Deployment yaml template to create pods of Backend.

```
apiVersion: v1
kind: Service
metadata:
  name: backend-service
spec:
  selector:
    app: backend
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 30007
```

Figure 7. Deployment yaml template to create service of Backend.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  labels:
    app: frontend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
```

```
  spec:
    containers:
    - name: frontend
      image: meshudn/node-todo-frontend:latest
      env:
      - name: BACKEND_URL
        value: "http://134.109.233.159:30007/"
      ports:
      - containerPort: 8080
```

Figure 8. Deployment yaml template to create pods of Frontend.

```
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  selector:
    app: frontend
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
      nodePort: 30036
```

Figure 9. Deployment yaml template to create service of Frontend.

We created a deployment YAML to deploy MongoDB instances and services in the port 27017 (**figure 4**, **5**), an instance of backend microservices and load balancer service with 30007 nodePort (**figure 6**, **7**), a frontend instance and services with an external load balancer IP address at NodePort 30036 (**figure 8**, **9**). Our final endpoint is http://134.109.233.159:30036.

4.3 Autoscaling

Edited by Meshu Deb Nath

One of the most important features of Kubernetes is auto-scaling. It monitors applications and services built with container technology without the help of human interaction. There are three auto scalers in the latest version of Kubernetes.

- Horizontal Pod Autoscaler (HPA) **[1]** handles pod adjustments more dynamically and powerfully by monitoring resource consumption of the applications.

- Vertical Pod Autoscaler (VPA)**[1]** instantly changes the YAML specifications, such as pods minimum requirement and maintains the number of working pods. Therefore, it kills the existing pods and thus disrupts the state of the applications and services.
- Cluster Autoscaler (CA) **[1]** raises the number of nodes only when it is not possible to schedule pods on the existing nodes. Cluster Autoscaler only works on cloud platforms run by the commercial company such as Google Cloud Platform **[11]** and Amazon Web Services **[10]**.

4.3.1 Horizontal Pod Autoscaling (HPA)

In Kubernetes, HPA is a great tool that automatically increases or decreases the number of pods based on the overall computational and processing power of the application's without affecting the current running pods (see **Figure 10**). HPA is a controlled loop implemented by the controller manager. In every 15 seconds (called a sync cycle), the Kube controller manager compares the metrics (collected from either the resource metrics API or the custom metrics API) with the resource utilization **[1]**.

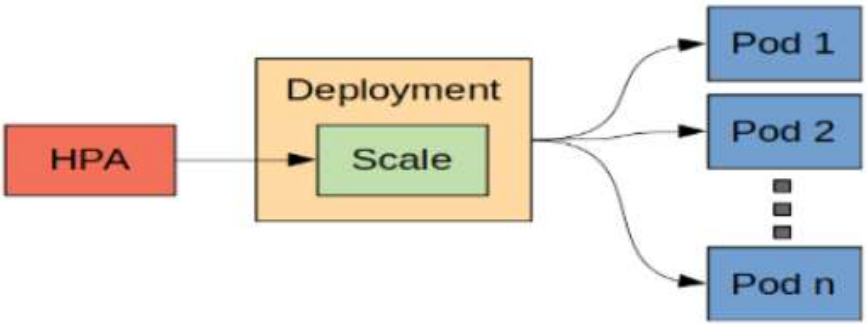

Figure 10. The Architecture of Kubernetes HPA **[12]**.

The HPA controller operates according to the algorithm in equation **(1)**.

desiredReplicas = currentReplicas * (currentMetricValue / desiredMetricValue) **[1]** (1)

Where desired replicas is the number of pods after autoscaling on the other side current replica defined as the number of pods which are currently running, current metric value denoted as the latest collected metric value and finally, the target threshold is defined as desiredMetricValue.

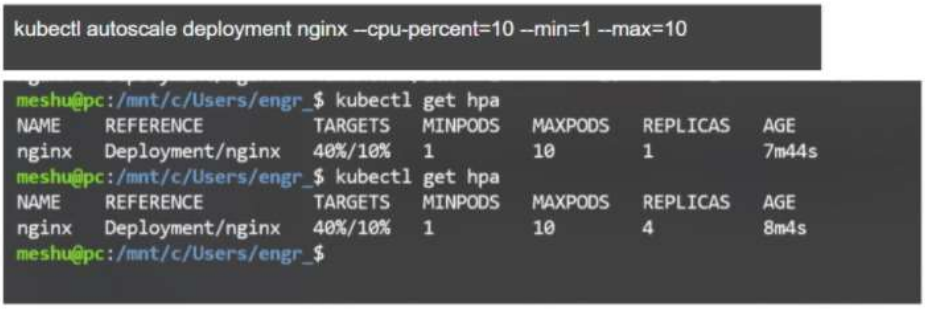

Figure 11. Metric Data of Demo-application.

Figure 11. shows the configuration and metrics of our Demo application. Where "MINPODS" and "MAXPODS" are the requirements for the Kubernetes cluster which refer to the minimum and maximum number of pods. In the figure, the minimum pod's requirement for the cluster is 1 and the maximum is 10. The Kube controller manager continuously tracks this replica set number and compares it with the above requirements. For the scaling operation, CPU usages are going to be used as a metric. We set a 10% threshold for the CPU usages. Therefore, when the application reaches the 10% threshold, HPA will automatically increase the number of pods according to equation **(1)**.

For example, according to **figure 11**, if the current metric value is 40% and the desired value is 10%, the number of replicas will be (1 x (0.40 / 0.10)) or 4. Therefore, the Kube controller manager calls the Kube API server to create 3 more pods. If the current metric value declines to 20% CPU usage, then the desired replicas will be (4 x (0.20/0.10)) or 8. Therefore, 4 more pods will create. On the other side, If the current value is instead 5%, HPA will halve the number of replicas that means 4, since (8 x (0.05 / 0.10)) or 0.5 [2]. To normalize the metrics fluctuations, a little delay period is maintained before removing a pod from the cluster which is 5 minutes **[1]**. HPA will stop the scaling if the ratio is adequately close to 1.0 **[1]**.

Furthermore, HPA can follow several metrics at a time, each of the metrics has its own threshold value. HPA will operate its scaling procedure when any of these metrics crosses its threshold mark **[2]**.

5. Conclusion and Research Problems

Edited by Meshu Deb Nath and Diwakar Dangal

In this report, we have provided an overview of Kubernetes, when to use it and when not and its powerful feature HPA. We demonstrated a microservice application to showcase the use of Kubernetes. Unlike Kubernetes built-in monitoring system which only utilizes the CPU, further research can be made to take the metrics from the application perspective. Also, we think more deployment flexibility can be made in the future.

6. Bibliography

[1] Kubernetes. [Online], Available: **https://kubernetes.io/docs/** (Accessed in 15.12.2020)

[2] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, and S. Kim, "Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration", Sensors, vol. 20, no. 16, p. 4621, 2020. [Online], Available: **https://www.mdpi.com/1424-8220/20/16/4621**. (Accessed in 15.12.2020)

[3] T. Menouer, "KCSS: Kubernetes container scheduling strategy", The Journal of Supercomputing. The Journal of Supercomputing, 2020. [Online], Available: **https://link.springer.com/article/10.1007/s11227-020-03427-3**. (Accessed in 11.12.2020)

[4] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned", 2018. [Online], Available:**https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8457916**. (Accessed in 11.12.2020)

[5] Q. Wu, J. Yu, L. Lu, S. Qian, and G. Xue, "Dynamically Adjusting Scale of a Kubernetes Cluster under QoS Guarantee", 2019. [Online], Available:**https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8975761**. (Accessed in 10.12.2020)

[6] N. Dragoni, I. Lanese, S. T. Larsen, M. Mazzara, R. Mustafin, and L. Safina, "Microservices: How To Make Your Application Scale", in MultiMedia Modeling, MultiMedia Modeling, 2018, pp. 95–104. [Online], Available: **https://link.springer.com/chapter/10.1007/978-3-319-74313-4_8**. (Accessed in 11.01.2021)

[7] Lukša, Marko. "Kubernetes in Action." 2018, doi:10.3139/9783446456020. [Online], Available:**https://pdfs.semanticscholar.org/75f2/d163fa0d73fa6a74c263608a169be8405643.** (Accessed in 11.01.2021)

[8] Vayghan, Leila Abdollahi et al. "Kubernetes as an Availability Manager for Microservice Applications." ArXiv abs/1901.04946 (2019): n. pag. [Online], Available: **https://arxiv.org/ftp/arxiv/papers/1901/1901.04946.pdf** (Accessed in 13.01.2021)

[9] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe and F. Khendek, "Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned, " 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), San Francisco, CA, 2018, pp. 970-973, doi: 10.1109/CLOUD.2018.00148. [Online], Available: **https://ieeexplore.ieee.org/abstract/document/8457916** (Accessed in 20.01.2021)

[10] Amazon Web Services. [Online], Available: **https://aws.amazon.com**. (Accessed on 16 Jan 2021).

[11] Google Cloud Platform. [Online], Available: **https://cloud.google.com**. (Accessed on 16 Jan 2021).

[12] D. Balla, C. Simon, and M. Maliosz, “Adaptive scaling of Kubernetes pods”, 2020. [Online], Available: **https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9110428**. (Accessed on 16 Jan 2021).

[13] C. Pahl, "Containerization and the PaaS Cloud, " in IEEE Cloud Computing, vol. 2, no. 3, pp. 24-31, May-June 2015, doi: 10.1109/MCC.2015.51. [Online], Available: **https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7158965**. (Accessed on 16 Jan 2021)

[14] B. D. Martino, G. Cretella, A. Esposito, (2015) Advances in applications portability and services interoperability among multiple clouds, IEEE Cloud Computing 2 (2) (2015) 22-28. [Online], Available: **https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7116435**. (Accessed on 17 Jan 2021)

[15] M. Toeroe and F. Tam, Service Availability: Principles and Practice. John Wiley & Sons, 2012. [Online], Available: **https://books.google.de/books** . (Accessed in 13.01.2021)

[16] M. Nabi, M. Toeroe, and F. Khendek, “Availability in the cloud: State of the art,” J. Netw. Comput. Appl., vol. 60, pp. 54–67, Jan. 2016. [Online], Available: **https://www.sciencedirect.com/science/article/abs/pii/S1084804515002957**. (Accessed in 12.01.2021)

[17] Kubernetes Deployment Strategy. [online], Available: **https://www.weave.works/blog/kubernetes-deployment-strategies** (Accessed in 15.01.2021).

[18] I. M. A. Jawarneh et al., "Container Orchestration Engines: A Thorough Functional and Performance Comparison," ICC 2019 - 2019 IEEE International Conference on Communications (ICC), Shanghai, China, 2019, pp. 1-6, doi: 10.1109/ICC.2019.8762053. [Online], Available: **https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8762053**. (Accessed in 15.01.2021)

[19] Daniel Weibel, “Architecting Kubernetes clusters — choosing a worker node size”. [Online], Available: **https://learnk8s.io/kubernetes-node-size (Accessed in 18.01.2021)**. (Accessed in 18.01.2021)

[20] Ready/NotReady with PLEG issues. Github [Online], Available: **https://github.com/kubernetes/kubernetes/issues/45419** . (Accessed in 18.01.2021)

[21] Docker Hub. [Online], Available: **https://hub.docker.com/**. (Accessed in 18.01.2021)

[22] Docker. [Online], Available:**https://www.docker.com/** . (Accessed in 18.01.2021)

[23] Mesos. [Online], Available:**http://mesos.apache.org/**. (Accessed in 18.01.2021)

[24] Container Journal. [Online], Available:**https://containerjournal.com/** . (Accessed in 18.01.2021)

[25] Casalicchio, Emiliano. "Autonomic Orchestration of Containers: Problem Definition and Research Challenges." VALUETOOLS. 2016. [Online], Available: **https://www.researchgate.net/profile/Emiliano_Casalicchio/publication** . (Accessed in

09.01.2021)