

Diese Vorlage nutzt ein zweispaltiges Layout. Ersetzen Sie die Platzhalter, fügen Sie Screenshots unter **screenshots/** hinzu und beschreiben Sie die einzelnen Schritte kurz und prägnant.

AWS Docker Swarm Workshop – Dokumentation

Kurs / Modul: <eintragen>

Autor: <Name> Matrikel: <Nr.>

Datum: 14. Oktober 2025

Hinweis

Diese Vorlage nutzt ein zweispaltiges Layout. Ersetzen Sie die Platzhalter, fügen Sie Screenshots unter `screenshots/` hinzu und beschreiben Sie die einzelnen Schritte kurz und prägnant.

Inhaltsverzeichnis

1 Überblick	2
2 Provisionierung der AWS-Ressourcen	2
2.1 VPC und Subnet	2
3 EC2-Instanzen (Nodes)	2
3.1 Validierung	3
4 Docker Swarm Setup	3
5 Portainer Installation	4
6 Service-Deployment, Scale-Out und Scale-In	4
7 Kurzantworten	6
8 Anhang – Wichtige Befehle	6
9 Kubernetes Deployment Workshop mit Minikube	6
9.1 Start und Vorbereitung	6
9.2 Deployment erstellen und skalieren	7
9.3 Testzugriff auf einen Pod	7
9.4 ClusterIP Service (intern)	7
9.5 NodePort Service (extern)	7
9.6 LoadBalancer Service (simuliert)	8
9.7 Vergleich der Service-Typen	8
9.8 YAML-Spezifikationen	8
9.9 Zusammenfassung	9

1 Überblick

Ziel: Provisionierung von AWS-Ressourcen, Aufbau eines Docker Swarm-Clusters, Installation von Portainer und Analyse von Skalierung und Ausfallszenarien.

2 Provisionierung der AWS-Ressourcen

2.1 VPC und Subnet

Neue VPC mit öffentlichem Subnet über den Assistenten im AWS-VPC-Dashboard erstellen. Security Group mit folgenden offenen Ports konfigurieren:

- 22/TCP – SSH (öffentlich)
- 80/TCP – HTTP (öffentlich)
- 2377/TCP, 7946/TCP/UDP, 4789/UDP – innerhalb der SG

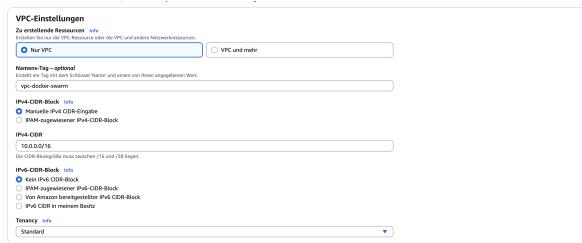


Abbildung 1: Einstellung des VPCs inklusive manueller CIDR Einstellung 10.0.0.0/16

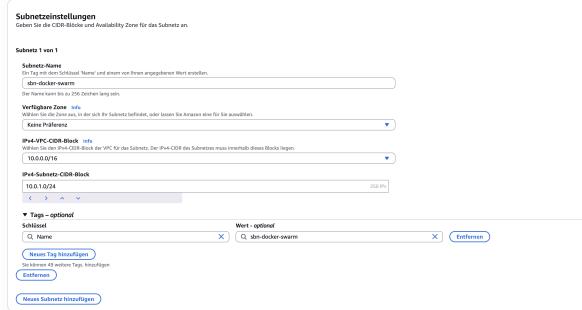


Abbildung 2: Öffentliches Subnet mit Subnet CIDR-Block 10.0.1.0/24



Abbildung 3: Automatische Vergabe von öffentlicher ipv4 Adresse aktiviert

Die Sicherheitsgruppe musste in zwei Schritte erstellt werden, dass sie den Vorgaben entspricht.

Schritt 1:

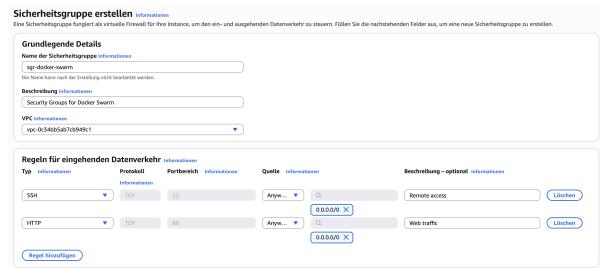


Abbildung 4: Erstellung der Sicherheitsgruppe mit SSH und HTTP

Schritt 2: Erst nach dem die Sicherheitsgruppe erstellt wurde konnten wir für die Docker regeln die eigene Sicherheitsgruppe als Quelle angeben.

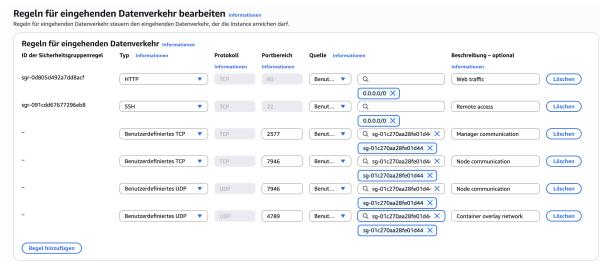


Abbildung 5: Anpassung der vorher erstellten Sicherheitsgruppe mit zusätzlichen Regeln für Docker

3 EC2-Instanzen (Nodes)

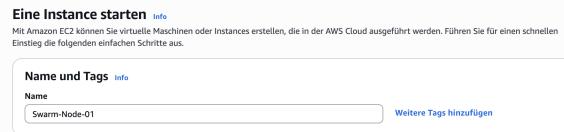


Abbildung 6: Erstellung der ersten Instanz in AWS

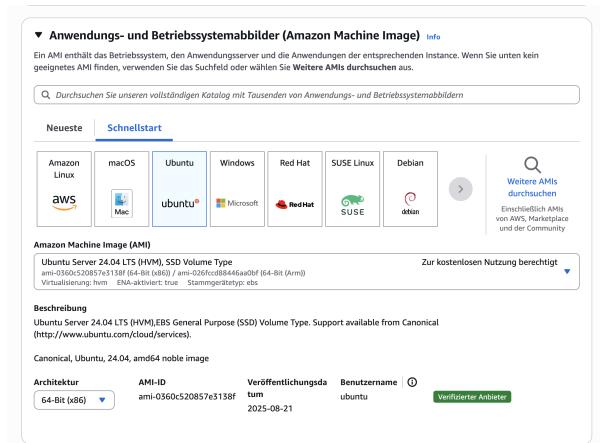


Abbildung 7: AMI: Ubuntu (neueste Version).

Abbildung 8: Micro als Instance Type

Abbildung 9: Aktivierung von Auto-assign Public IP.

Abbildung 10: Einfügung von User-Data-Skript

```
#!/bin/bash
curl -o /home/ubuntu/install-docker.sh https://get.docker.com/
sh /home/ubuntu/install-docker.sh
```

Listing 1: EC2 User Data – Docker Installation

Abbildung 11: EC2 Launch – Übersicht der Einstellungen

3.1 Validierung

```
ssh -i <key.pem> ubuntu@<PUBLIC_IP>
docker --version
```

Abbildung 12: SSH-Verbindung und Docker-Version

4 Docker Swarm Setup

```
docker swarm init --advertise-addr <PRIVATE_IP>
docker node ls
```

Abbildung 13: docker swarm init auf der ersten Instanz

```
ubuntu@ip-10-0-1-135:~$ sudo docker node ls
ID          HOSTNAME   STATUS  AVAILABILITY  MANAGER STATUS ENGINE VERSION
1wtvfldegzatys0akxhe386z  ip-10-0-1-135 Ready   Active        Leader      28.5.1
```

Abbildung 14: docker swarm init auf der ersten Instanz

```
ubuntu@ip-10-0-1-46:~$ sudo docker swarm join --token SWMTKN-1-5z9kkz027ot0npy0dhah6ulz1xum6gyw889e517bzlxhm6awo-8o3x4yyj7u1vifyspdrcghif 10.0.1.135:2377
This node joined a swarm as a worker.
ubuntu@ip-10-0-1-32:~$ sudo docker swarm join --token SWMTKN-1-5z9kkz027ot0npy0dhah6ulz1xum6gyw889e517bzlxhm6awo-8o3x4yyj7u1vifyspdrcghif 10.0.1.135:2377
This node joined a swarm as a worker.
```

Abbildung 15: docker swarm Hinzufügen von zwei Worker

```
ubuntu@ip-10-0-1-135:~$ sudo docker swarm join-token manager
To add a manager to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-5z9k
    kzo27ot0npy0dha6u1z1xum6gymw889e5l7bzlxhmn6
    awo-4ues15ehcaoqur9v0zjp2pvf5 10.0.1.135:23
    77
```

Abbildung 16: generieren von Manager-tokens

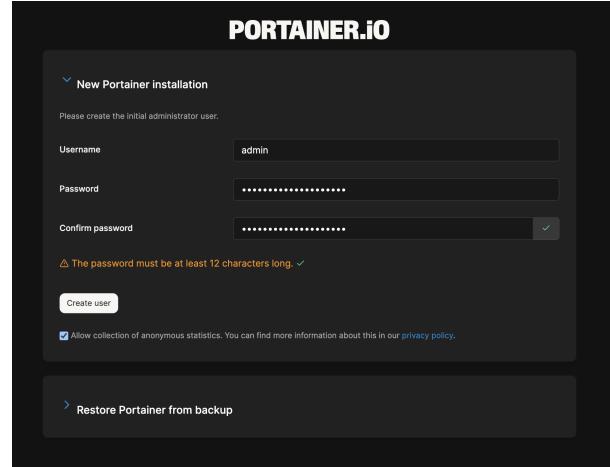


Abbildung 20: Erstellung eines neuen Logins für Portainer Installation

```
ubuntu@ip-10-0-1-62:~$ sudo docker swarm join --token SWMTKN-1-5z9kkzo27ot0npy0dha6u1z1xum6gymw889e5l7bzlxhmn6awo-4ues15ehcaoqur9v0zjp2pvf5 10.0.1.135:2377
This node joined a swarm as a manager.
```

Abbildung 17: Hinzufügen von zweiten Manager

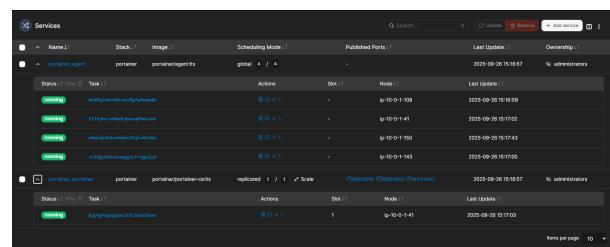


Abbildung 21: Angezeigte Services nach der Installation von Portainer

```
ubuntu@ip-10-0-1-135:~$ sudo docker node ls
ID           HOSTNAME   STATUS  AVAILABILITY  MANAGER STATUS      ENGINE VERSION
Cm07tp1xrnx6dy883t9ejya  ip-10-0-1-32  Ready   Active        28.5.1
knfduwfaf01ttfpus5hsh8p002 ip-10-0-1-46  Ready   Active        28.5.1
saejhubo0rjeau6ap1qlq4mvz ip-10-0-1-62  Ready   Active        Leader      28.5.1
iwtvfldegzatays0akxhe386z * ip-10-0-1-135 Ready   Active        Leader      28.5.1
```

Abbildung 18: docker node ls mit 4 Knoten

5 Portainer Installation

```
curl -L https://downloads.portainer.io/ce-lts/portainer-agent-stack.yml -o portainer-agent-stack.yml

docker stack deploy -c portainer-agent-stack.yml portainer
```

```
ubuntu@ip-10-0-1-62:~$ sudo su
root@ip-10-0-1-62:/home/ubuntu# docker node ls
ID           HOSTNAME   STATUS  AVAILABILITY  MANAGER STATUS      ENGINE VERSION
Cm07tp1xrnx6dy883t9ejya  ip-10-0-1-32  Ready   Active        28.5.1
knfduwfaf01ttfpus5hsh8p002 ip-10-0-1-46  Ready   Active        28.5.1
saejhubo0rjeau6ap1qlq4mvz * ip-10-0-1-62  Ready   Active        Leader      28.5.1
iwtvfldegzatays0akxhe386z ip-10-0-1-135 Ready   Active        Reachable   28.5.1
root@ip-10-0-1-62:/home/ubuntu# curl -L https://downloads.portainer.io/ce-lts/portainer-agent-stack.yml -o portainer-agent-stack.yml
  % Total    % Received % Xferd  Average Speed   Time  Time Current
  100  795  100  795  0     0       0      0 --:--:-- --:--:-- --:--:-- 100
root@ip-10-0-1-62:/home/ubuntu# docker stack deploy -c portainer-agent-stack.yml portainer
Since --detach=false was not specified, tasks will be created in the background.
To start them in detached mode, use --detach=true.
Creating network portainer_agent_network
Creating service portainer_portainer
Creating service portainer_agent
root@ip-10-0-1-62:/home/ubuntu#
```

Abbildung 19: Portainer Installation über Kommandozeile

6 Service-Deployment, Scale-Out und Scale-In

```
docker service create \
  --name hello \
  --replicas 3 \
  --publish published=80,target=80 \
  karthequian/helloworld:latest
```

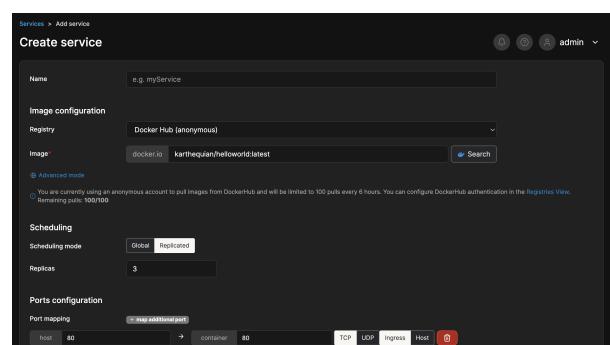


Abbildung 22: Service mit 3 Replicas und Port 80 erstellen



Abbildung 23: Die 3 Replicas erhalten alle einen eigenen Slot



Abbildung 27: Die 10 Replicas werden auf die 4 verschiedenen Nodes auf dem AWS verteilt



Abbildung 24: Die 3 Replicas werden auf die 4 verschiedenen Nodes auf dem AWS verteilt

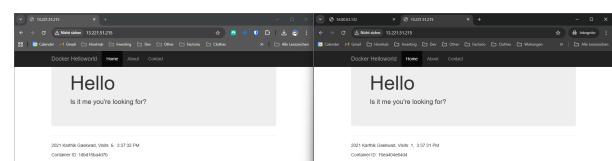


Abbildung 28: Hello World Seite wird nun in verschiedenen Containern angezeigt

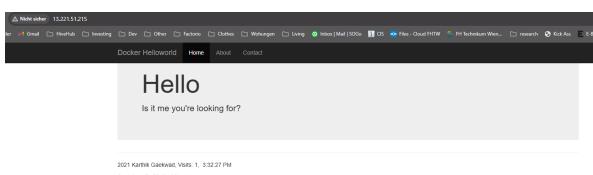


Abbildung 25: Hello World Seite nach deploy des eben erstellen Service

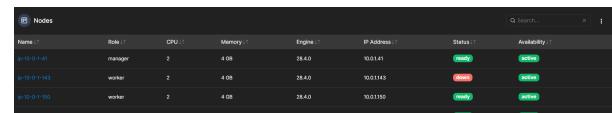


Abbildung 29: Node wird als "downängzeigt, nachdem eine EC2 heruntergefahren wurde.

```
docker service scale hello=10
```

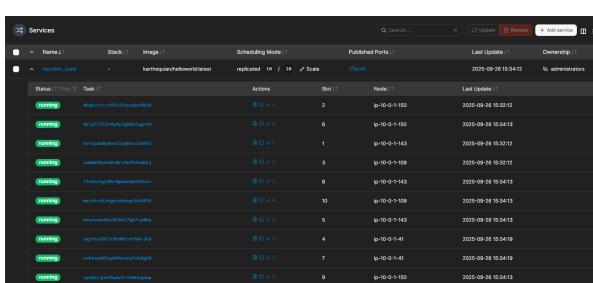


Abbildung 26: Die 10 Replicas werden nach dem Hochskallieren ebenfalls auf verschiedene Slots verteilt

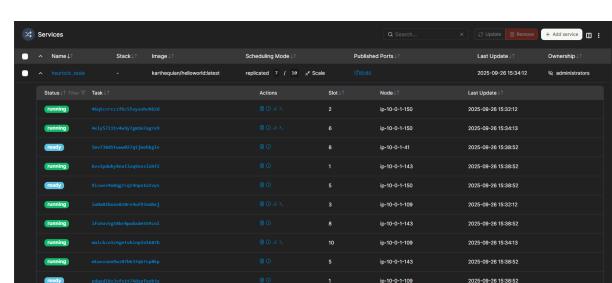


Abbildung 30: Die services, welche auf diesem Node laufen haben in den Status "ready" gewechselt. Nach einer kurzen Zeit waren sie im Status **blutdown** und neue Services waren auf den anderen Nodes verteilt.

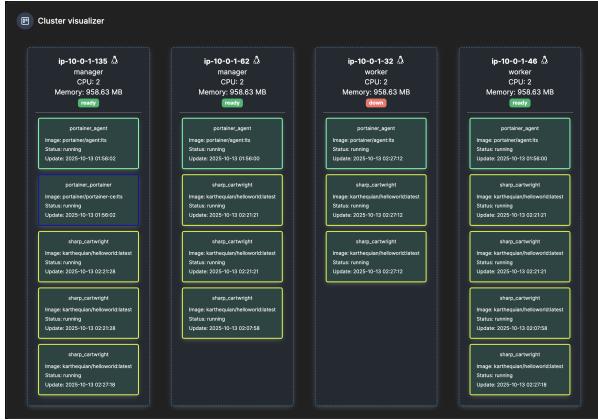


Abbildung 31: Gut zu sehen, dass der Node down ist mit seinen zwei Services aber auf den restlichen 3 Nodes insgesamt 10 services laufen.

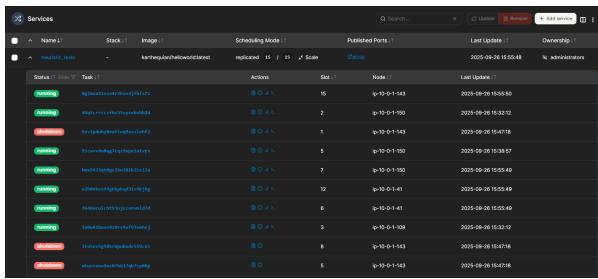


Abbildung 32: Nach dem Hochfahren des Nodes, waren in unserem fall die herunter gefahrener Nodes immer noch heruntergefahren. Dafür wurde aber die restlichen Services wieder sauber auf alle 4 Nodes verteilt.



Abbildung 33: Beim Scale Down der Services wurden die restlichen 3 Services auf den 4 Nodes wieder sauber verteilt. Dafür sind die vorhin heruntergefahrenen Services wie Zombies auf dem Node zurückgeblieben.

7 Kurzantworten

- Skalierung des Clusters:** Skalieren Sie Ihren Cluster von 3 auf 10 Maschinen. *Beobachtung:* Beim Öffnen der Hello-World-Seite wird jeweils ein anderer Container angezeigt.

- Wiederinbetriebnahme eines Workers:** Wenn Sie den Worker wieder online schalten, werden die Services automatisch auf alle vier Nodes verteilt.

- Erneute Skalierung:** Bei einer erneuten Skalierung werden die drei Services erneut gleichmäßig auf die vier Nodes verteilt.

8 Anhang – Wichtige Befehle

```
docker service ls
docker service ps hello
docker node ls
docker service scale hello=10
```

9 Kubernetes Deployment Workshop mit Minikube

Ziel: In diesem Workshop wurde ein Kubernetes-Cluster mit **Minikube** erstellt, ein **nginx**-Deployment angelegt und verschiedene Service-Typen (ClusterIP, NodePort, LoadBalancer) getestet.

9.1 Start und Vorbereitung

```
minikube start
kubectl get nodes
```

Listing 2: Start von Minikube und Überprüfung der Umgebung



Abbildung 34: Start von Minikube mit einer einzelnen Node (Control Plane)

```
kubectl cluster-info
kubectl get all
```

Listing 3: Überprüfung des Cluster-Status



Abbildung 35: Minikube Clusterinformationen im Terminal

9.2 Deployment erstellen und skalieren

```
kubectl create deployment nginx --image=nginx:  
    stable  
kubectl get deployments  
kubectl get pods
```

Listing 4: nginx Deployment mit einem Pod erstellen



Abbildung 36: Erstellung des nginx Deployments

```
kubectl scale deployment nginx --replicas=4  
kubectl get pods -o wide
```

Listing 5: Deployment auf vier Replikas skalieren



Abbildung 37: Vier Replikas des nginx Deployments sind aktiv

9.3 Testzugriff auf einen Pod

```
kubectl port-forward deployment/nginx 8080:80
```

Listing 6: Port-Forwarding für lokalen Zugriff

Im Browser: <http://localhost:8080>



Abbildung 38: Zugriff auf nginx über Port 8080 via Port-Forwarding

9.4 ClusterIP Service (intern)

```
kubectl expose deployment nginx \  
  --name=nginx-clusterip \  
  --type=ClusterIP \  
  --port=8080 \  
  --target-port=80
```

```
--port=8080 \  
--target-port=80  
kubectl get svc nginx-clusterip
```

Listing 7: Erstellung eines internen ClusterIP-Service



Abbildung 39: Interner ClusterIP-Service für nginx Deployment

```
kubectl run test-client --rm -it --image=  
curlimages/curl --restart=Never -- sh  
curl -I http://nginx-clusterip:8080
```

Listing 8: Test im Cluster über temporären Curl-Pod



Abbildung 40: Erfolgreicher Curl-Test des internen ClusterIP-Service

9.5 NodePort Service (extern)

```
kubectl expose deployment nginx \  
  --name=nginx-nodeport \  
  --type=NodePort \  
  --port=8080 \  
  --target-port=80  
kubectl get svc nginx-nodeport
```

Listing 9: Erstellung eines NodePort-Service

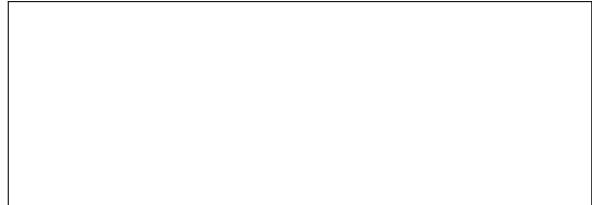


Abbildung 41: NodePort-Service mit zugewiesinem externen Port (z. B. 31245)

```
minikube service nginx-nodeport --url
```

Listing 10: Abrufen der externen URL über Minikube

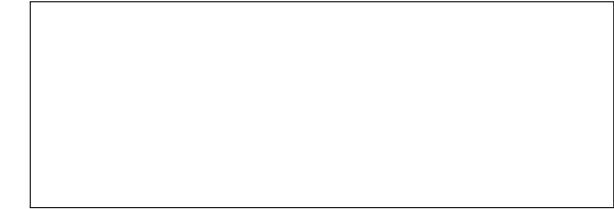


Abbildung 42: Von Minikube generierte URL für NodePort-Zugriff

Zugriff im Browser oder mit curl:

```
curl -I $(minikube service nginx-nodeport --url)
```

9.6 LoadBalancer Service (simuliert)

```
kubectl expose deployment nginx \
--name=nginx-lb \
--type=LoadBalancer \
--port=8080 \
--target-port=80
```

Listing 11: Erstellung eines LoadBalancer-Service

Da Minikube keinen echten Cloud Load Balancer bereitstellt, wird die Funktion über einen lokalen Tunnel simuliert.

```
minikube tunnel
```

Listing 12: Start des Minikube Tunnels (neues Terminal)

Abbildung 44: LoadBalancer mit zugewiesener externer IP-Adresse

```
curl -I http://<EXTERNAL-IP>:8080
```

Listing 13: Zugriff über die externe IP



Abbildung 45: Erfolgreicher Zugriff über den LoadBalancer-Service

9.7 Vergleich der Service-Typen



Abbildung 46: Übersicht der drei Service-Typen im Kubernetes Dashboard

9.8 YAML-Spezifikationen

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  replicas: 4
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:stable
```

Abbildung 43: Simulierter LoadBalancer über Minikube Tunnel

Prüfen der externen IP:

```
kubectl get svc nginx-lb
```

Typ	Erreichbarkeit	Einsatzgebiet / Beschreibung
ClusterIP	Nur innerhalb des Clusters	Standardtyp; ermöglicht Kommunikation zwischen Pods oder internen Diensten.
NodePort	Von außen über Node-IP und festen Port erreichbar	Zugriff über <NodeIP>:<NodePort> (z. B. für Tests).
LoadBalancer	Öffentliche IP / Cloud Load Balancer	Für produktive Setups; verteilt Traffic über alle Replikas.

Tabelle 1: Kubernetes Service-Typen

```
ports:
  - containerPort: 80
```

Listing 14: Deployment YAML – nginx

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-lb
  labels:
    app: nginx
spec:
  type: LoadBalancer
  selector:
    app: nginx
  ports:
    - name: http
      port: 8080
      targetPort: 80
```

Listing 15: Service YAML – LoadBalancer

- **Fazit:** Der Workshop verdeutlicht, wie Deployments und Services in Kubernetes strukturiert sind und welche Service-Typen für interne bzw. externe Zugriffe geeignet sind.



Abbildung 47: Erfolgreiche Anwendung der YAML-Dateien mit `kubectl apply -f`

9.9 Zusammenfassung

- **Cluster:** Ein lokales Kubernetes-Cluster mit Minikube wurde gestartet.
- **Deployment:** Ein nginx Deployment mit vier Replikas wurde erstellt.
- **Service-Typen:** ClusterIP, NodePort und LoadBalancer wurden erfolgreich getestet.
- **Zugriff:** Über Port-Forwarding, NodePort und Minikube Tunnel konnte der Webserver erreicht werden.