Kubernetes [1,2]

IT Systems, Summer Term 2026

Dr. Matthes Elstermann

The topics of cloud computing and Kubernetes are addressed in two presentations, of which this is the second one. It focuses on Kubernetes as container orchestrator for cloud infrastructures.

1 Introduction

Please start with a review of earlier concepts as suggested next.

1.1 Retrieval practice

- What is digital sovereignty?
  - Introduced as course goal
  - Revisited in OS part
    - With free software as precondition
      - Similarly for free firmware (and hardware)
- What is Kubernetes?
- What are scalability and replication?
- What is HTTP?
- What is an IP address?

Please take a brief break and write down answers to these questions, without using previous class material.

Agenda

- Part 1
  - Introduction
  - Distributed Systems
  - Cloud Computing
  - Serverless Computing
- Part 2
  - Container Orchestration
  - Kubernetes (K8s)
  - K8s Examples
  - Conclusions

Given the fundamentals of containerization and cloud computing, we now introduce container orchestration in general, before we look at Kubernetes as dominant software project, followed by examples that highlight essential concepts.

2 Container Orchestration

We consider cloud infrastructures with containerized applications. Depending on an application's architecture, it may consist of hundreds of containers, e.g., with microservice architectures (which are beyond our scope).

Even with simple architectures, say a web application with JavaScript frontend and some backend including a database, lots of containers may be used for fault tolerance and scalability, e.g., with replication.

A container orchestrator is the software that manages the containers for an application or a set of applications.

[1] This PDF document is an inferior version of an OER in HTML format; free/libre Org mode source repository.

[2] Material created by Jens Lechtenbörger; see end of document for license information.

2.1 Orchestrator Features

Container orchestrators provide features listed subsequently.

- Resource limit control
- Scheduling
- Load balancing
- Health check
- Fault tolerance
- Autoscaling

Resource limits may define constraints for reservations of CPU or memory resources for containers. The orchestrator communicates such constraints to container managers, which in turn enforce them for containers.

Scheduling by the orchestrator determines which containers (or pods in Kubernetes) to assign to what machines of the cluster. Thus, this notion of scheduling is unrelated to CPU Scheduling in OSs.

Load balancing aims to distribute incoming requests among multiple container instances. The simplest policy may just use round-robin assignments, but more complex policies based on actual load are possible as well.

Health checks serve to determine whether a container is still available, i.e., able to answer requests.

With health checks, the orchestrator can provide fault tolerance: If a health check fails, the container is considered to be faulty and can be destroyed, to be replaced by a newly started container. In addition, containers can be replicated, and the orchestrator makes sure that a predefined number of healthy replicas is available.

Finally, depending on the current load, autoscaling automatically adds and removes containers (for horizontal scaling) or resizes assigned resources (for vertical scaling).

(Source: (Casalicchio 2019))

3 Kubernetes (K8s)

Let us see some details for the container orchestrator Kubernetes.

The typical abbreviation for Kubernetes is K8s, a numeronym of first and last letters, with the number of missing characters in between.

3.1 Assorted Facts

- K8s is a free container orchestrator

Figure 1: "Kubernetes logo" under Kubernetes Branding Guidelines; from GitHub

- Originally developed at Google
- Maintained by Cloud Native Computing Foundation (CNCF)
  - (Project of the Linux Foundation)
  - Variety of distributions
  - **The** cloud infrastructure (end of 2024), "production use + those piloting or actively evaluating Kubernetes = 93% of respondents, meaning it's going to be hard to find an organization not using K8s at some level"

Kubernetes was originally developed by Google as container orchestrator. Nowadays, it is free software maintained by the Cloud Native Computing Foundation, which in turn is a project of the Linux Foundation. Kubernetes comes in a variety of distributions, and it is, and has been for some years, the dominant cloud infrastructure.

- "Datacenter as a Service"
  - Declarative description of cluster with compute, storage, networking
    - YAML files

Kubernetes is also informally called datacenter as a service, as it enables the management of functionality and services for entire datacenters. Importantly, Kubernetes users declaratively describe entire clusters with abstractions for compute, storage, and networking.

Configuration for Kubernetes clusters is usually kept in simple text files using YAML syntax. Some examples follow on later slides.

3.2 Architecture Diagram

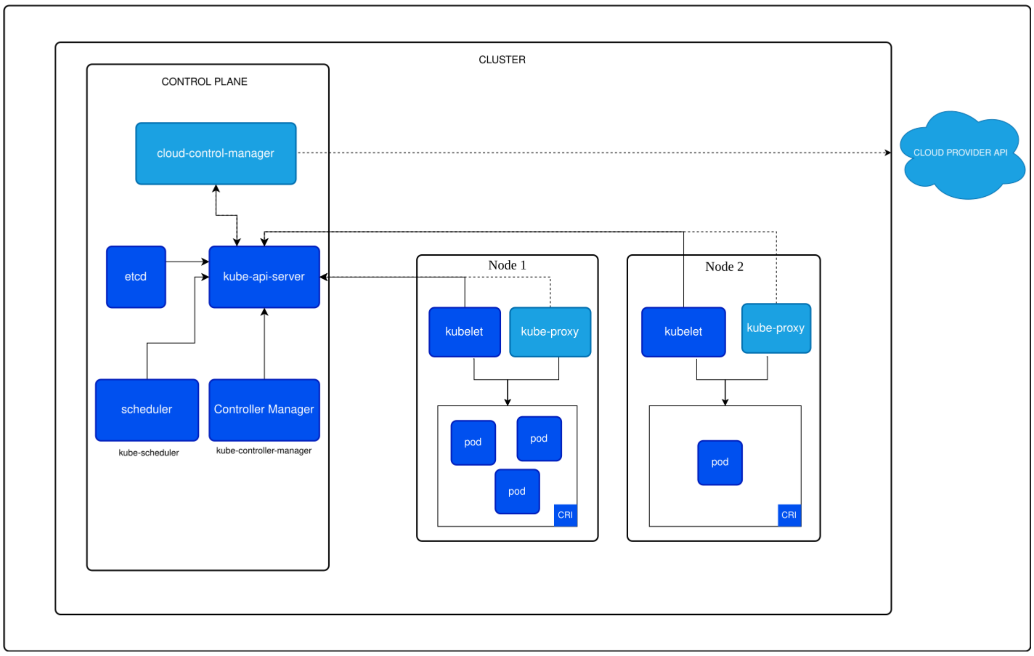

Figure 2: "Kubernetes Cluster Architecture" by © 2024 The Kubernetes Authors under CC BY 4.0; from Kubernetes Documentation

This diagram illustrates the differentiation of the so-called **control plane** with orchestration tasks from ordinary **worker nodes**.

To the right, we see Kubernetes Nodes as worker nodes, where Pods are running, based on containers. (In the figure, "CRI" is short for Container Runtime Interface, which provides an abstraction for a variety of containerization implementations, including Docker.)

A **node** is just a machine, possibly virtualized, on which to run application containers. However, Kubernetes does not assign containers to nodes. Instead, the basic unit of scheduling and deployment in Kubernetes is called **pod**. Pods may just contain a single container each, but more complex use cases exist as well. In general, each pod implements a "part" of an application, e.g., different pods for frontend, caching, API, and database. For the management of entire applications, Kubernetes includes the concept of **workloads**, which manage sets of related pods. (However, we will not consider workloads subsequently.)

Importantly, being a unit of scheduling means that containers of a pod are started and stopped together, co-located on the same node. Horizontal scaling of applications is achieved by adding pod replicas.

Before continuing with other names in the right part, let us focus on the Kubernetes control plane.

The control plane orchestrates communication across the entire Kubernetes cluster. It exposes an API and interfaces to manage the lifecycle of containers and stores cluster configuration information in a distributed key-value store, called etcd. The **Kubernetes API** offers the interface to configure and monitor the cluster. Scheduling in Kubernetes is extensible; it allocates pods to nodes, taking the resource allocation on each node and the resource constraints of pods into account. Importantly, a **controller manager** and various controllers for different types of resources exist, which monitor different parts of the cluster in **reconciliation loops**, to make sure that the observed state matches the desired cluster state.

E.g., node failures are detected, followed by scheduling of affected pods towards other nodes. As another example, controllers for replication make sure that the desired number of healthy replicas exist.

Besides, an optional cloud controller manager provides bridging functionality to run Kubernetes on different cloud environments, each of which may come with its own specific API.

As a side remark, functionality of the control plane is running in the form of special-purpose pods, on one or more special-purpose nodes.

Returning to the right part, a **kubelet** per node manages the pods and containers on that node. It keeps track of the desired state as maintained in the control plane and performs pod management operations as necessary; e.g., it monitors the health of desired pods and restarts them if necessary. Finally, the **kube-proxy** is responsible for networking, including routing and load balancing.

3.3 Basic Concepts

- Node, pod, container, controller: Previous slide

This slide lists important Kubernetes concepts.

Pods are collections of containers and serve as units of scheduling, to be executed on nodes, as explained previously.

- Resources
  - Entities representing state
  - Selected examples (full documentation):
    - Namespace: Working area, separates different environments
    - Pod: Collection of containers, unit of scheduling
    - Service: Abstraction for exposing network application with one or more pods; think of load balancer
    - Deployment: API object managing pods (including replication)
    - PersistentVolume: Piece of (cloud) storage
    - PersistentVolumeClaim: Request to create PersistentVolume

In general, Kubernetes manages a variety of resources, which are entities representing state.

Namespaces provide working areas, separating different environments.

Pods as units of scheduling are important resources in Kubernetes.

A service exposes a network application consisting of one or more pods, to which the service forwards requests. Without a service, pods cannot be accessed from outside the cluster.

A deployment provides declarative updates for pods and their containers. It describes a desired state, e.g., the number of desired replicas. The deployment controller monitors the current state, changing it to the desired state if necessary.

A PersistentVolume provides an abstraction for pieces of storage, e.g., on a network filesystem or in a cloud storage.

With a PersistentVolumeClaim, a user can request storage of a specific size with a specific access mode.

Note that this list is just an excerpt. The hyperlink on the slides points to the full documentation.

4 K8s Examples

Let us see some examples for Kubernetes.

4.1 Minikube Installation

- Based on `https://learnk8s.io/deploying-nodejs-kubernetes`

1. Install kubectl
2. Install minikube
   - (Blog article points there for Windows)

- (Browser-based alternative at `https://labs.play-with-k8s.com/`)

Please experiment with Kubernetes on your own machine. The blog post hyperlinked here recommends `minikube` (and mentions alternatives). You also need `kubectl`.

Browser-based alternatives exist as well, but your instructor found them to be unreliable.

4.2 Create K8s Cluster with nginx

```
minikube start # Just one node; use options for more
kubectl cluster-info
kubectl get nodes
kubectl get pods -A # Pods of all namespaces; so far, control plane
kubectl apply -f https://gitlab.com/oer/cs/programming/-/raw/main/k8s/nginx-deployment.yaml # Add nginx
kubectl get pods -l run=my-nginx -o wide # Note names and IP addresses of pods
minikube ssh
curl <pod-ip-address> # Performs GET request to nginx in pod; shows HTML
exit
kubectl apply -f https://gitlab.com/oer/cs/programming/-/raw/main/k8s/nginx-service.yaml
minikube service nginx-service # Connect to nginx cluster
kubectl exec -it <pod-name-from-above> -- bash # Maybe change index.html of nginx
kubectl explain deployment
kubectl explain deployment.spec.selector
minikube delete --all
```

After the installation, experiment with a local cluster.

The first line shows how to create a cluster with a single (virtual) node. To create clusters with more nodes, command line options exist.

Lines 2 to 4 show commands to inspect the cluster. In particular, they show pods making up the control plane.

Line 5 creates a deployment from a YAML file, which is explained on the next slide. Briefly, that deployment describes a desired state with 3 replicas of pods running nginx.

Line 6 shows the 3 pods, including names and IP addresses. Verify in your output that each pod runs under a separate IP address, making the 3 nginx servers available under different IP addresses in the cluster. Without service, no external access is possible, though.

As side concept, lines 7 to 9 show how to access the cluster with secure shell. Note that `ssh` is a usual command for cryptographically secured shell access to remote computers.

Inside the cluster, tools such as `curl` can be used for HTTP requests. Here, `curl` retrieves the default HTML file of nginx and shows it on the command line.

Line 10 creates a service from a YAML specification to be explained subsequently. Briefly, the service acts as load balancer for the 3 pods.

Line 11 makes that service available with minikube and contacts it in the browser.

Again as side concept, line 12 shows how to execute a bash inside a pod. Then, you could change the served HTML file, for example.

What happens if you create different HTML files in different pods?

Finally, Kubernetes can explain concepts in detail.

Maybe delete the cluster as shown in line 15. Note that the explain command only works with a running cluster.

4.2.1 Sample Deployment

```
# SPDX-FileCopyrightText: 2024 Jens Lechtenbörger
# SPDX-License-Identifier: CC0-1.0

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 3
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: nginx-container
        image: nginx
        ports:
        - containerPort: 80
```

This is the nginx deployment with 3 replicas used on the previous slide.

YAML files generally contain configuration information with key-value pairs. After initial comments, line 4 specifies the API version, followed by the kind of the resource in line 5, here a deployment. Among metadata, line 7 assigns a name to the deployment.

The major part consists of the specification starting in line 8. According to line 12, 3 replicas are desired. Thus, this deployment is an example for horizontal scaling with replication.

Lines 9 to 11 define a selector, which is used to select pods for replication, here based on the label in line 11.

The template for the pod starting in line 13 defines the label of line 11 as label for the pods in line 16. Thus, the pods selected for replication are precisely the ones created from the template.

Finally, the specification for the template defines the containers to be created, here nginx containers listening on port 80.

4.2.2 Sample Service

```
# SPDX-FileCopyrightText: 2024 Jens Lechtenbörger
# SPDX-License-Identifier: CC0-1.0

apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    run: my-nginx
  ports:
    - port: 80
      targetPort: 80
  type: LoadBalancer
```

This is the specification of a load balancer service for our 3 pods.

Similarly to the YAML file for the deployment, lines at the beginning define the API version, kind, and name. Importantly, line 5 specifies the kind of the resource defined here to be a service.

The service specification starts in line 8. Note that the selector label in line 10 is precisely the template label of our nginx deployment. Thus, the service provides access to our nginx pods. According to line 12, the service provides access at port 80; as specified in line 13, it does so by internally accessing the target port 80 of the deployment.

Line 14 specifies this service to be a load balancer.

(Other types exist as well, see the service documentation if you are interested.)

4.3 Web App with Frontend and Backend

- Stateful vs stateless servers
  - **Stateless**: No local state, can just spawn new replicas
    - E.g., web server
    - Horizontal scaling with replicas

    Two major state models exist for servers in distributed systems, namely stateful and stateless ones.

    Stateless servers do not maintain state information between requests; every request is served independently of previous requests.

    A web server for a static web page is a typical example. Importantly, stateless servers can be scaled horizontally by creating replicated pods, as you saw previously.

  - **Stateful**: Maintain local state, need recovery in case of failures
    - E.g., database server, filesystem

    In contrast, stateful servers maintain state, which changes during operation. E.g., think of a web shop where inventory levels are updated in response to sales. With replicated pods running as isolated containers, each pod would maintain inventory levels independently of other pods. Clearly, this would not work.

    Instead, sales of all pods need to be reflected in one shared inventory level. In Kubernetes, persistent volumes can be used to specify such shared pieces of storage.

- In class: Scale knote
  - Note-taking app
    - Stateless frontend
    - Text in MongoDB, images in object storage MinIO; both with PersistentVolumeClaim

In class, we revisit the note-taking app knote with deployments and services. That application has a stateless frontend for user interaction, paired with a stateful backend that persistently stores textual notes in a database and images in an object store.

4.4 Self-Study

- Install minikube
- Create k8s cluster and experiment with it

Take a break to experiment with Kubernetes.

4.5 Aside: Meta's Hyperscale Infrastructure

- Video: `https://cacm.acm.org/videos/metas-hyperscale`
  - Paper: (Tang 2025)

Beyond class topics, if you are interested how Meta's global cloud infrastructure looks like as an example for a so-called hyperscaler, check out the video hyperlinked here or the cited paper.

5 Conclusions

Let us conclude.

5.1 Cloud Repatriation

- Cloud **repatriation**: Bring workload back from cloud
  - See examples in BBC article, June 2024
    - Details from 37signals, Oct 2024
- Reasons
  - Rising cloud bills (depending on use, millions of Euro per year)
    - Instead, invest in own infrastructure and personnel
    - See (Murugesan 2024) for examples and discussion
      - Michelin as different example in June 2025 (44% reduction of infrastructure costs, 85% reduction in software upgrade time)
  - Digital sovereignty
    - Public cloud is not "much easier" any more
      - Private cloud with Kubernetes or Portainer
    - Full control over private cloud
    - Security concerns, e.g., confidentiality of R&D data or proprietary code

Although cloud computing is highly popular, it comes with downsides that may cause cloud users to move their workloads back home from the cloud, which is called cloud repatriation. (In fact, those downsides may prevent others from using public cloud offerings in the first place.)

An article from June 2024 hyperlinked here provides examples.

A major reason for this change is high cloud fees. Depending on companies' cloud use, fees could just become too high. So, rather than spending this money on the cloud, companies may decide to build and manage their own technology infrastructure and teams. The research article cited here provides examples and a discussion.

In addition, Michelin offers a case study as of June 2025, hyperlinked on the slide: The company switched from a vendor-based K8s infrastructure to an open source one. Importantly, they appreciate not having to wait for the vendor to fix issues but to be in control themselves, reducing software upgrade leadtimes by 85%.

Indeed, besides financial considerations, gaining digital sovereignty is an important reason for shifting away from cloud infrastructures controlled by others. In the past, relying on external cloud services might have seemed easier, but today, switching to self-managed cloud solutions based on platforms like Kubernetes (or alternatives with graphical user interfaces such as Portainer) has become feasible. Clearly, a private cloud infrastructure offers full control, and it does neither require to share confidential data nor software with third parties.

5.2 Summary

- Distributed systems are everywhere, based on Internet

Distributed systems form the foundation of modern computing landscapes, with the Internet serving as a primary infrastructure facilitating interactions between interconnected devices worldwide. Through messaging, these machines coordinate activities and share resources efficiently.

- Cloud computing provides infrastructure for distributed systems
  - With different service models for different applications, potentially "serverless"

Cloud computing extends the reach of distributed systems by offering service models upon which developers can construct innovative applications, from simple file storage to arbitrarily complex functionality. Among cloud computing offerings, serverless computing emerges as a prominent paradigm, empowering users to create and deploy applications, with pay-per-use pricing, without worrying about operational concerns.

- Software architecture may contain numerous containers
  - Container orchestrator for management
    - K8s as dominant software solution
    - "Datacenter as a service", with declarative specification in YAML

When designing software architecture, it is common to include multiple containers within the system. The purpose of container orchestrators lies in the management of all these containers, and Kubernetes is the dominant solution in industry. With its help, developers create clusters containing both a control plane and worker nodes, which facilitate efficient coordination among different components.

The configuration and behavior of parts of Kubernetes clusters are often described through declarative definitions written in YAML files. These descriptions enable reconciliation loops performed by controllers responsible for ensuring desired states align with actual conditions. Additionally, pods serve as fundamental units of scheduling, and services provide essential networking capabilities.

- Cloud repatriation

- Migration of cloud workloads "back home", potentially with K8s

Cloud repatriation refers to the process of migrating workloads previously hosted on cloud servers back to local, on-premises infrastructure. Two primary motivators behind this shift are cost reduction and digital sovereignty. Notably, Kubernetes can help as infrastructure for cloud repatriation as its mechanisms and functionalities stay the same, regardless of whether businesses operate primarily in remote datacenters or locally managed facilities.

Bibliography

Casalicchio, Emiliano. 2019. "Container Orchestration: A Survey." In *Systems Modeling: Methodologies and Tools*, edited by Antonio Puliafito and Kishor S. Trivedi, 221–35. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-92378-9_14.

Murugesan, Ganesh Kumar. 2024. "Cloud Services — Boon or Bane: A Comprehensive Review." In *Southeastcon 2024*, 108–12. https://doi.org/10.1109/SoutheastCon52093.2024.10500027.

Tang, Chunqiang. 2025. "Meta's Hyperscale Infrastructure: Overview and Insights." *Commun. Acm* 68 (2): 52–63. https://doi.org/10.1145/3701296.

The bibliography contains references used in this presentation.

License Information