Agenda

Backlog

17. Kubernetes coil (egress - gateway)

    - coil

18. Kubernetes NetworkPolicy

    - Einfache Übung Network Policy

19. Calico NetworkPolicy

    - Protecting Services
    - Calico Logging Firewall Rules
    - Exercise calico Network Policy

20. Kubernetes Scheduler

    - Koordinator scheduler, that also network bandwith into account
    - Why is CPU pinning not used as default in kubernetes scheduler anymore ?

21. Helm (Kubernetes Paketmanager)

    - Helm Grundlagen
    - Helm Warum ?
    - Helm Example

22. Kubernetes - RBAC

    - Nutzer einrichten microk8s ab kubernetes 1.25
    - Besser: Nutzer einrichten mit Zertifikat

23. Kubernetes Advanced

    - Curl api-server kubernetes aus pod heraus

24. Kubernetes Load Balancer / metallb (on premise)

    - Kubernetes Load Balancer

25. Kubernetes API Reference

    - API-Reference
    - OpenAPI
    - Swagger Editor Online

26. Kubernetes Istio

    - Why istio and benefits
    - Overview sidecar pattern
    - Ambient Mode istio
    - Istio vs. Ingress Überblick
    - Istio installieren und Addons bereitsstellen
    - Istion Überblick - egress und ingress - gateway
    - Istio - Deployment of simple application
    - Istio - Grafana Dashboard
    - Mesh / istio
    - Benchmarks istio ambient,sidecar und baseline

27. Kubernetes Deployment Scenarios

    - Deployment green/blue,canary,rolling update
    - Service Blue/Green
    - Praxis-Übung A/B Deployment

28. Kubernetes - Hilfreiche Tools / Networking /Debugging

    - NetworkPolicyEditor
    - Debug on node with root-priviliges with kubectl debug

29. Kubernetes Security

    - NodeShell vs. kubectl debug (for node)
    - Kube Api Server - Watch Internals

Backlog

1. Docker Überblick

    - Docker Aufbau
    - Was sind Container
    - Was sind container images
    - Was ist ein Dockerfile

2. Kubernetes - Überblick

    - Installation - Welche Komponenten from scratch

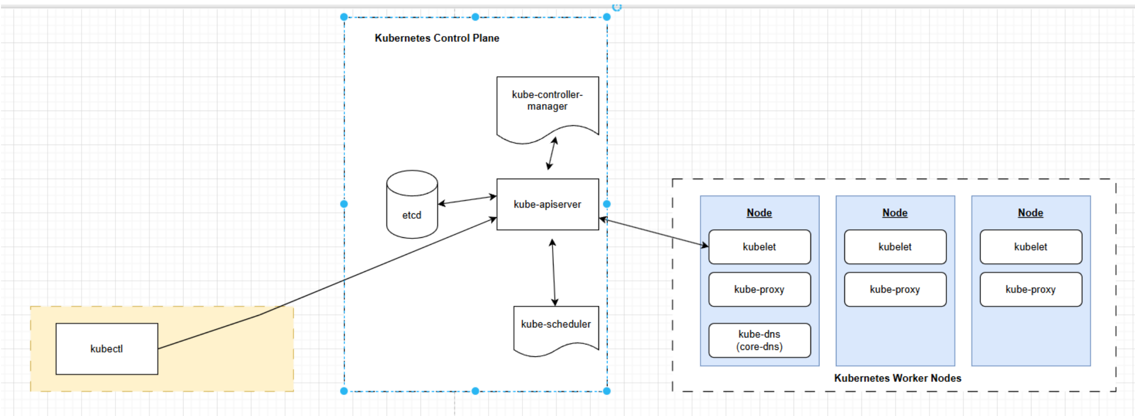
3. Kubernetes - microk8s (Installation und Management)

    - kubectl unter windows - Remote-Verbindung zu Kuberenets (microk8s) einrichten

Kubernetes Grundlagen

Aufbau Allgemein

Schaubild



Komponenten / Grundbegriffe

Master (Control Plane)

Aufgaben

- Der Master koordiniert den Cluster
- Der Master koordiniert alle Aktivitäten in Ihrem Cluster
  - Planen von Anwendungen
  - Verwalten des gewünschten Status der Anwendungen
  - Skalieren von Anwendungen
  - Rollout neuer Updates.

Komponenten des Masters

ETCD

- Verwalten der Konfiguration des Clusters (key/value - pairs) und des Status

KUBE-CONTROLLER-MANAGER

- Zuständig für die Überwachung der Stati im Cluster mit Hilfe von endlos loops.
- kommuniziert mit dem Cluster über die kubernetes-api (bereitgestellt vom kube-api-server)

KUBE-API-SERVER

- provides api-frontend for administration (no gui)
- Exposes an HTTP API (users, parts of the cluster and external components communicate with it)
- REST API

KUBE-SCHEDULER

- assigns Pods to Nodes.
- scheduler determines which Nodes are valid placements for each Pod in the scheduling queue (according to constraints and available resources)
- The scheduler then ranks each valid Node and binds the Pod to a suitable Node.
- Reference implementation (other schedulers can be used)

Nodes

- Worker Nodes (Knoten) sind die Arbeiter (Maschinen), die Anwendungen ausführen
- Ref: https://kubernetes.io/de/docs/concepts/architecture/nodes/

Pod/Pods

- Pods sind die kleinsten einsetzbaren Einheiten, die in Kubernetes erstellt und verwaltet werden können.
- Ein Pod (übersetzt Gruppe) ist eine Gruppe von einem oder mehreren Containern
  - gemeinsam genutzter Speicher- und Netzwerkressourcen
  - Befinden sich immer auf dem gleich virtuellen Server

Control Plane (former: master node) - components

Worker Node - components

General

- On the nodes we will rollout the applications

kubelet

```
Node Agent that runs on every node (worker)
Er stellt sicher, dass Container in einem Pod ausgeführt werden.
```

Kube-proxy

- Läuft auf jedem Node

- = Netzwerk-Proxy für die Kubernetes-Netzwerk-Services.
- Kube-proxy verwaltet die Netzwerkkommunikation innerhalb oder außerhalb Ihres Clusters.

Referenzen

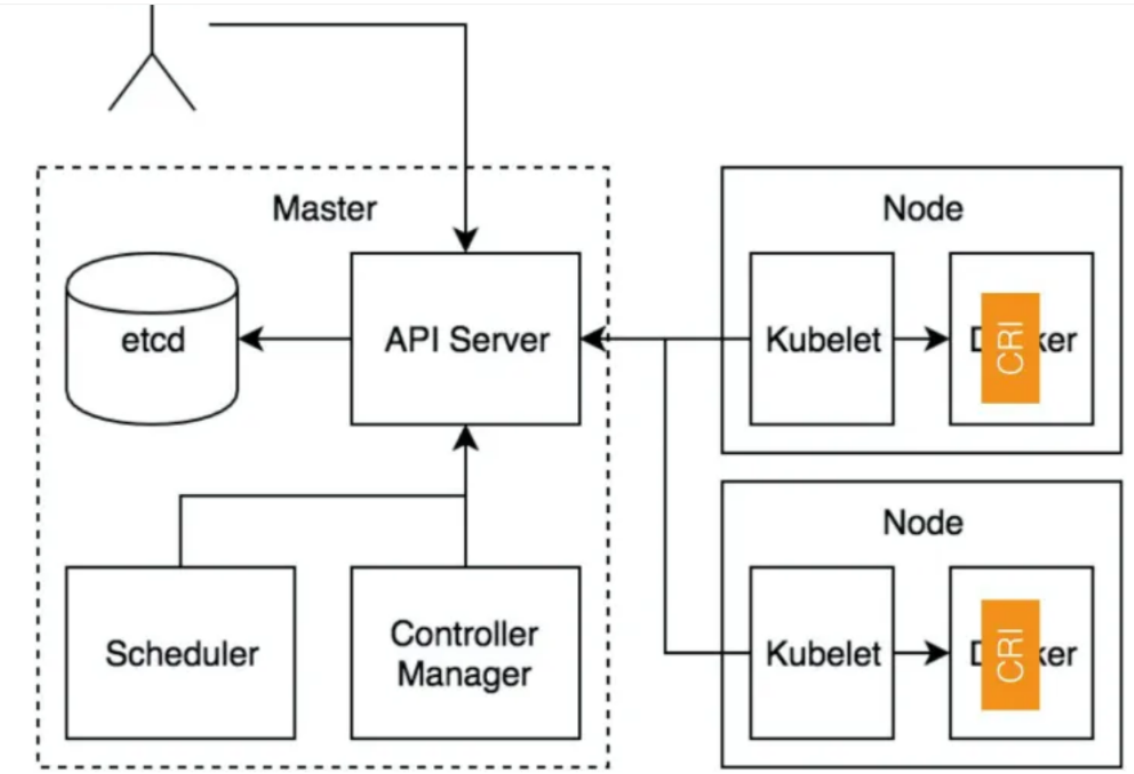
- https://www.redhat.com/de/topics/containers/kubernetes-architecture

Structure Kubernetes Deep Dive

- https://github.com/jmetzger/training-kubernetes-advanced/assets/1933318/1ca0d174-f354-43b2-81cc-67af8498b56c

CRI - Container Runtime interface

Where is it embedded



What is it for ?

- Abstraction layer called by kubelet to make it possible to use other container runtimes
- The CRI uses gRPC as its communication protocol.

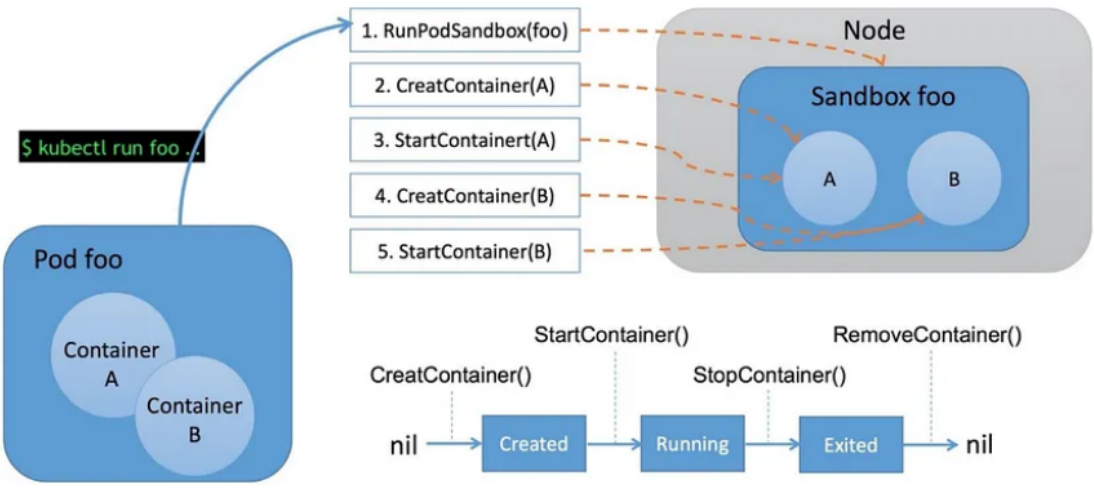
kubelet calls the CRI with its subcommands

- Expected commands are

```
Sandbox:
  Delete
  Create
  List
Image:
  Pull
  List
Container.
  Create
  Start
  Exec
```

Steps in the CRI

Container Lifecycle Management Through the CRI



Ports und Protokolle

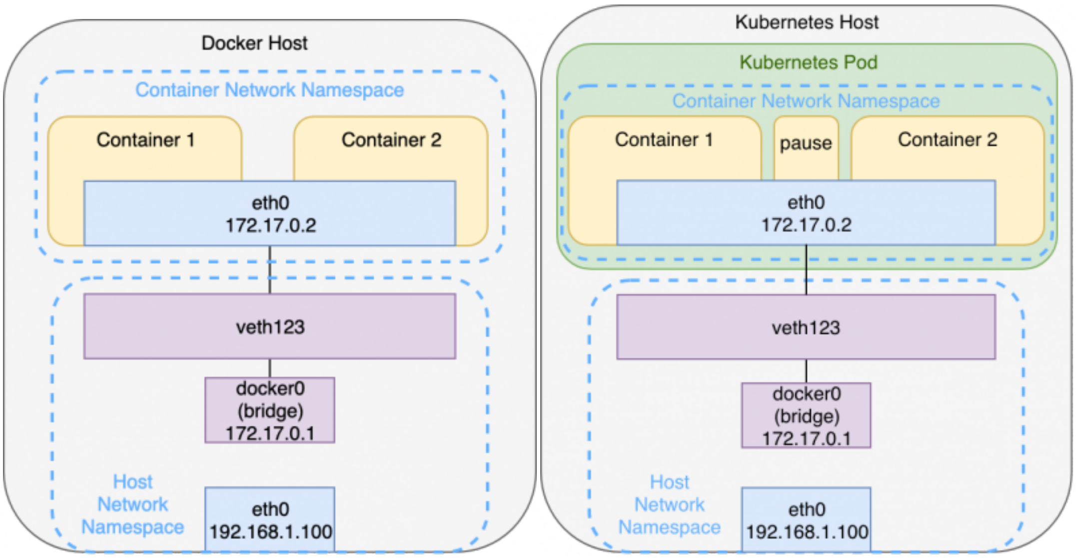
- https://kubernetes.io/docs/reference/networking/ports-and-protocols/

Kubernetes Networking Internals

Networking Internal Overview

Network Namespace for each pod

Overview



General

- Each pod will have its own network namespace
  - with routing, networkdevices
- Connection to default namespace to host is done through veth - Link to bridge on host network
  - similar like on docker to docker0

```
 Each container is connected to the bridge via a veth-pair. This interface pair functions like a virtual point-to-point ethernet
connection and connects the network namespaces of the containers with the network namespace of the host
```

- Every container is in the same Network Namespace, so they can communicate through localhost

- Example with hashicorp/http-echo container 1 and busybox container 2

Pod-To-Pod Communication (across nodes)

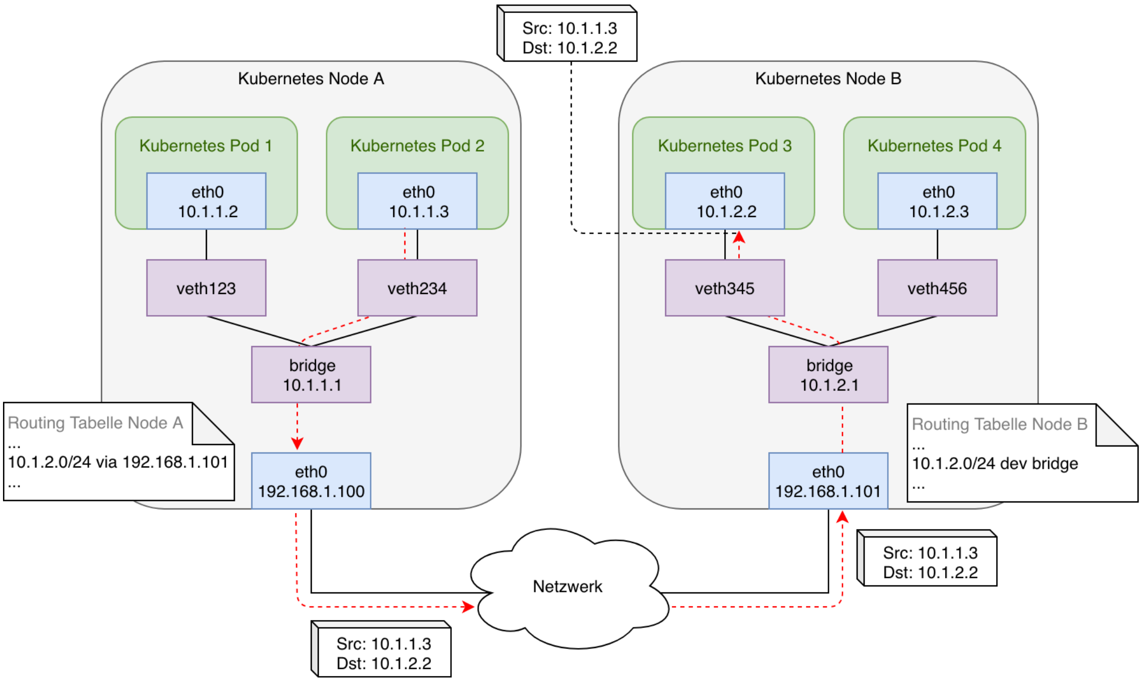
Prerequisites

- pods on a single node as well as pods on a topological remote can establish communication at all times
- Each pod receives a unique IP address, valid anywhere in the cluster. Kubernetes requires this address to not be subject to network address translation (NAT)
- Pods on the same node through virtual bridge (see image above)

General (what needs to be done) - and could be done manually

- local bridge networks of all nodes need to be connected
- there needs to be an IPAM (IP-Address Managemenet) so addresses are only used once
- The need to be routes so, that each bridge can communicate with the bridge on the other network
- Plus: There needs to be a rule for incoming network
- Also: A tunnel needs to be set up to the outside world.

General - Pod-to-Pod Communication (across nodes) - what would need to be done



General - Pod-to-Pod Communication (side-note)

- This could of cause be done manually, but it is too complex
- So Kubernetes has created an Interface, which is well defined
  - The interface is called CNI (common network interface)
  - Funtionally is achieved through Network Plugin (which use this interface)
    - e.g. calico / cilium / weave net / flannel

CNI

- CNI only handles network connectivity of container and the cleanup of allocated resources (i.e. IP addresses) after containers have been deleted (garbage collection) and therefore is lightweight and quite easy to implement.
- There are some basic libraries within CNI which do some basic stuff.

Hidden Pause Container

What is for ?

- Holds the network - namespace for the pod
- Gets started first and falls asleep later
- Will still be there, when the other containers die

```
cd
mkdir -p manifests
cd manifests
mkdir pausetest
cd pausetest
nano 01-nginx.yml
```

```
## vi nginx-static.yml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pausetest
  labels:
    webserver: nginx:1.21
spec:
  containers:
  - name: web
    image: nginx
```

```
kubectl apply -f .
## als root auf dem worker node
ctr -n k8s.io c list | grep pause
```

References

- https://www.inovex.de/de/blog/kubernetes-networking-part-1-en/
- https://www.inovex.de/de/blog/kubernetes-networking-2-calico-cilium-weavenet/

Kubernetes Einrichtung

kubectl verbindung mit namespace einrichten

config einrichten

```
cd
mkdir -p .kube
cd .kube
cp -a /tmp/config config
ls -la
## das bekommt ihr aus Eurem Cluster Management Tool
```

```
kubectl cluster-info
```

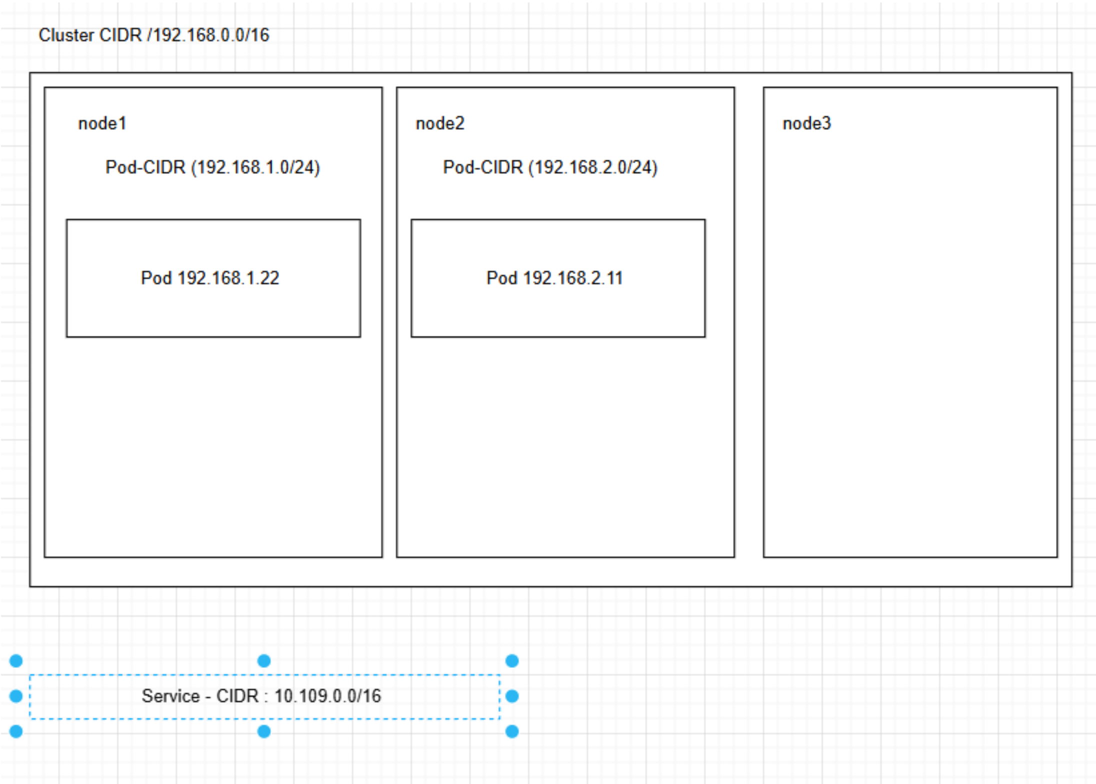
Arbeitsbereich konfigurieren

```
kubectl create ns jochen
kubectl get ns
kubectl config set-context --current --namespace jochen
kubectl get pods
```

Kubernetes Networking (Grundlagen)

Cluster-CIDR, POD-CIDR und Service-CIDR

Grafik



Cluster CIDR - IP-Bereich für das gesamte Kubernetes Cluster

```
## Netzbereich für mein gesamtes Cluster
10.244.0.0/16
```

POD-CIDR - Teilbereich aus der Cluster - CIDR pro Node

```
## Jede Node bekommt ein Teilnetz
Beispiel cilium

node 1 -> network.cilium.io/ipv4-pod-cidr: 10.244.0.0/25
node 2 -> network.cilium.io/ipv4-pod-cidr: 10.244.0.128/25
node 3 -> network.cilium.io/ipv4-pod-cidr: 10.244.1.128/25
node 4 -> network.cilium.io/ipv4-pod-cidr: 10.244.1.0/25
```

POD-IP

- Wird aus POD-CIDR des jeweiligen Nodes vergeben

```
## pod bekommt aus netzbereich POD-CIDR auf Node eine IP-Adresse zugewiesen
## CILIUM CNI macht das z.B.
POD-CIDR: 10.244.1.128/25
-> POD - IP: 10.244.1.180
```

Service-CIDR

```
Netzbereich für IP-Adressen der Services
z.B. 10.109.0.0/16
```

Wann wird podIP vergeben ?

Example (that does work)

```
## Show the pods that are running
kubectl get pods

## Synopsis (most simplistic example
## kubectl run NAME --image=IMAGE_EG_FROM_DOCKER
## example
kubectl run nginx --image=nginx:1.23
```

```
kubectl get pods
## on which node does it run ?
kubectl get pods -o wide
```

Example (that does not work)

```
kubectl run foo2 --image=foo2
## ImageErrPull - Image konnte nicht geladen werden
kubectl get pods
## Weitere status - info
kubectl describe pods foo2
```

Ref:

- https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#run

Kubernetes Netzwerk einrichten

CNI-Provider calico einrichten

Walkthrough

```
## Step 1 - Install the operator
kubectl create -f https://raw.githubusercontent.com/projectcalico/calico/v3.30.1/manifests/tigera-operator.yaml
## Step 2 - Install the custom resources
kubectl create -f https://raw.githubusercontent.com/projectcalico/calico/v3.30.1/manifests/custom-resources.yaml
```

Testing

```
kubectl -n tigera-operator get pods
kubectl -n calico-system get pods
kubectl -n calico-apiserver get pods
## Sind die nodes schon ready
kubectl get nodes

kubectl -n kube-system get pods
kubectl -n kube-system get pods coredns-7c65d6cfc9-f6f56
kubectl -n kube-system describe pods coredns-7c65d6cfc9-f6f56
```

Reference

- https://docs.tigera.io/calico/latest/getting-started/kubernetes/quickstart

Andere Cluster-CIDR verwenden - calico/kubeadm

Voraussetzungen:

- Alle Nodes sind bereits mit os (z.B. Ubuntu 24.04.) und installierten Kernkomponenten versehen

```
kubelet
kubeadm
containerd
```

Schritt 1: Nur auf control plane notwendig (d.h. init)

```
## Initalisiert das control plane mit anderer cluster cidr
kubeadm init --pod-network-cidr=10.178.0.0/16
```

Schritt 2: weitere Cluster Nodes ausrollen und joinen

```
....
```

Schritt 3: Installation Object für calico anders bauen

```
## This section includes base Calico installation configuration.
## For more information, see: https://docs.tigera.io/calico/latest/reference/installation/api#operator.tigera.io/v1.Installation
apiVersion: operator.tigera.io/v1
kind: Installation
metadata:
  name: default
spec:
  # Configures Calico networking.
  calicoNetwork:
    ipPools:
    - name: default-ipv4-ippool
      blockSize: 26
```

```
      cidr: 10.178.0.0/16
      encapsulation: VXLANCrossSubnet
      natOutgoing: Enabled
      nodeSelector: all()

---

## This section configures the Calico API server.
## For more information, see: https://docs.tigera.io/calico/latest/reference/installation/api#operator.tigera.io/v1.APIServer
apiVersion: operator.tigera.io/v1
kind: APIServer
metadata:
  name: default
spec: {}

---
```

Kubernetes Networking Übung

Wann wird die PodIP vergeben

Example (that does work)

```
## Show the pods that are running
kubectl get pods

## Synopsis (most simplistic example
## kubectl run NAME --image=IMAGE_EG_FROM_DOCKER
## example
kubectl run nginx --image=nginx:1.23

kubectl get pods
## on which node does it run ?
kubectl get pods -o wide
```

Example (that does not work)

```
kubectl run foo2 --image=foo2
## ImageErrPull - Image konnte nicht geladen werden
kubectl get pods
## Weitere status - info
kubectl describe pods foo2
```

Ref:

- https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#run

Host erforschen z.B. CNI mit kubectl debug

```
kubectl debug node/worker1 -it --image=ubuntu
## in der bash
cd /host/etc/cni/net.d
ls -la
apt update
apt install jq # im container
cat 10-calico.conflist | jq
```

Kubernetes CNI

Wie funktioniert das unter der Haube

Referenz:

- https://isovalent.com/blog/post/demystifying-cni/

Ablauf

- Containerd ruft CNI plugin über subcommandos: ADD, DEL, CHECK, VERSION auf (mehr subcommandos gibt es nicht)
- Was gemacht werden soll wird über JSON-Objekt übergeben
- Die Antwort kommt auch wieder als JSON zurück

Plugins die Standardmäßig schon da sind

- https://www.cni.dev/plugins/current/

CNI-Provider

- Ein Kubernetes-Cluster braucht immer ein CNI-Provider, sonst funktioniert die Kommunikation nicht und die Nodes im Cluster stehen auf NotReady
- Beispiele: Calico, WeaveNet, Antrea, Cilium, Flannel

IPAM - IP Address Management

- Ziel ist, dass Adressen nicht mehrmals vergeben werden.
- Dazu wird ein Pool bereitgestellt.
- Es gibt 3 CNI IPAM - Module:
  - host-local
  - dhcp
  - static

```
* IPAM: IP address allocation
dhcp : Runs a daemon on the host to make DHCP requests on behalf of a container
host-local : Maintains a local database of allocated IPs
static : Allocates static IPv4/IPv6 addresses to containers
```

Beispiel json für antrea (wird verwendet beim Aufruf von CNI)

```
root@worker1:/etc/cni/net.d# cat 10-antrea.conflist
{
    "cniVersion":"0.3.0",
    "name": "antrea",
    "plugins": [
        {
            "type": "antrea",
            "ipam": {
                "type": "host-local"
            }
        }
        ,
        {
            "type": "portmap",
            "capabilities": {"portMappings": true}
        }
        ,
        {
            "type": "bandwidth",
            "capabilities": {"bandwidth": true}
        }
    ]
}
```

Überblick cni provider

CNI

- Common Network Interface
- Feste Definition, wie Pod mit Netzwerk-Bibliotheken kommunizieren

Docker - Container oder andere

- Pod (Pause Container) wird hochgefahren -> über CNI -> zieht Netzwerk - IP hoch.
- Pod (Pause Container) witd runtergahren -> uber CNI -> Netzwerk - IP wird released

Welche gibt es ?

- Flannel
- Canal
- Calico
- Cilium
- Antrea (vmware)
- Weave Net

Flannel

Generell

- Flannel is a CNI which gives a subnet to each host for use with container runtimes.

Overlay - Netzwerk

- virtuelles Netzwerk was sich oben drüber und eigentlich auf Netzwerkebene nicht existiert
- VXLAN

Vorteile

- Guter einfacher Einstieg
- reduziert auf eine Binary flanneld

Nachteile

- keine Firewall - Policies möglich
- keine klassichen Netzwerk-Tools zum Debuggen möglich.

Guter Einstieg in flannel

- https://mvallim.github.io/kubernetes-under-the-hood/documentation/kube-flannel.html

Canal

General

- Auch ein Overlay - Netzwerk
- Unterstützt auch policies
- Kombination aus Flannel (Overlay) und den NetworkPolicies aus Calico

Calico

calico

Komponenten

Calico API server

- Lets you manage Calico resources directly with kubectl.

Felix

```
Main task: Programs routes and ACLs, and anything else required on the host to provide desired connectivity for the endpoints on
that host. Runs on each machine that hosts endpoints. Runs as an agent daemon.
```

BIRD

- Gets routes from Felix and distributes to BGP peers on the network for inter-host routing. Runs on each node that hosts a Felix agent. Open source, internet routing daemon.

confd

```
Monitors Calico datastore for changes to BGP configuration and global defaults such as AS number, logging levels, and IPAM
information. Open source, lightweight configuration management tool.

Confd dynamically generates BIRD configuration files based on the updates to data in the datastore. When the configuration file
changes, confd triggers BIRD to load the new files
```

Dikastes

```
Enforces NetworkPolicy for istio service mesh
```

CNI plugin

Datastore plugin

IPAM plugin

kube-controllers

```
Main task: Monitors the Kubernetes API and performs actions based on cluster state. kube-controllers.

The tigera/kube-controllers container includes the following controllers:

Policy controller
Namespace controller
Serviceaccount controller
Workloadendpoint controller
Node controller
```

Typha

```
Typha maintains a single datastore connection on behalf of all of its clients like Felix and confd. It caches the datastore state
and deduplicates events so that they can be fanned out to many listeners.
```

calicoctl

- Wird heute selten gebraucht, da das meiste heute mit kubectl über den Calico API Server realisiert werden kann
- Früher haben die neuesten NetworkPolicies/v3 nur über calioctl funktioniert

Generell

- klassische Netzwerk (BGP) - kein Overlay
- klassische Netzwerk-Tools können verwendet werden.
- eBPF ist implementiert, aber muss aktiviert

Vorteile gegenüber Flannel

- Policy über Kubernetes Object (NetworkPolicies)

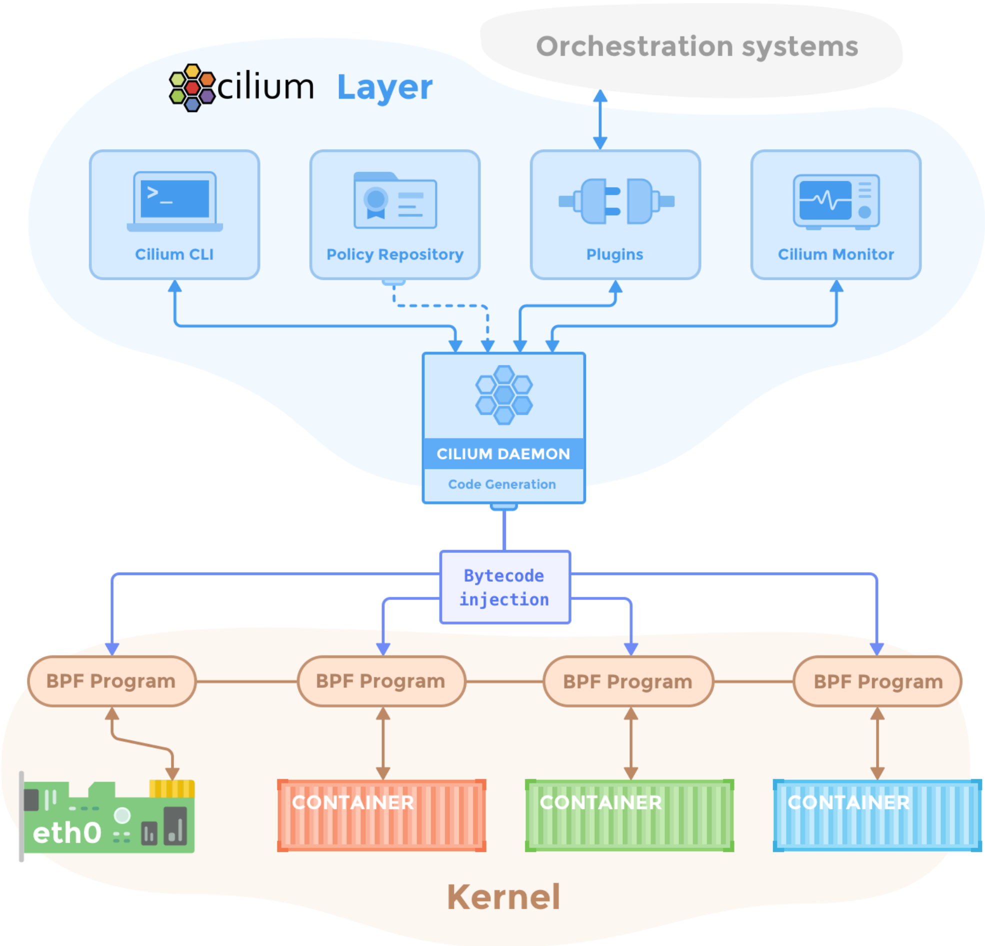
Vorteile

- ISTIO integrierbar (Service Mesh)
- Performance etwas besser als Flannel (weil keine Encapsulation)

Referenz

- https://projectcalico.docs.tigera.io/security/calico-network-policy

Cilium



Komponenten:

Cilium Agent

- Läuft auf jeder Node im Cluster
- Lauscht auf events from Orchestrierer (z.B. container gestoppt und gestartet)
- Managed die eBPF - Programme, die Linux kernel verwendet um den Netzwerkzugriff aus und in die Container zu kontrollieren

Client (CLI)

- Wird im Agent mit installiert (interagiert mit dem agent auf dem gleichen Node)
- Kann aber auch auf dem Client installiert werden auf dem kubectl läuft.

Cilium Operator

- Zuständig dafür, dass die Agents auf den einzelnen Nodes ausgerollt werden
- Es gibt ihn nur 1x im Cluster
- Ist unkritisch, sobald alles ausgerollt ist.
    - wenn dieser nicht läuft funktioniert das Networking trotzdem

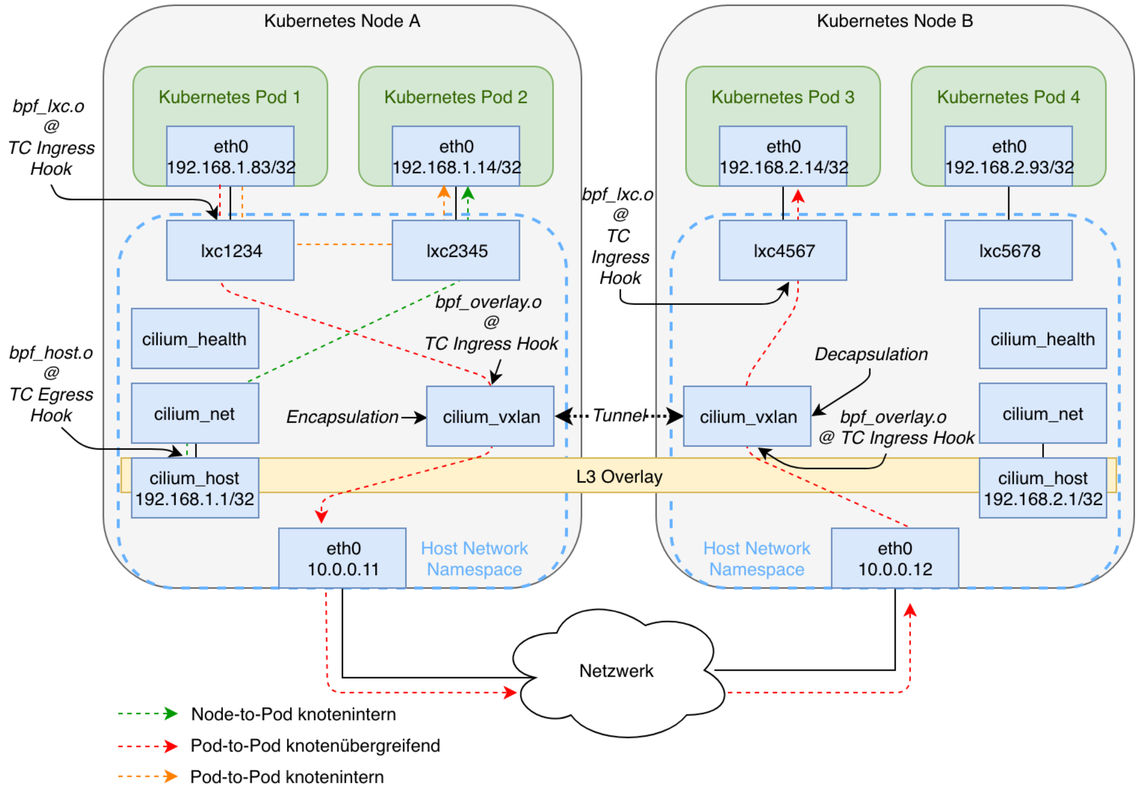
cilium CNI - Plugin

- Ist ein binary auf dem server (worker)
- wird durch die Container Runtime ausgeführt.
- cilium cni plugin interagiert mit der Cilium API auf dem Node

Datastore

- Daten werden per Default in CRD (Custom Resource Defintions) gespeichert
- Diese Resource Objekte werden von Cilium definiert und angelegt.
  - Wenn Sie angelegt sind, sind die Daten dadurch automatisch im etc - Speicher
  - Mit der weiteren Möglichkeit den Status zu speichern.
- Alternative: Speichern der Daten direkt in etcd

Generell



- Quelle: https://www.inovex.de/de/blog/kubernetes-networking-2-calico-cilium-weavenet/
- Verwendet keine Bridge sondern Hooks im Kernel, die mit eBPF aufgesetzt werden
  - Bessere Performance
- eBPF wird auch für NetworkPolicies unter der Haube eingesetzt
- Mit Ciliums Cluster Mesh lassen sich mehrere Cluster miteinander verbinden:

Vorteile

- Höhere Leistung mit eBPF-Ansatz. (extended Berkely Packet Filter)
  - JIT - Just in time compiled -
  - Bytecode wird zu MaschineCode kompiliert (Miniprogramme im Kernel)
- Ersatz für iptables (wesentlich schneller und keine Degredation wie iptables ab 5000 Services)
- Gut geeignet für größere Cluster

Weave Net

- Ähnlich calico
- Verwendet overlay netzwerk
- Sehr stabil bzgl IPV4/IPV6 (Dual Stack)
- Sehr grosses Feature-Set
- mit das älteste Plugin

Kubernetes Load Balancer / metallb (on premise)

Kubernetes Load Balancer - metallb

General

- Supports bgp and arp
- Divided into controller, speaker

Installation Ways

- helm
- manifests

Step 1: install metallb

```
## Just to show some basics
## Page from metallb says that digitalocean is not really supported well
## So we will not install the speaker .

helm repo add metallb https://metallb.github.io/metallb
```

```
## Eventually disabling speaker
## vi values.yml
```

```
helm install metallb metallb/metallb --namespace=metallb-system --create-namespace --version 0.14.8
```

Step 2: addresspool und Propagation-type (config)

```
cd
mkdir -p manifests
cd manifests
mkdir lb
cd lb
nano 01-addresspool.yml
```

```
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: first-pool
  namespace: metallb-system
spec:
  addresses:
  # we will use our external ip here
  - 134.209.231.154-134.209.231.154
  # both notations are possible
  - 157.230.113.124/32
```

```
kubectl apply -f .
```

```
nano 02-advertisement.yml
```

```
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: example
  namespace: metallb-system
```

```
kubectl apply -f .
```

Schritt 4: Test do i get an external ip

```
nano 03-deploy.yml
```

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

```
nano 04-service.yml
```

```
apiVersion: v1
kind: Service
metadata:
  name: svc-nginx
  labels:
    svc: nginx
spec:
  type: LoadBalancer
  ports:
  - port: 80
    protocol: TCP
  selector:
    run: web-nginx
```

```
kubectl apply -f .
kubectl get pods
kubectl get svc
```

```
## auf dem client
curl http://<ip aus get svc>
```

```
kubectl delete -f 03-deploy.yml 04-service.yml
```

Schritt 5: Referenz:

- https://metallb.io/installation/#installation-with-helm

Feste IP beziehen

Beispiel

```
cd
mkdir -p manifests/lb/
```

```
## Service yaml anpassen
nano 03-service.yaml
```

```
apiVersion: v1
kind: Service
metadata:
  name: svc-nginx
  labels:
    svc: nginx
spec:
  type: LoadBalancer
## eure ip aus dem pool nehmen
  loadBalancerIP: 167.99.130.85
  ports:
  - port: 80
    protocol: TCP
  selector:
    run: web-nginx
```

```
kubectl apply -f .
## ist es die von oben ?
kubectl get svc
```

Kubernetes Networking Internals

Weg von Pod zu Host -> veth / calicoctl get wep

Info: keine iptables - Regeln innerhalb veth -paar

Explanation

> ***When a packet goes from one end of a veth pair to the other — does that action trigger iptables rules?***

Answer: No, the *act* of a packet traveling **through the veth pair itself** (from one end to the other) does **not** hit iptables directly.

Think of it like this:

The `veth` pair is just a **pipe** between two network interfaces.

- When the pod sends a packet, it hits the **pod's** **`eth0`** (which is a veth interface).
- That packet is instantly transferred to the **peer interface** on the host (also a veth).
- That **transfer itself** — the jump between the veth ends — is **not processed by iptables**.

---

iptables only comes in when:

- The **host kernel processes the packet**, e.g., routing, forwarding, delivering to an application.
- The packet is being **routed** or **bridged**, or undergoes **NAT**, etc.

Visualization:

```
Pod eth0 (vethX)  <=====>  vethY on host  → bridge (cni0) → rest of the network

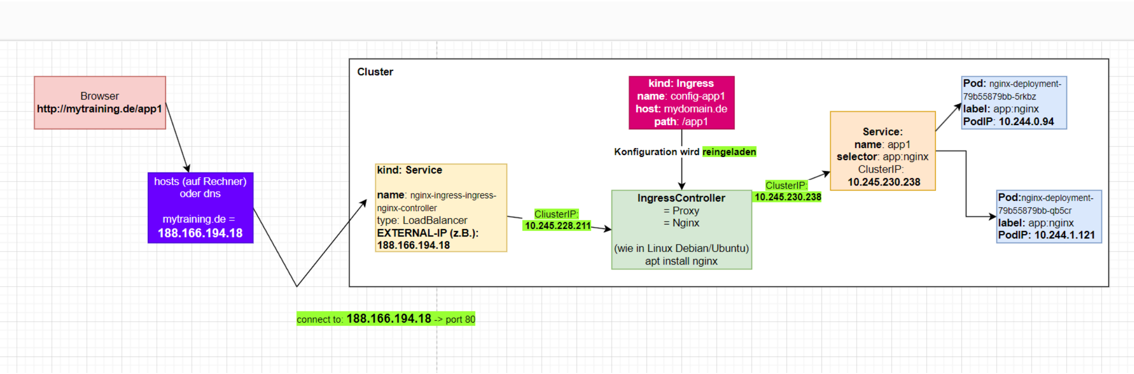
              [no iptables]        →        [iptables here]
```

So to summarize:

✅ **iptables is involved** when the **packet leaves the veth** and is handled by the host stack (e.g., routed, forwarded, NATed).
❌ **iptables is not involved** in the **link between the two ends** of the veth — that part is a simple packet transfer.

Kubernetes - IngressController

Vom Browser über den Ingress bis zum Pod - Schaubild



Ingress Controller installieren mit helm

Basics

- Das Verfahren funktioniert auch so auf anderen Plattformen, wenn helm verwendet wird und noch kein IngressController vorhanden
- Ist kein IngressController vorhanden, werden die Ingress-Objekte zwar angelegt, es funktioniert aber nicht.

Prerequisites

- kubectl und helm muss eingerichtet sein

Walkthrough (Setup Ingress Controller)

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
## Falls repo bereits mit add hinzugefügt war
## helm repo update

helm install nginx-ingress ingress-nginx/ingress-nginx --namespace ingress --create-namespace --version 4.12.2

## See when the external ip comes available
kubectl -n ingress get all
kubectl -n ingress get svc
```

```
## Output
NAME                                     TYPE           CLUSTER-IP     EXTERNAL-IP      PORT(S)                      AGE
SELECTOR
nginx-ingress-ingress-nginx-controller   LoadBalancer   10.245.78.34   157.245.20.222   80:31588/TCP,443:30704/TCP   4m39s
app.kubernetes.io/component=controller,app.kubernetes.io/instance=nginx-ingress,app.kubernetes.io/name=ingress-nginx

## Now setup wildcard - domain for training purpose
## inwx.com
*.lab1.t3isp.de A 157.245.20.222
```

Wie funktioniert das Reload und Endpunkte

- https://kubernetes.github.io/ingress-nginx/how-it-works/#nginx-configuration

How many requests per second

- https://blog.nginx.org/blog/testing-performance-nginx-ingress-controller-kubernetes

Multiple IngresController nginx

Set different helm-values and install second release

```
controller:
  electionID: ingress-controller-leader
  ingressClass: internal-nginx  # default: nginx
  ingressClassResource:
    name: internal-nginx  # default: nginx
    enabled: true
    default: false
    controllerValue: "k8s.io/internal-ingress-nginx"  # default: k8s.io/ingress-nginx
```

Reference:

- https://kubernetes.github.io/ingress-nginx/user-guide/multiple-ingress/

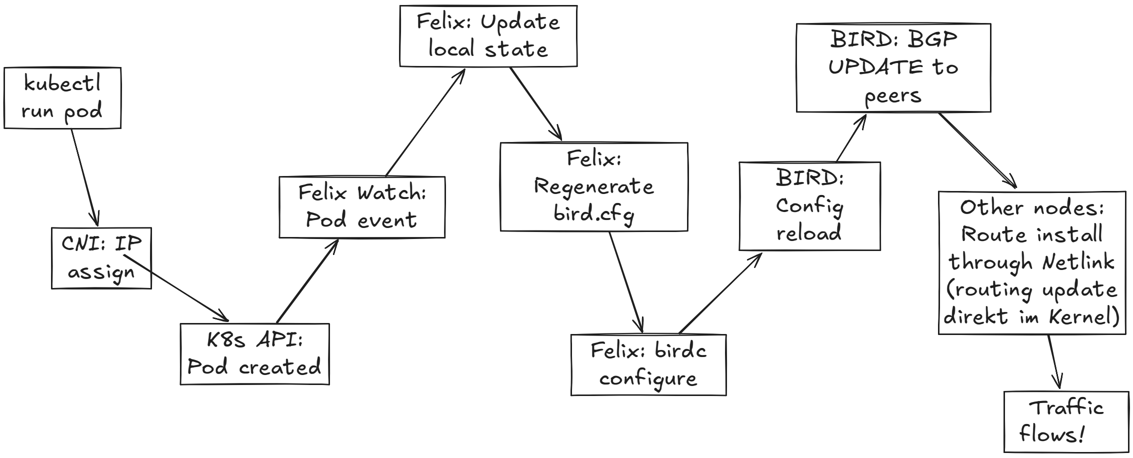
Kubernetes - Ingress - Examples

Routing with Header

- https://github.com/jmetzger/training-kubernetes-networking/blob/main/kubernetes/ingress/nginx/examples/header-abfragen.md

Kubernetes - Calico - BGP

Calico BGP - Wirkweise Schaubild



Kubernetes NetworkPolicy

Einfache Übung Network Policy

Schritt 1: Deployment und Service erstellen

```
KURZ=jm
kubectl create ns policy-demo-$KURZ
```

```
cd
mkdir -p manifests
cd manifests
mkdir -p np
cd np
```

```
nano 01-deployment.yml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 1
  template:
    metadata:
```

```
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.23
        ports:
        - containerPort: 80
```

```
kubectl -n policy-demo-$KURZ apply -f .
```

```
nano 02-service.yml
```

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  type: ClusterIP # Default Wert
  ports:
  - port: 80
    protocol: TCP
  selector:
    app: nginx
```

```
kubectl -n policy-demo-$KURZ apply -f .
```

Schritt 2: Zugriff testen ohne Regeln

```
## lassen einen 2. pod laufen mit dem auf den nginx zugreifen
kubectl run --namespace=policy-demo-$KURZ access --rm -ti --image busybox
```

```
## innerhalb der shell
wget -q nginx -O -
```

```
## Optional: Pod anzeigen in 2. ssh-session zu jump-host
kubectl -n policy-demo-$KURZ get pods --show-labels
```

Schritt 3: Policy festlegen, dass kein Zugriff erlaubt ist.

```
nano 03-default-deny.yaml
```

```
## Schritt 2: Policy festlegen, dass kein Ingress-Traffic erlaubt
## in diesem namespace: policy-demo-$KURZ
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: default-deny
spec:
  podSelector:
    matchLabels: {}
```

```
kubectl -n policy-demo-$KURZ apply -f .
```

Schritt 3.5: Verbindung mit deny all Regeln testen

```
kubectl run --namespace=policy-demo-$KURZ access --rm -ti --image busybox
```

```
## innerhalb der shell
wget -q nginx -O -
```

Schritt 4: Zugriff erlauben von pods mit dem Label run=access (alle mit run gestarteten pods mit namen access haben dieses label per default)

```
nano 04-access-nginx.yaml
```

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: access-nginx
spec:
  podSelector:
    matchLabels:
```

```
      app: nginx
  ingress:
    - from:
      - podSelector:
          matchLabels:
            run: access
```

```
kubectl -n policy-demo-$KURZ apply -f .
```

Schritt 5: Testen (zugriff sollte funktionieren)

```
## lassen einen 2. pod laufen mit dem auf den nginx zugreifen
## pod hat durch run -> access automatisch das label run:access zugewiesen
kubectl run --namespace=policy-demo-$KURZ access --rm -ti --image busybox
```

```
## innerhalb der shell
wget -q nginx -O -
```

Schritt 6: Pod mit label run=no-access - da sollte es nicht gehen

```
kubectl run --namespace=policy-demo-$KURZ no-access --rm -ti --image busybox
```

```
## in der shell
wget -q nginx -O -
```

Schritt 7: Aufräumen

```
kubectl delete ns policy-demo-$KURZ
```

Ref:

- https://projectcalico.docs.tigera.io/security/tutorials/kubernetes-policy-basic

Kubernetes policy api - neu noch alpha

- https://network-policy-api.sigs.k8s.io/reference/examples/

Calico - GUI (since calico 3.30)

Calico GUI

Important (current state: calico 3.30.x)

- In current version by default traffic to whisker is not allowed (only through port-forwarding)
- Whisker and Goldmane are the first components in calico that create NetworkPolicy Objects
- Example:

```
kubectl get networkpolicies  -A
kubectl -n calico-system get networkpolicy whisker -o yaml
```

```
tln1@k8s-client:~$ kubectl -n calico-system get networkpolicy whisker -o yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  creationTimestamp: "2025-06-02T10:58:54Z"
  generation: 1
  name: whisker
  namespace: calico-system
  ownerReferences:
  - apiVersion: operator.tigera.io/v1
    blockOwnerDeletion: true
    controller: true
    kind: Whisker
    name: default
    uid: 2bfd420b-b36e-4d07-9ea0-21989d7233ca
  resourceVersion: "12513"
  uid: a319f3a2-aa84-46eb-82a1-be7a2f85764c
spec:
  egress:
  - ports:
    - port: 7443
      protocol: TCP
    to:
    - podSelector:
        matchLabels:
          app.kubernetes.io/name: goldmane
  - ports:
    - port: 53
      protocol: TCP
    - port: 53
      protocol: UDP
    to:
    - namespaceSelector:
        matchLabels:
          projectcalico.org/name: kube-system
      podSelector:
        matchLabels:
          k8s-app: kube-dns
  podSelector:
    matchLabels:
      app.kubernetes.io/name: whisker
  policyTypes:
  - Ingress
  - Egress
```

Labelling of namespaces

- Namespaces are labelled when create

```
kubectl describe ns ingress
```

Labels of pods (ingress controller)

```
kubectl -n ingress get pods --show-labels
```

Prerequisites:

- ingress controller installed (nginx) in namespace ingress.
- wildcard dns - setup for ip of ingress controller

```
## Look for external ip
kubectl -n ingress get svc
```

Walkthrough

```
cd
mkdir -p manifests/whisker
cd manifests/whisker
```

```
## credentials erstellen
htpasswd -c auth admin  # Enter your desired password
kubectl create secret generic whisker-basic-auth --from-file=auth -n calico-system
```

```
nano 01-ingress.yaml
```

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: whisker
  namespace: calico-system
  annotations:
     nginx.ingress.kubernetes.io/auth-type: basic
     nginx.ingress.kubernetes.io/auth-secret: whisker-basic-auth
     nginx.ingress.kubernetes.io/auth-realm: "Authentication Required"
spec:
  ingressClassName: nginx
  rules:
    - host: <hier-domain>.t3isp.de
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: whisker
                port:
                  number: 8081
```

```
nano 02-networkpolicy.yaml
```

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: whisker-ingress
  namespace: calico-system
spec:
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: ingress
      podSelector:
        matchLabels:
          app.kubernetes.io/component: controller
          app.kubernetes.io/name: ingress-nginx
    ports:
    - port: 8081
      protocol: TCP
  podSelector:
    matchLabels:
      app.kubernetes.io/name: whisker
  policyTypes:
  - Ingress
```

```
kubectl apply -f .
```

Reference:

- https://github.com/orgs/projectcalico/discussions/10395

Gateway API

Overview

Features

- Responsibility separation
- TrafficRouting based on RequestHeader / Environment Variable
- LoadBalancing (Gewichtung: 10% an Service 1, 90% an Service 2)
- TCP und gRPC-Routing

Komponenten

1. GatewayController (Kong, Nginx, Traefik, HAProxy, Istio) - Software
2. GatewayClass (Stable)
3. Gateway (Stable)
4. HttpRoute (GA) / TCPRoute (experimentell) / gRPCRoute (experimentell)

Shared responsibility

Gateway API can be split into different responsibility roles

Hallo

Reference

- https://gateway-api.sigs.k8s.io/

Implementations

Kong

- Ref: https://github.com/kong/kubernetes-ingress-controller

Nginx

- Ref: https://github.com/nginxinc/nginx-gateway-fabric

Traefik

- Important: Traefik currently only partly supports spec v1.0.0
- Currently: v1.1.0 from Kubernetes is already out

```
## Please do not use Traefik right now for the gateway api
```

Reference:

- https://gateway-api.sigs.k8s.io/implementations/

Conformance

- https://github.com/kubernetes-sigs/gateway-api/tree/main/conformance/reports

Guides

- https://gateway-api.sigs.k8s.io/guides/

Conformance Report - What is implemented in which software

- https://github.com/kubernetes-sigs/gateway-api/tree/main/conformance/reports

Kubernetes - Misc

Bash completion installieren

Walkthrough

```
## Eventuell, wenn bash-completion nicht installiert ist.
apt install bash-completion
source /usr/share/bash-completion/bash_completion
## is it installed properly
type _init_completion
```

```
## activate for all users
kubectl completion bash | sudo tee /etc/bash_completion.d/kubectl > /dev/null

## verifizieren - neue login shell
su -

## zum Testen
kubectl g<TAB>
kubectl get
```

Alternative für k als alias für kubectl

```
source <(kubectl completion bash)
complete -F __start_kubectl k
```

Reference

- https://kubernetes.io/docs/tasks/tools/included/optional-kubectl-configs-bash-linux/

vim support for yaml

Ubuntu (im Unterverzeichnis /etc/vim/vimrc.local - systemweit)

```
hi CursorColumn cterm=NONE ctermbg=lightred ctermfg=white
autocmd FileType y?ml setlocal ts=2 sts=2 sw=2 ai number expandtab cursorline cursorcolumn
```

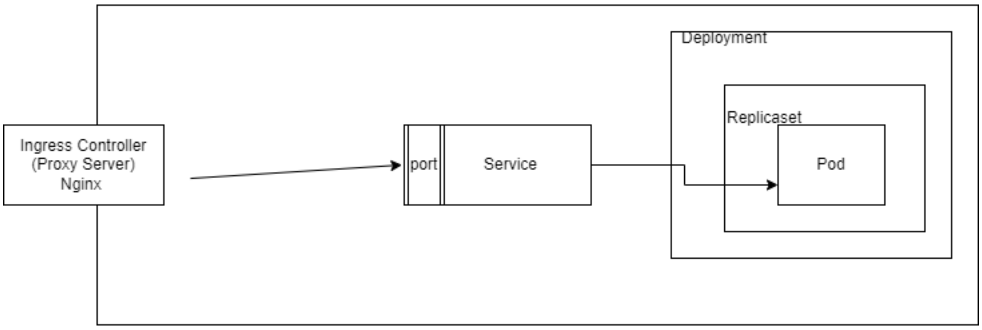
Testen

```
vim test.yml
Eigenschaft: <return> # springt eingerückt in die nächste Zeile um 2 spaces eingerückt

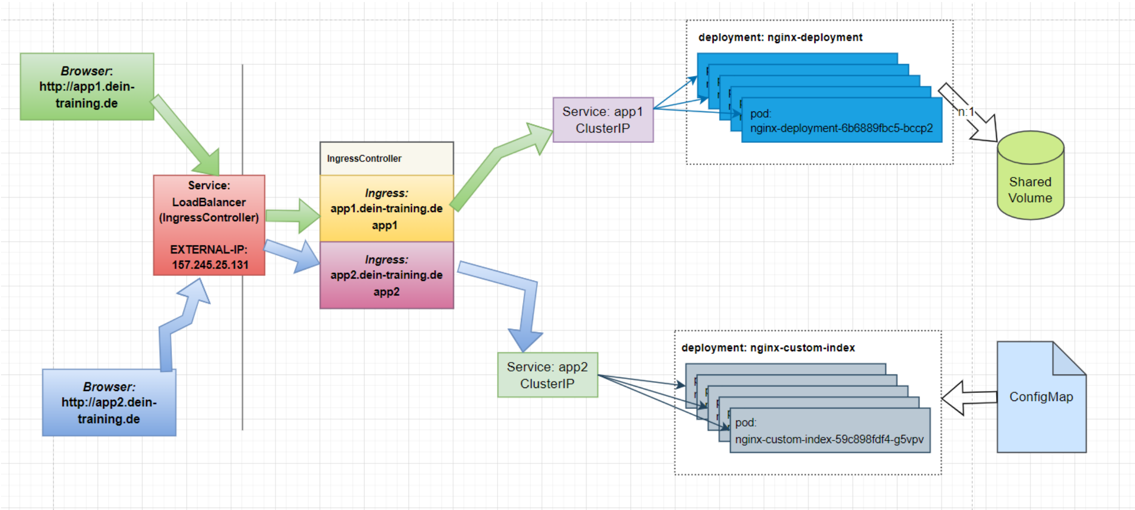
## evtl funktioniert vi test.yml auf manchen Systemen nicht, weil kein vim (vi improved)
```

Kubernetes - Projekt Applikation

Bauen einer Applikation mit Resource Objekten

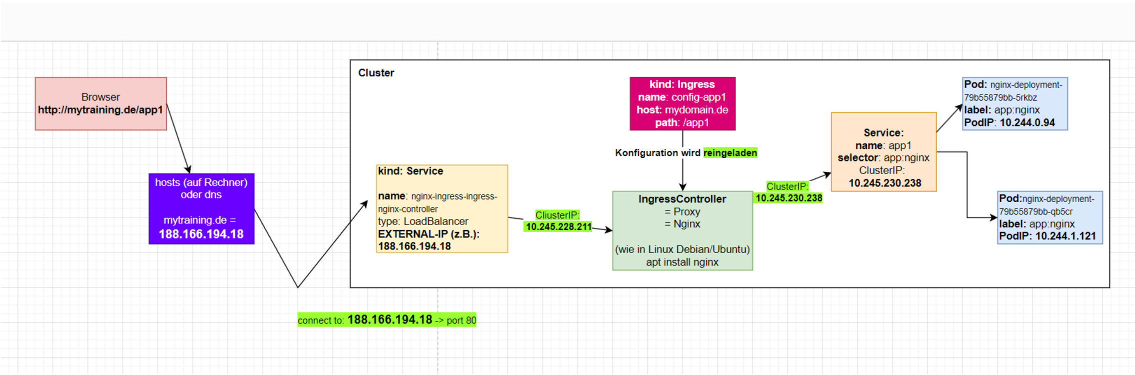


Anatomie einer Applikation



Kubernetes - IngressController

Vom Browser über den Ingress bis zum Pod - Schaubild



Ingress Controller installieren mit helm

Basics

- Das Verfahren funktioniert auch so auf anderen Plattformen, wenn helm verwendet wird und noch kein IngressController vorhanden
- Ist kein IngressController vorhanden, werden die Ingress-Objekte zwar angelegt, es funktioniert aber nicht.

Prerequisites

- kubectl und helm muss eingerichtet sein

Walkthrough (Setup Ingress Controller)

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
## Falls repo bereits mit add hinzugefügt war
## helm repo update

helm install nginx-ingress ingress-nginx/ingress-nginx --namespace ingress --create-namespace --version 4.12.2

## See when the external ip comes available
kubectl -n ingress get all
kubectl -n ingress get svc
```

```
## Output
NAME                                        TYPE           CLUSTER-IP     EXTERNAL-IP      PORT(S)                      AGE
SELECTOR
nginx-ingress-ingress-nginx-controller   LoadBalancer   10.245.78.34   157.245.20.222   80:31588/TCP,443:30704/TCP   4m39s
app.kubernetes.io/component=controller,app.kubernetes.io/instance=nginx-ingress,app.kubernetes.io/name=ingress-nginx

## Now setup wildcard - domain for training purpose
## inwx.com
*.lab1.t3isp.de A 157.245.20.222
```

Wie funktioniert das Reload und Endpunkte

- https://kubernetes.github.io/ingress-nginx/how-it-works/#nginx-configuration

How many requests per second

- https://blog.nginx.org/blog/testing-performance-nginx-ingress-controller-kubernetes

Kubernetes - Projekt Applikation - Step by Step (Netzwerk)

kubectl/manifest/deployments

```
cd
mkdir -p manifests
cd manifests
mkdir 03-deploy
cd 03-deploy
```

```
nano deploy.yml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 8 # tells deployment to run 8 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.21
        ports:
        - containerPort: 80
```

```
kubectl apply -f deploy.yml
```

```
## kubectl get -f deploy.yml -o yaml
kubectl get all
```

kubectl/manifest/service

Schritt 1: Deployment

```
cd
mkdir -p manifests
cd manifests
mkdir 04-service
cd 04-service
##vi 01-deploy.yml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 7 # tells deployment to run 8 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.23
        ports:
        - containerPort: 80
```

```
kubectl apply -f .
```

Schritt 2:

```
nano 02-svc.yml
```

```
apiVersion: v1
kind: Service
metadata:
  name: svc-nginx
spec:
  type: ClusterIP
  ports:
  - port: 80
    protocol: TCP
  selector:
    app: nginx
```

```
kubectl apply -f .
```

Schritt 3 Endpunkte gefunden ?

```
kubectl get svc svc-nginx
kubectl describe svc svc-nginx
```

```
kubectl get endpoints svc-nginx
kubectl get endpoints svc-nginx -o yaml
```

Schritt 4 Löschen deployment und prüfen die Endpunkte

```
kubectl delete -f deploy.yml
## Uups Endpunkte sind weg
kubectl describe svc svc-nginx
```

```
kubectl apply -f .
## Endpunkte sind wieder da, aber neue POD-IPs
kubectl describe svc svc-nginx
```

Ref.

- https://kubernetes.io/docs/concepts/services-networking/connect-applications-service/

DNS - Resolution - Services

What is resolved

```
svc-nginx
svc-nginx.jochen # svc-nginx.<namespace>
svc-nginx.jochen.svc
svc-nginx.jochen.svc.cluster.local
```

Exercise

```
kubectl run podtest --rm -ti --image busybox
If you don't see a command prompt, try pressing enter.
/ # wget -O - http://apple-service.jochen
Connecting to apple-service.jochen (10.245.39.214:80)
writing to stdout
apple-tln1
-                    100%
|****************************************************************************************************************|    11  0:00:00
ETA
written to stdout
/ # wget -O - http://apple-service.jochen.svc.cluster.local
Connecting to apple-service.jochen.svc.cluster.local (10.245.39.214:80)
writing to stdout
apple-tln1
-                    100%
|****************************************************************************************************************|    11  0:00:00
ETA
written to stdout
/ # wget -O - http://apple-service
Connecting to apple-service (10.245.39.214:80)
writing to stdout
apple-tln1
-                    100%
|****************************************************************************************************************|    11  0:00:00
ETA
written to stdout
```

How to find the FQDN (Full qualified domain name)

```
## in busybox (clusterIP)
nslookup 10.109.6.53
name = svc-nginx.jochen.svc.cluster.local
```

Beispiel Ingress mit Hostnamen

Walkthrough

Step 1: pods and services

```
cd
mkdir -p manifests
cd manifests
mkdir abi
cd abi
```

```
nano apple.yml
```

```
## apple.yml
## vi apple.yml
kind: Pod
apiVersion: v1
metadata:
  name: apple-app
  labels:
    app: apple
spec:
  containers:
    - name: apple-app
      image: hashicorp/http-echo
      args:
        - "-text=apple-<dein-name>"
---

kind: Service
apiVersion: v1
metadata:
  name: apple-service
```

```
spec:
  selector:
    app: apple
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5678 # Default port for image
```

```
kubectl apply -f apple.yml
```

```
nano banana.yml
```

```
## banana
## vi banana.yml
kind: Pod
apiVersion: v1
metadata:
  name: banana-app
  labels:
    app: banana
spec:
  containers:
    - name: banana-app
      image: hashicorp/http-echo
      args:
        - "-text=banana-<dein-name>"

---

kind: Service
apiVersion: v1
metadata:
  name: banana-service
spec:
  selector:
    app: banana
  ports:
    - port: 80
      targetPort: 5678 # Default port for image
```

```
kubectl apply -f banana.yml
```

Step 2: Ingress

```
nano ingress.yaml
```

```
## Ingress
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: example-ingress
  annotations:
    ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
  - host: "<euername>.lab1.t3isp.de"
    http:
      paths:
        - path: /apple
          backend:
            serviceName: apple-service
            servicePort: 80
        - path: /banana
          backend:
            serviceName: banana-service
            servicePort: 80
```

```
## ingress
kubectl apply -f ingress.yml
kubectl get ing
```

Reference

- https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-ingress-guide-nginx-example.html

Find the problem

```
## Hints

## 1. Which resources does our version of kubectl support
## Can we find Ingress as "Kind" here.
kubectl api-ressources

## 2. Let's see, how the configuration works
kubectl explain --api-version=networking.k8s.io/v1 ingress.spec.rules.http.paths.backend.service

## now we can adjust our config
```

Direct Solution

```
nano ingress.yaml
```

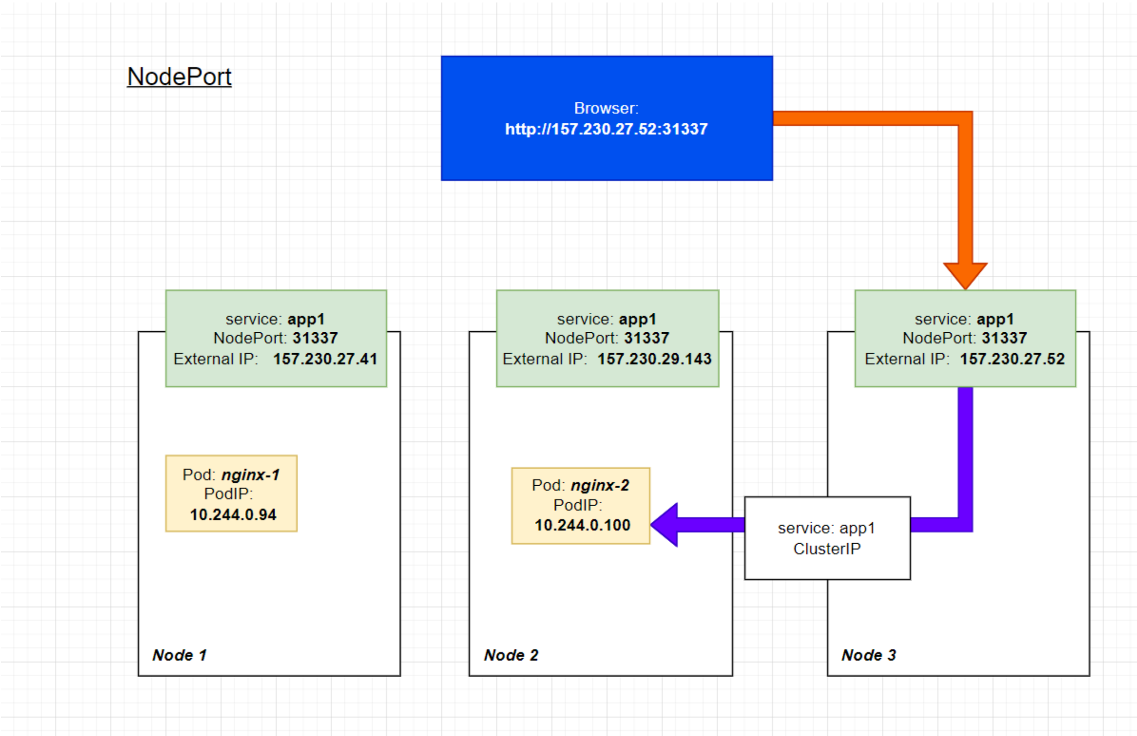
```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
  annotations:
    ingress.kubernetes.io/rewrite-target: /
    # with the ingress controller from helm, you need to set an annotation
    # old version useClassName instead
    # otherwice it does not know, which controller to use
    # kubernetes.io/ingress.class: nginx
spec:
  ingressClassName: nginx
  rules:
## <deinname> ersetzen durch dienen namen, z.B. jochen
## jochen.lab1.t3isp.de
  - host: "<deinname>.lab1.t3isp.de"
    http:
      paths:
        - path: /apple
          pathType: Prefix
          backend:
            service:
              name: apple-service
              port:
                number: 80
        - path: /banana
          pathType: Prefix
          backend:
            service:
              name: banana-service
              port:
                number: 80
```

```
kubectl apply -f .
kubectl describe ingress example-ingress
```

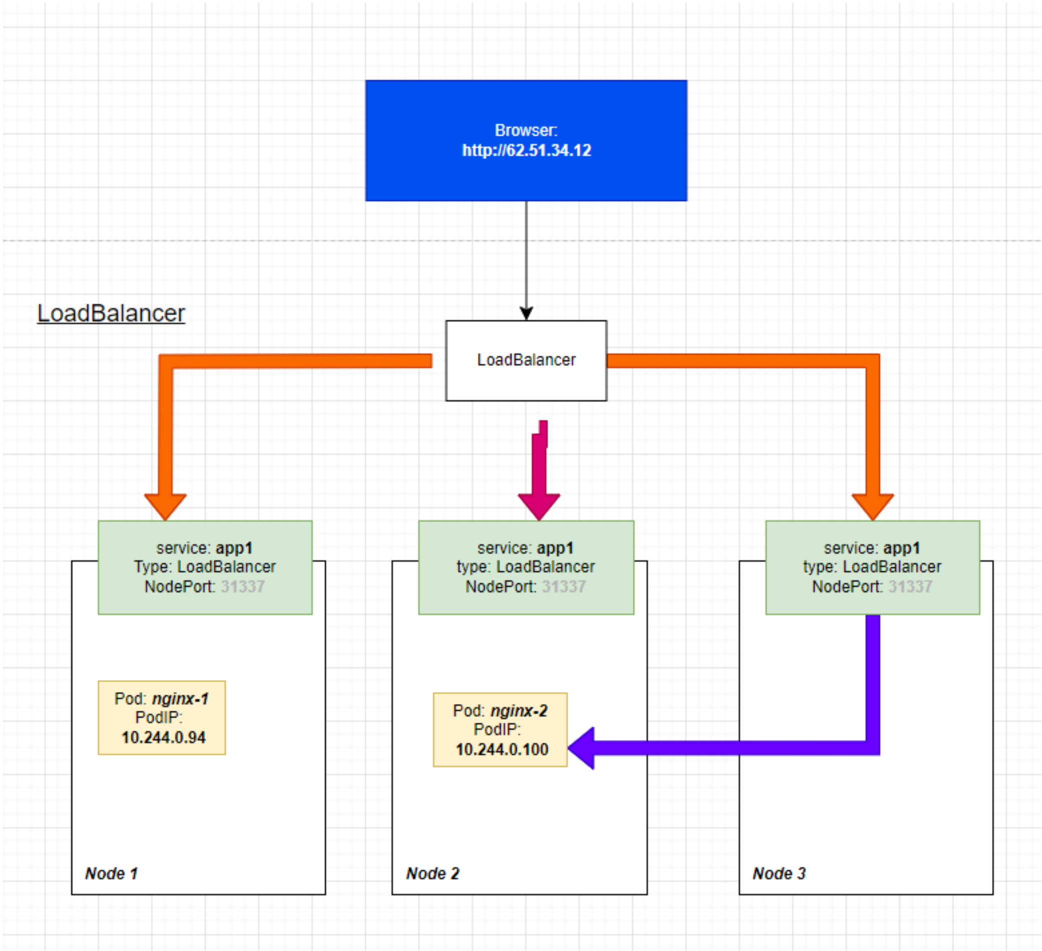
```
## Testen im browser oder curl mit hostnamen
## Variante 1:
## z.B.
curl http://jochen.lab1.t3isp.de/apple
curl http://jochen.lab1.t3isp.de/banana
```

Kubernetes Services (Schaubilder)

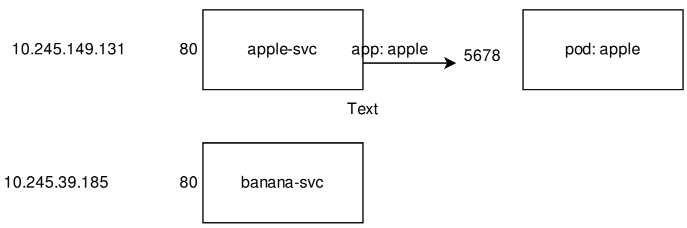
Services -> type: NodePort



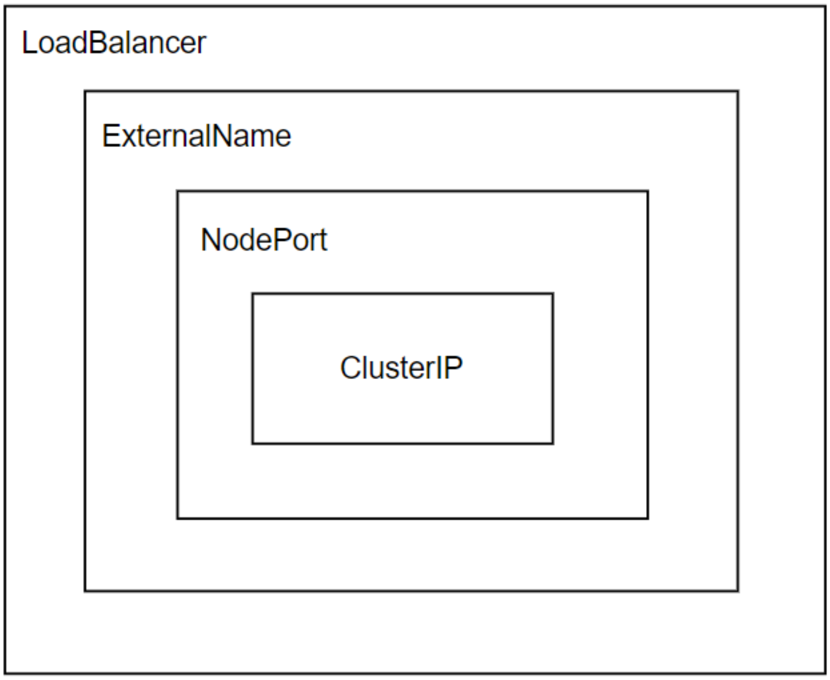
Services -> type: LoadBalancer



Services - Aufbau



Service Typen / Ebenen - Schaubild



Kubernetes Cheatsheet

Das Tool kubectl (Devs/Ops) - Spickzettel

Allgemein

```
## Zeige Information über das Cluster
kubectl cluster-info

## Welche api-resources gibt es ?
kubectl api-resources

## Hilfe zu object und eigenschaften bekommen
kubectl explain pod
kubectl explain pod.metadata
kubectl explain pod.metadata.name
```

Arbeiten mit manifesten

```
kubectl apply -f nginx-replicaset.yml
## Wie ist aktuell die hinterlegte config im system
kubectl get -o yaml -f nginx-replicaset.yml

## Änderung in nginx-replicaset.yml z.B. replicas: 4
## dry-run - was wird geändert
kubectl diff -f nginx-replicaset.yml

## anwenden
kubectl apply -f nginx-replicaset.yml

## Alle Objekte aus manifest löschen
kubectl delete -f nginx-replicaset.yml
```

Ausgabeformate

```
## Ausgabe kann in verschiedenen Formaten erfolgen
kubectl get pods -o wide # weitere informationen
## im json format
kubectl get pods -o json
```

```
## gilt natürluch auch für andere kommandos
kubectl get deploy -o json
kubectl get deploy -o yaml

## get a specific value from the complete json - tree
kubectl get node k8s-nue-jo-ff1p1 -o=jsonpath='{.metadata.labels}'
```

Zu den Pods

```
## Start einen pod // BESSER: direkt manifest verwenden
## kubectl run podname image=imagename
kubectl run nginx image=nginx

## Pods anzeigen
kubectl get pods
kubectl get pod
## Format weitere Information
kubectl get pod -o wide
## Zeige labels der Pods
kubectl get pods --show-labels

## Pods aus allen Namespaces anzeigen
kubectl get pods -A

## Zeige pods mit einem bestimmten label
kubectl get pods -l app=nginx

## Status eines Pods anzeigen
kubectl describe pod nginx

## Pod löschen
kubectl delete pod nginx

## Kommando in pod ausführen
kubectl exec -it nginx -- bash
## direkt in den 1. Pod des Deployments wechseln
kubectl exec -it deployment/name-des-deployments -- bash
```

Logs ausgeben

```
kubectl logs podname
## -n = namespace
## | less -> seitenweise Ausgabe
kubectl -n ingress logs nginx-ingress-ingress-nginx-controller-7bc7c7776d-jpj5h | less
```

Arbeiten mit namespaces

```
## Welche namespaces auf dem System
kubectl get ns
kubectl get namespaces
## Standardmäßig wird immer der default namespace verwendet
## wenn man kommandos aufruft
kubectl get deployments

## Möchte ich z.B. deployment vom kube-system (installation) aufrufen,
## kann ich den namespace angeben
kubectl get deployments --namespace=kube-system
kubectl get deployments -n kube-system

## wir wollen unseren default namespace ändern
kubectl config set-context --current --namespace <dein-namespace>
```

Referenz

- https://kubernetes.io/de/docs/reference/kubectl/cheatsheet/

Kubernetes - Imperative Behle

kubectl example with run

Example (that does work)

```
## Show the pods that are running
kubectl get pods

## Synopsis (most simplistic example
```

```
## kubectl run NAME --image=IMAGE_EG_FROM_DOCKER
## example
kubectl run nginx --image=nginx:1.23

kubectl get pods
## on which node does it run ?
kubectl get pods -o wide
```

Example (that does not work)

```
kubectl run foo2 --image=foo2
## ImageErrPull - Image konnte nicht geladen werden
kubectl get pods
## Weitere status - info
kubectl describe pods foo2
```

Ref:

- https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#run

Kubernetes - Wartung / Debugging

Netzwerkverbindung zu pod testen

Situation

```
Managed Cluster und ich kann nicht auf einzelne Nodes per ssh zugreifen
```

Behelf: Eigenen Pod starten mit busybox

```
## laengere Version
kubectl run podtest --rm -ti --image busybox -- /bin/sh
```

```
## kuerzere Version
kubectl run podtest --rm -ti --image busybox
```

Example test connection

```
## wget befehl zum Kopieren
wget -O - http://10.244.0.99
ping 10.244.0.99
```

```
## -O -> Output (grosses O (buchstabe))
kubectl run podtest --rm -ti --image busybox -- /bin/sh
/ # wget -O - http://10.244.0.99
/ # exit
```

Debug Container

Walkthrough Debug Container

```
kubectl run ephemeral-demo --image=registry.k8s.io/pause:3.1 --restart=Never
kubectl exec -it ephemeral-demo -- sh

kubectl debug -it ephemeral-demo --image=busybox
```

Example with nginx

```
kubectl run --image=nginx nginx
### debug this container
kubectl debug -it nginx --image=busybox
```

Walkthrough Debug Node

```
kubectl get nodes
kubectl debug node/mynode -it --image=ubuntu
```

Reference

- https://kubernetes.io/docs/tasks/debug/debug-application/debug-running-pod/#ephemeral-container

Curl from pod api-server

https://nieldw.medium.com/curling-the-kubernetes-api-server-d7675cfc398c

mtr (mytraceroute) from pod to pod

Setup

```
cd
mkdir -p manifests
cd manifests
mkdir -p traceroute
cd traceroute
```

```
nano pods.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-worker1
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    kubernetes.io/hostname: worker1
---
apiVersion: v1
kind: Pod
metadata:
  name: nginx-worker2
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    kubernetes.io/hostname: worker2
---
```

```
kubectl apply -f .
```

my traceroute

```
kubectl debug -it node/worker1 --image nicolaka/netshoot
```

```
mtr <ip-addresse-des-pod-auf-worker2>
## ohne dns auflösung
mtr -n <ip-addresse-des-pod-auf-worker2>
```

```


### Debugging mit tcpdump - Beispiel Ingress


### Prerequisites: Project abi is up and running



### Debug traffic to pod
```

IP des pod apple-app rausfiltern

kubectl get pods -o wide

kubectl debug apple-app -it --image nicolaka/netshoot

Show processes of other container first

kubectl debug apple-app -it --image nicolaka/netshoot --target=apple-app

```
#### in pod
```

ps aux tcpdump -n port 5678

```

#### in 2. Session (kubectl)
```

kubectl run -it --rm podtester --image=busybox

```

```

wget -O - :5678

```
### Debug traffic to ingress controller

#### mit netshoot connecten

##### Variante 1: Direkt
```

kubectl -n ingress debug nginx-ingress-ingress-nginx-controller-7bc7c7776d-jpj5h -it --image nicolaka/netshoot

```
```

in der shell

tcpdump -n port 80

write to file in pcap format

Older versions of tcpdump truncate packets to 68 or 96 bytes. If this is the case, use -s to capture full-sized packets: tcpdump -i -s 65535 -w

```
#### Variante 2: Im Hintergrund laufen lassen und connecten
```

kubectl -n ingress debug nginx-ingress-ingress-nginx-controller-7bc7c7776d-jpj5h --image nicolaka/netshoot -- sleep infinite kubectl -n ingress exec -it nginx-ingress-ingress-nginx-controller-7bc7c7776d-jpj5h -c debugger-gwvsr -- zsh

```
```

in der shell

tcpdump -n port 80

write to file in pcap format

Older versions of tcpdump truncate packets to 68 or 96 bytes. If this is the case, use -s to capture full-sized packets: tcpdump -i -s 65535 -w

```
#### Testen
```

Im browser url aufrufen

z.B.

http://jochen.lab1.t3isp.de

```
### Debugging mit tcpdump - Beispiel NodePort

### Schritt 1: Ausgangsbasis
```

Basis 03-deploy

cd cd manifests cd 03-deploy ls -la

```
### Schritt 2: Wir passen den Service an und Anzahl der Replicas in Deploy
```

nano 02-svc.yml

```
```

Ändern Zeile mit type:

Ändern in type: NodePort

```
```

nano deploy.yml

```
```

Ändern Zeile mit replicas:

Ändern in replicas: 1

```
```

kubectl apply -f .

```
### Schritt 2: Daten sammeln
```

kubectl get svc svc-nginx

Beispiel -> 32682

Worker öffentliche IP rausfinden

kubectl get node -o wide | grep worker1

worker1 -> 164.92.131.128

```
### Schritt 3: 2. Session öffnen
```

ein debug-pod auf worker1 starten

kubectl debug -it node/worker1 --image nicolaka/netshoot .# im pod tcpdump auf port von Schritt 2 tcdump -i eth0 port 32682

```
### Schritt 4: 1. Session
```

while kontinuierlich

curl http://164.92.131.128:32682

```
### Schritt 5: 2. Session ausgabe tcpdump



### Optional: Schritt 6
```

Welcher Pod von nginx -> daneben einen debug container starten kubectl get pods kubectl debug -it nginx-deployment-5948f7484f-sbq9v --image nicolaka/netshoot

```


## Kubernetes - Network - Interna

### Kubernetes CIDR


### Grafik

![image](https://github.com/user-attachments/assets/87bb7926-d962-4f71-b8b4-f04b7ab44ec6)

### Cluster CIDR - IP-Bereich für das gesamte Kubernetes Cluster
```

Netzbereich für mein gesamtes Cluster

10.244.0.0/16

```
### POD-CIDR - Teilbereich aus der Cluster - CIDR pro Node
```

Jede Node bekommt ein Teilnetz

Beispiel cilium

node 1 -> network.cilium.io/ipv4-pod-cidr: 10.244.0.0/25 node 2 -> network.cilium.io/ipv4-pod-cidr: 10.244.0.128/25 node 3 -> network.cilium.io/ipv4-pod-cidr: 10.244.1.128/25 node 4 -> network.cilium.io/ipv4-pod-cidr: 10.244.1.0/25

```
### POD-IP
```

```
  * Wird aus POD-CIDR des jeweiligen Nodes vergeben
```

pod bekommt aus netzbereich POD-CIDR auf Node eine IP-Adresse zugewiesen

CILIUM CNI macht das z.B.

POD-CIDR: 10.244.1.128/25 -> POD - IP: 10.244.1.180

```
### Service-CIDR
```

Netzbereich für IP-Adressen der Services z.B. 10.109.0.0/16

```
## Kubernetes - Network - CNI Interna

### Wann wird CNI aufgerufen ?

  * https://github.com/jmetzger/training-kubernetes-networking/blob/main/kubernetes/internals/cni/wann-wird-cni-aufgerufen.md

## Kubernetes - Netzwerk (CNI's) / Mesh

### Netzwerk Interna


### Network Namespace for each pod

#### Overview

![Overview](https://www.inovex.de/wp-content/uploads/2020/05/Container-to-Container-Networking_2_neu-400x401.png)
![Overview Kubernetes Networking](https://www.inovex.de/wp-content/uploads/2020/05/Container-to-Container-Networking_3_neu-
400x412.png)

#### General

  * Each pod will have its own network namespace
    * with routing, networkdevices
  * Connection to default namespace to host is done through veth - Link to bridge on host network
    * similar like on docker to docker0
```

Each container is connected to the bridge via a veth-pair. This interface pair functions like a virtual point-to-point ethernet connection and connects the network namespaces of the containers with the network namespace of the host

```
  * Every container is in the same Network Namespace, so they can communicate through localhost
    * Example with hashicorp/http-echo container 1 and busybox container 2


### Pod-To-Pod Communication (across nodes)

#### Prerequisites

  * pods on a single node as well as pods on a topological remote can establish communication at all times
   * Each pod receives a unique IP address, valid anywhere in the cluster. Kubernetes requires this address to not be subject to
network address   translation (NAT)
   * Pods on the same node through virtual bridge (see image above)

#### General (what needs to be done) - and could be done manually

   * local bridge networks of all nodes need to be connected
   * there needs to be an IPAM (IP-Address Managemenet) so addresses are only used once
   * The need to be routes so, that each bridge can communicate with the bridge on the other network
   * Plus: There needs to be a rule for incoming network
   * Also: A tunnel needs to be set up to the outside world.

#### General - Pod-to-Pod Communication (across nodes) - what would need to be done

![pod to pod across nodes](https://www.inovex.de/wp-content/uploads/2020/05/Pod-to-Pod-Networking.png)


#### General - Pod-to-Pod Communication (side-note)

  * This could of cause be done manually, but it is too complex
  * So Kubernetes has created an Interface, which is well defined
    * The interface is called CNI (common network interface)
```

```
  * Funtionally is achieved through Network Plugin (which use this interface)
    * e.g. calico / cilium / weave net / flannel


#### CNI

 * CNI only handles network connectivity of container and the cleanup of allocated resources (i.e. IP addresses) after containers have been deleted (garbage collection) and therefore is lightweight and quite easy to implement.
 * There are some basic libraries within CNI which do some basic stuff.





### Hidden Pause Container

#### What is for ?

 * Holds the network - namespace for the pod
 * Gets started first and falls asleep later
 * Will still be there, when the other containers die
```

cd mkdir -p manifests cd manifests mkdir pausetest cd pausetest nano 01-nginx.yml

```

```

vi nginx-static.yml

apiVersion: v1 kind: Pod metadata: name: nginx-pausetest labels: webserver: nginx:1.21 spec: containers:

- name: web image: nginx

```

```

kubectl apply -f .

als root auf dem worker node

ctr -n k8s.io c list | grep pause

```


### References

 * https://www.inovex.de/de/blog/kubernetes-networking-part-1-en/
 * https://www.inovex.de/de/blog/kubernetes-networking-2-calico-cilium-weavenet/

### Wirkweise cni





### Referenz:

 * https://isovalent.com/blog/post/demystifying-cni/


### Ablauf
  * Containerd ruft CNI plugin über subcommandos: ADD, DEL, CHECK, VERSION auf (mehr subcommandos gibt es nicht)
  * Was gemacht werden soll wird über JSON-Objekt übergeben
  * Die Antwort kommt auch wieder als JSON zurück

### Plugins die Standardmäßig schon da sind

  * https://www.cni.dev/plugins/current/

### CNI-Provider

  * Ein Kubernetes-Cluster braucht immer ein CNI-Provider, sonst funktioniert die Kommunikation nicht und die Nodes im Cluster stehen auf NotReady
  * Beispiele: Calico, WeaveNet, Antrea, Cilium, Flannel

### IPAM - IP Address Management

  * Ziel ist, dass Adressen nicht mehrmals vergeben werden.
  * Dazu wird ein Pool bereitgestellt.
  * Es gibt 3 CNI IPAM - Module:
```

```
    * host-local
    * dhcp
    * static
```

- IPAM: IP address allocation dhcp : Runs a daemon on the host to make DHCP requests on behalf of a container host-local : Maintains a local database of allocated IPs static : Allocates static IPv4/IPv6 addresses to containers

```

### Beispiel json für antrea (wird verwendet beim Aufruf von CNI)

![image](https://github.com/jmetzger/training-kubernetes-networking/assets/1933318/85dcbcf4-0c01-4fe0-a737-dd0f7d04231f)

### Übersicht Netzwerke (cni-provider)


### CNI

  * Common Network Interface
  * Feste Definition, wie Pod  mit Netzwerk-Bibliotheken kommunizieren

### Docker - Container oder andere

  * Pod (Pause Container) wird hochgefahren -> über CNI -> zieht Netzwerk - IP  hoch.
  * Pod (Pause Container) witd runtergahren -> uber CNI -> Netzwerk - IP wird released

### Welche gibt es ?

  * Flannel
  * Canal
  * Calico
  * Cilium
  * Antrea (vmware)
  * Weave Net

### Flannel

#### Generell

  * Flannel is a CNI which gives a subnet to each host for use with container runtimes.

#### Overlay - Netzwerk

  * virtuelles Netzwerk was sich oben drüber und eigentlich auf Netzwerkebene nicht existiert
  * VXLAN

#### Vorteile

  * Guter einfacher Einstieg
  * reduziert auf eine Binary flanneld

#### Nachteile

  * keine Firewall - Policies möglich
  * keine klassichen Netzwerk-Tools zum Debuggen möglich.

#### Guter Einstieg in flannel

  * https://mvallim.github.io/kubernetes-under-the-hood/documentation/kube-flannel.html

### Canal

#### General

  * Auch ein Overlay - Netzwerk
  * Unterstützt auch policies
  * Kombination aus Flannel (Overlay) und den NetworkPolicies aus Calico

### Calico

![calico](https://tanzu.vmware.com/developer/guides/container-networking-calico-refarch/images/calico-components.png)

#### Komponenten

##### Calico API server

  * Lets you manage Calico resources directly with kubectl.

##### Felix
```

Main task: Programs routes and ACLs, and anything else required on the host to provide desired connectivity for the endpoints on that host. Runs on each machine that hosts endpoints. Runs as an agent daemon.

```
##### BIRD

  * Gets routes from Felix and distributes to BGP peers on the network for inter-host routing. Runs on each node that hosts a
Felix agent. Open source, internet routing daemon.

#### confd
```

Monitors Calico datastore for changes to BGP configuration and global defaults such as AS number, logging levels, and IPAM information. Open source, lightweight configuration management tool.

Confd dynamically generates BIRD configuration files based on the updates to data in the datastore. When the configuration file changes, confd triggers BIRD to load the new files

```
#### Dikastes
```

Enforces NetworkPolicy for istio service mesh

```
#### CNI plugin

#### Datastore plugin

#### IPAM plugin

#### kube-controllers
```

Main task: Monitors the Kubernetes API and performs actions based on cluster state. kube-controllers.

The tigera/kube-controllers container includes the following controllers:

Policy controller Namespace controller Serviceaccount controller Workloadendpoint controller Node controller

```
#### Typha
```

Typha maintains a single datastore connection on behalf of all of its clients like Felix and confd. It caches the datastore state and deduplicates events so that they can be fanned out to many listeners.

```
#### calicoctl

  * Wird heute selten gebraucht, da das meiste heute mit kubectl über den Calico API Server realisiert werden kann
  * Früher haben die neuesten NetworkPolicies/v3 nur über calioctl funktioniert

#### Generell

  * klassische Netzwerk (BGP) - kein Overlay
  * klassische Netzwerk-Tools können verwendet werden.
  * eBPF ist implementiert, aber muss aktiviert

#### Vorteile gegenüber Flannel

  * Policy über Kubernetes Object (NetworkPolicies)

#### Vorteile

  * ISTIO integrierbar (Service Mesh)
  * Performance etwas besser als Flannel (weil keine Encapsulation)

#### Referenz
  * https://projectcalico.docs.tigera.io/security/calico-network-policy

### Cilium

![Cilium Architecture](https://docs.cilium.io/en/stable/_images/cilium-arch.png)

#### Komponenten:

##### Cilium Agent

  * Läuft auf jeder Node im Cluster
  * Lauscht auf events from Orchestrierer (z.B. container gestoppt und gestartet)
  * Managed die eBPF - Programme, die Linux kernel verwendet um den Netzwerkzugriff aus und in die Container zu kontrollieren
```

```
##### Client (CLI)

  * Wird im Agent mit installiert (interagiert mit dem agent auf dem gleichen Node)
  * Kann aber auch auf dem Client installiert werden auf dem kubectl läuft.

##### Cilium Operator

  * Zuständig dafür, dass die Agents auf den einzelnen Nodes ausgerollt werden
  * Es gibt ihn nur 1x im Cluster
  * Ist unkritisch, sobald alles ausgerollt ist.
    * wenn dieser nicht läuft funktioniert das Networking trotzdem

##### cilium CNI - Plugin

  * Ist ein binary auf dem server (worker)
  * wird durch die Container Runtime ausgeführt.
  * cilium cni plugin interagiert mit der Cilium API auf dem Node

#### Datastore

  * Daten werden per Default in CRD (Custom Resource Defintions) gespeichert
  * Diese Resource Objekte werden von Cilium definiert und angelegt.
    * Wenn Sie angelegt sind, sind die Daten dadurch automatisch im etc - Speicher
    * Mit der weiteren Möglichkeit den Status zu speichern.
  * Alternative: Speichern der Daten direkt in etcd

#### Generell

![Cilium](https://www.inovex.de/wp-content/uploads/2020/05/Cilium.png)

  * Quelle: https://www.inovex.de/de/blog/kubernetes-networking-2-calico-cilium-weavenet/

  * Verwendet keine Bridge sondern Hooks im Kernel, die mit eBPF aufgesetzt werden
    * Bessere Performance
  * eBPF wird auch für NetworkPolicies unter der Haube eingesetzt
  * Mit Ciliums Cluster Mesh lassen sich mehrere Cluster miteinander verbinden:

#### Vorteile

  * Höhere Leistung mit eBPF-Ansatz. (extended Berkely Packet Filter)
    * JIT - Just in time compiled -
    * Bytecode wird zu MaschineCode kompiliert (Miniprogramme im Kernel)
  * Ersatz für iptables (wesentlich schneller und keine Degredation wie iptables ab 5000 Services)
  * Gut geeignet für größere Cluster

### Weave Net

  * Ähnlich calico
  * Verwendet overlay netzwerk
  * Sehr stabil bzgl IPV4/IPV6 (Dual Stack)
  * Sehr grosses Feature-Set
  * mit das älteste Plugin

### Calico/Cilium - nginx example NetworkPolicy


### Schritt 1: Deployment und Service erstellen
```

KURZ=jm kubectl create ns policy-demo-$KURZ

cd mkdir -p manifests cd manifests mkdir -p np cd np

nano 01-deployment.yml

apiVersion: apps/v1 kind: Deployment metadata: name: nginx-deployment spec: selector: matchLabels: app: nginx replicas: 1 template: metadata: labels: app: nginx spec: containers: - name: nginx image: nginx:1.23 ports: - containerPort: 80

kubectl -n policy-demo-$KURZ apply -f .

nano 02-service.yml

```
```

apiVersion: v1 kind: Service metadata: name: nginx spec: type: ClusterIP # Default Wert ports:

- port: 80 protocol: TCP selector: app: nginx

```
```

kubectl -n policy-demo-$KURZ apply -f .

```
### Schritt 2: Zugriff testen ohne Regeln
```

lassen einen 2. pod laufen mit dem auf den nginx zugreifen

kubectl run --namespace=policy-demo-$KURZ access --rm -ti --image busybox

```
```

innerhalb der shell

wget -q nginx -O -

```
```

Optional: Pod anzeigen in 2. ssh-session zu jump-host

kubectl -n policy-demo-$KURZ get pods --show-labels

```
### Schritt 3: Policy festlegen, dass kein Zugriff erlaubt ist.
```

nano 03-default-deny.yaml

```
```

Schritt 2: Policy festlegen, dass kein Ingress-Traffic erlaubt

in diesem namespace: policy-demo-$KURZ

kind: NetworkPolicy apiVersion: networking.k8s.io/v1 metadata: name: default-deny spec: podSelector: matchLabels: {}

```
```

kubectl -n policy-demo-$KURZ apply -f .

```
### Schritt 3.5: Verbindung mit deny all Regeln testen
```

kubectl run --namespace=policy-demo-$KURZ access --rm -ti --image busybox

```
```

innerhalb der shell

wget -q nginx -O -

```
### Schritt 4: Zugriff erlauben von pods mit dem Label run=access (alle mit run gestarteten pods mit namen access haben dieses
label per default)
```

nano 04-access-nginx.yaml

```
```

apiVersion: networking.k8s.io/v1 kind: NetworkPolicy metadata: name: access-nginx spec: podSelector: matchLabels: app: nginx ingress: - from: - podSelector: matchLabels: run: access

```
```

kubectl -n policy-demo-$KURZ apply -f .

```
### Schritt 5: Testen (zugriff sollte funktionieren)
```

lassen einen 2. pod laufen mit dem auf den nginx zugreifen

pod hat durch run -> access automatisch das label run:access zugewiesen

kubectl run --namespace=policy-demo-$KURZ access --rm -ti --image busybox

```
```

innerhalb der shell

wget -q nginx -O -

```


### Schritt 6: Pod mit label run=no-access - da sollte es nicht gehen
```

kubectl run --namespace=policy-demo-$KURZ no-access --rm -ti --image busybox

```
```

in der shell

wget -q nginx -O -

```

### Schritt 7: Aufräumen
```

kubectl delete ns policy-demo-$KURZ

```


### Ref:

  * https://projectcalico.docs.tigera.io/security/tutorials/kubernetes-policy-basic

### Beispiele Ingress Egress NetworkPolicy


### Links

  * https://github.com/ahmetb/kubernetes-network-policy-recipes
  * https://k8s-examples.container-solutions.com/examples/NetworkPolicy/NetworkPolicy.html

### Example with http (Cilium !!)
```

apiVersion: "cilium.io/v2" kind: CiliumNetworkPolicy description: "L7 policy to restrict access to specific HTTP call" metadata: name: "rule1" spec: endpointSelector: matchLabels: type: l7-test ingress:

- fromEndpoints:
  - matchLabels: org: client-pod toPorts:
  - ports:
    - port: "8080" protocol: TCP rules: http:
      - method: "GET" path: "/discount"

```

### Downside egress

  * No valid api for anything other than IP's and/or Ports
  * If you want more, you have to use CNI-Plugin specific, e.g.

#### Example egress with ip's
```

Allow traffic of all pods having the label role:app

egress only to a specific ip and port

apiVersion: networking.k8s.io/v1 kind: NetworkPolicy metadata: name: test-network-policy namespace: default spec: podSelector: matchLabels: role: app policyTypes:

- Egress egress:
- to:
  - ipBlock: cidr: 10.10.0.0/16 ports:
  - protocol: TCP port: 5432

```

### Example Advanced Egress (cni-plugin specific)
```

```
#### Cilium
```

apiVersion: v1 kind: Pod metadata: name: nginx-static-web labels: webserver: nginx spec: containers:

- name: web image: nginx

```

```

apiVersion: cilium.io/v2 kind: CiliumNetworkPolicy metadata: name: "fqdn-pprof"

namespace: msp

spec: endpointSelector: matchLabels: webserver: nginx egress:

- toFQDNs:
  - matchPattern: '*.google.com'
- toPorts:
  - ports:
    - port: "53" protocol: ANY rules: dns:
      - matchPattern: '*'

```

```

kubectl apply -f .

```
#### Calico

  * Only Calico enterprise
    * Calico Enterprise extends Calico’s policy model so that domain names (FQDN / DNS) can be used to allow access from a pod or
set of pods (via label selector) to external resources outside of your cluster.
    * https://projectcalico.docs.tigera.io/security/calico-enterprise/egress-access-controls

#### Using isitio as mesh (e.g. with cilium/calico )

##### Installation of sidecar in calico

  * https://projectcalico.docs.tigera.io/getting-started/kubernetes/hardway/istio-integration

##### Example
```

apiVersion: networking.k8s.io/v1 kind: NetworkPolicy metadata: name: test-network-policy namespace: default spec: podSelector: matchLabels: role: app policyTypes:

- Egress egress:
- to:
  - ipBlock: cidr: 10.10.0.0/16 ports:
  - protocol: TCP port: 5432

```
### Kubernetes Ports/Protokolle

  * https://kubernetes.io/docs/reference/networking/ports-and-protocols/

### IPV4/IPV6 Dualstack

  * https://kubernetes.io/docs/concepts/services-networking/dual-stack/

### Gute Präsentation zu cni

  *
https://archive.fosdem.org/2023/schedule/event/network_cni_unleashed/attachments/slides/5713/export/events/attachments/network_cni_u


## Kubernetes calico (CNI-Plugin) - Part 1 Installation

### calico cni installieren


### Walkthrough
```

Step 1 - Install the operator

kubectl create -f https://raw.githubusercontent.com/projectcalico/calico/v3.30.1/manifests/tigera-operator.yaml

Step 2 - Install the custom resources

kubectl create -f https://raw.githubusercontent.com/projectcalico/calico/v3.30.1/manifests/custom-resources.yaml

```
### Testing
```

kubectl -n tigera-operator get pods kubectl -n calico-system get pods kubectl -n calico-apiserver get pods

Sind die nodes schon ready

kubectl get nodes

kubectl -n kube-system get pods kubectl -n kube-system get pods coredns-7c65d6cfc9-f6f56 kubectl -n kube-system describe pods coredns-7c65d6cfc9-f6f56

```
### Reference

  * https://docs.tigera.io/calico/latest/getting-started/kubernetes/quickstart

### calicoctl auf client installieren
```

cd /usr/local/bin curl -L https://github.com/projectcalico/calico/releases/download/v3.28.2/calicoctl-linux-amd64 -o calicoctl chmod +x ./calicoctl

```
### Install calicoctl in pod


### General

#### It was like that ....

  * calicoctl used to do validation locally in calicoctl for your manifests in the projectcalico/v3 api-version
  * This version was not available in kube-api-server

#### Now ....

  * Validation takes place on server side.
  * For this to work the kube-api-server needs to be configured with calico
  * Now the preferred method is to use kubectl (without dependencies to calicoctl) but not for.....
    * calicoctl node
    * calicoctl ipam
    * calicoctl convert
    * calicoctl version

#### Reference:

  * https://docs.tigera.io/calico/latest/operations/calicoctl/configure/kdd




### calicoctl Installation walkthrough (running in pod)

#### Find out version
```

welche version von calico setzen wir aktuell auf dem server ein

kubectl -n kube-system get ds calico-node -o=jsonpath='{.spec.template.spec.containers[0].image}'

docker.io/calico/node:v3.23.5

```
#### Pod erstellen für calicoctl auf Basis von
```

cd mkdir -p manifests cd manifests mkdir calicoctl cd calicoctl vi calicoctl.yaml

```

```

https://raw.githubusercontent.com/projectcalico/calico/v3.25.1/manifests/calicoctl.yaml

Calico Version master

https://projectcalico.docs.tigera.io/releases#master

This manifest includes the following component versions:

calico/ctl:v3.25.1

apiVersion: v1 kind: ServiceAccount metadata: name: calicoctl namespace: kube-system

---

apiVersion: v1 kind: Pod metadata: name: calicoctl namespace: kube-system spec: nodeSelector: kubernetes.io/os: linux hostNetwork: true serviceAccountName: calicoctl containers:

- name: calicoctl image: calico/ctl:v3.23.5 command:
  - /calicoctl args:
  - version
  - --poll=1m env:
  - name: DATASTORE_TYPE value: kubernetes

---

kind: ClusterRole apiVersion: rbac.authorization.k8s.io/v1 metadata: name: calicoctl rules:

- apiGroups: [""] resources:
  - namespaces
  - nodes verbs:
  - get
  - list
  - update
- apiGroups: [""] resources:
  - nodes/status verbs:
  - update
- apiGroups: [""] resources:
  - pods
  - serviceaccounts verbs:
  - get
  - list
- apiGroups: [""] resources:
  - pods/status verbs:
  - update
- apiGroups: ["crd.projectcalico.org"] resources:
  - bgppeers
  - bgpconfigurations
  - clusterinformations
  - felixconfigurations
  - globalnetworkpolicies
  - globalnetworksets
  - ippools
  - ipreservations
  - kubecontrollersconfigurations
  - networkpolicies
  - networksets
  - hostendpoints
  - ipamblocks
  - blockaffinities
  - ipamhandles
  - ipamconfigs verbs:
  - create
  - get
  - list
  - update
  - delete
- apiGroups: ["networking.k8s.io"] resources:
  - networkpolicies verbs:
  - get
  - list

---

apiVersion: rbac.authorization.k8s.io/v1 kind: ClusterRoleBinding metadata: name: calicoctl roleRef: apiGroup: rbac.authorization.k8s.io kind: ClusterRole name: calicoctl subjects:

- kind: ServiceAccount name: calicoctl namespace: kube-system

```
### calicoctl verwenden
```

this will always work, no matter what version

kubectl -n kube-system exec calicoctl -- /calicoctl version

this will only work without flags, if we have the same version

on both sides

```
## Kubernetes calico (CNI-Plugin) - Part 2 Internals

### Find corresponding networks - from pod to host


### Walkthrough  (without calicoctl)

```bash
## Step 1: create pod
kubectl run nginx-master --image=nginx
## Find out on which node it runs
kubectl get pods -o wide
## create a debug container
kubectl debug -it nginx-master --image=busybox
```

```
## now within debug pod found out interface
ip a | grep @
```

```
## Ausgabe
3: eth0@if22: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
```

```
## Log in to worker node  where pod runs and check interfaces
kubectl debug -it node/worker1 --image=busybox
```

```
## on worker node
## show matched line starting with 22 and then another 4 lines
ip a | grep -A 5 ^22
## e.g.
##
ip a | grep -A 5 ^22
22: cali42c2aab93f3@if3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff link-netns cni-5adf994b-3a7e-c344-5d82-ef1f7a293d88
    inet6 fe80::ecee:eeff:feee:eeee/64 scope link
       valid_lft forever preferred_lft forever
```

Get information with calicoctl (installed on client)

```
## für den namespace defaut bzw. den konfigurierten
calicoctl get wep
calicoctl get workloadendpoints

## für alle namespaces
calicoctl get wep -A
```

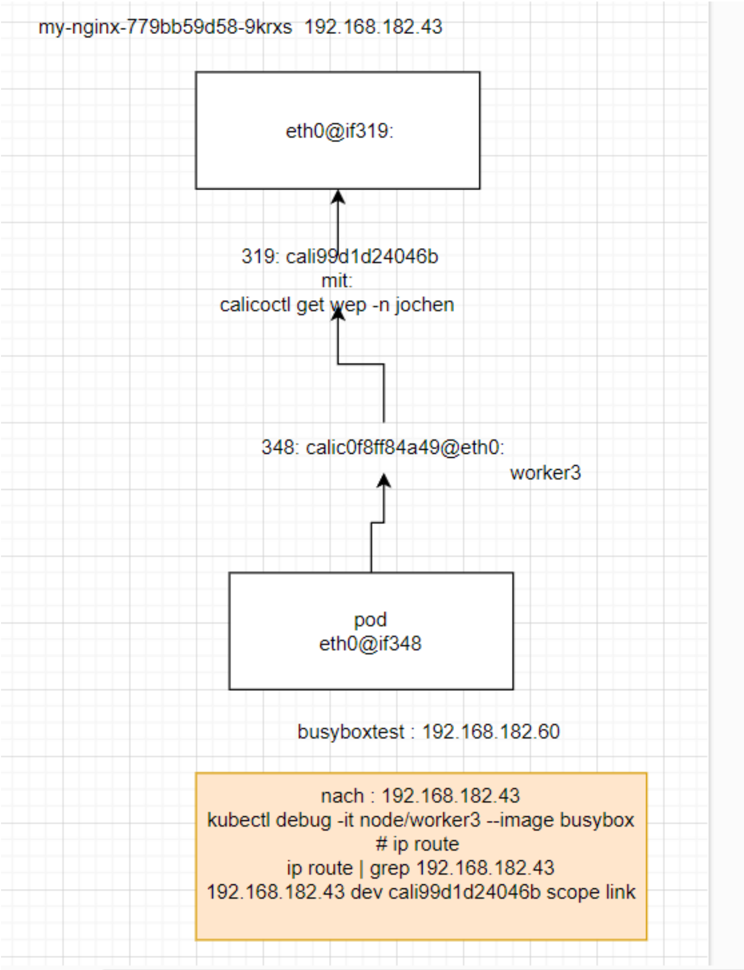
Firewall - Regeln

```
## Now you are able to determine the firewall rules
## you will find fw and tw rules (fw - from workload and tw - to workload)
iptables-legacy -L -v | grep  cali42c2aab93f3
```

```
## ... That is what you see as an example
Chain cali-tw-cali42c2aab93f3 (1 references)
 pkts bytes target     prot opt in     out     source               destination
   10  1384 ACCEPT     all  --  any    any     anywhere             anywhere             /* cali:WKA8EzdUNM0rVty1 */ ctstate
RELATED,ESTABLISHED
    0     0 DROP       all  --  any    any     anywhere             anywhere             /* cali:wr_OqGXKIN_LWnX0 */ ctstate
INVALID
    0     0 MARK       all  --  any    any     anywhere             anywhere             /* cali:kOUMqNj8np60A3Bi */ MARK and
0xfffeffff
```

Internals - Pod to Pod - Communication on Worker3 (node))


```



Übung (Teil 1)

```
cd
mkdir -p manifests/2pods
cd manifests/2pods
```

```
nano pod1.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-von
spec:
  nodeName: worker3
  containers:
    - name: nginx
      image: nginx:1.24
```

```
nano pod2.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-nach
spec:
  nodeName: worker3
  containers:
    - name: busybox
      image: busybox:1.35
      command: ["sleep", "3600"]
```

```
kubectl apply -f .
## Ip rausfinden
kubectl get pods -o wide

## mit dem host-system verbinden über eine debug node pod
kubectl debug -it node/worker3 --image=busybox
```

```
## in der bash
route -n | grep <ip-pod-nach>
```

Übung (Teil2) Auf den anderen worker nodes ?

```
nano pod3.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-auf-worker1
spec:
  nodeName: worker1
  containers:
    - name: busybox
      image: busybox:1.35
      command: ["sleep", "3600"]
```

```
kubectl apply -f .
kubectl get pods -o wide
calicoctl get wep
```

```
## z.B. wenn vorher worker3 -> dann worker1
kubectl debug node/worker1 -it --image=busybox
```

```
## in der bash
route -n | grep <ip-pod-nach>
```

Debug pod-2-pod on worker1

Walkthrough

```
## leave worker3 as is
kubectl label nodes worker1 machine=worker1
kubectl label nodes worker2 machine=worker2
kubectl get nodes --show-labels
```

1. Deployment auf worker1

```
cd
mkdir -p manifests
cd manifests
mkdir calicotest
cd calicotest
```

```
nano 01-deploy.yml
```

```
## nginx-deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment-calicotest
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

```
      nodeSelector:
        machine: worker1
```

2. noch ein Pod auf worker1

```
nano 02-pod.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-calicotest
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    machine: worker1
```

```
kubectl apply -f .
```

3. Find out about cali-interfaces

```
## Ip von ziel rausfinden
calicoctl get wep
```

```
kubectl debug -it node/worker1 --image=busybox
```

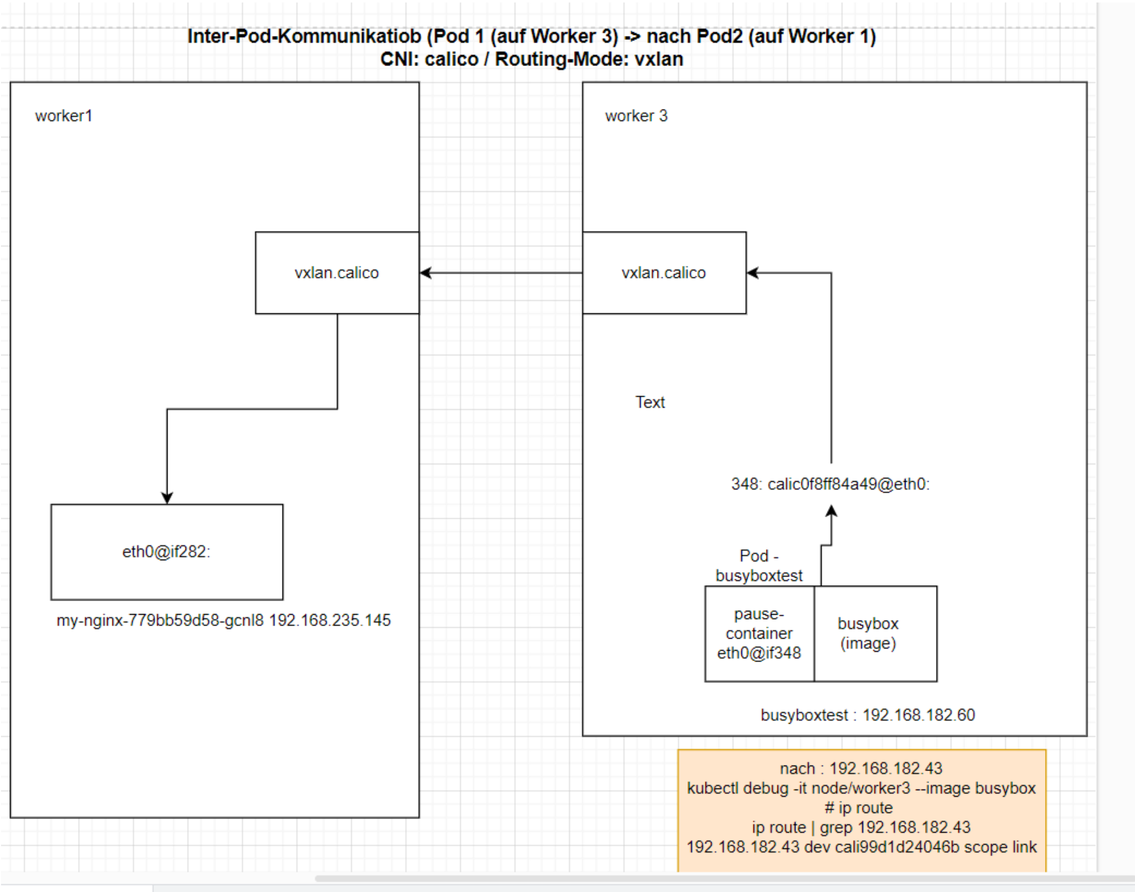
```
## Example
## nginx-calicotest (192.168.235.141) -> ngin-deployment-calicotest.....z6rmp (192.168.235.138)
## Now grep for the destination
route -n | grep 192.168.235.138
```

Internals - Inter-Pod - Communication (worker 3 -> worker 1) vxlan cross subnet

Wann ?

- bei vxlan (dann immer vxlan)
- oder: vxlan cross subnet (Wenn zwei Subnetze nicht direkt erreichbar sind, wg. z.B. Firewall)

Bild



iptables involved in veth - calico ?

Explanation

> ***When a packet goes from one end of a veth pair to the other — does that action trigger iptables rules?***

Answer: No, the *act* of a packet traveling **through the veth pair itself** (from one end to the other) does **not** hit iptables directly.

Think of it like this:

The `veth` pair is just a **pipe** between two network interfaces.

- When the pod sends a packet, it hits the **pod's** `eth0` (which is a veth interface).
- That packet is instantly transferred to the **peer interface** on the host (also a veth).
- That **transfer itself** — the jump between the veth ends — is **not processed by iptables**.

---

iptables only comes in when:

- The **host kernel processes the packet**, e.g., routing, forwarding, delivering to an application.
- The packet is being **routed** or **bridged**, or undergoes **NAT**, etc.

Visualization:

```
Pod eth0 (vethX)  <=====>  vethY on host  → bridge (cni0) → rest of the network

             [no iptables]       →        [iptables here]
```

So to summarize:

✅ **iptables is involved** when the **packet leaves the veth** and is handled by the host stack (e.g., routed, forwarded, NATed).
❌ **iptables is not involved** in the **link between the two ends** of the veth — that part is a simple packet transfer.

Kubernetes calico (CNI-Plugin) - Part 3

calicoctl Cheatsheet

vom client aus

```
calicoctl ipam show
calicoctl ipam show --show-blocks
calicoctl ipam check

calicoctl get felixconfiguration
calicoctl get felixconfiguration default -o yaml
## Wird bgp verwendet
kubectl -n calico-system logs ds/calico-node  | grep -i bgp

### von der node aus / dort calicoctl installieren
calicoctl node status
```

```
IPv4 BGP status
+--------------+-------------------+-------+------------+-------------+
| PEER ADDRESS |     PEER TYPE     | STATE |   SINCE    |    INFO     |
+--------------+-------------------+-------+------------+-------------+
| 10.135.0.10  | node-to-node mesh | up    | 2024-09-17 | Established |
| 10.135.0.29  | node-to-node mesh | up    | 2024-09-17 | Established |
| 10.135.0.12  | node-to-node mesh | up    | 2024-09-17 | Established |
+--------------+-------------------+-------+------------+-------------+
```

Welcher Routing-Mode wird im aktuellen Cluster verwendet

```
kubectl -n calico-system describe ds calico-node | grep -A 35  calico-node
## or specific
kubectl -n calico-system describe ds calico-node | egrep -i -e vxlan  -e cluster_type
```

```
Environment:
      DATASTORE_TYPE:                      kubernetes
      WAIT_FOR_DATASTORE:                  true
      CLUSTER_TYPE:                        k8s,operator,bgp
      CALICO_DISABLE_FILE_LOGGING:         false
      FELIX_DEFAULTENDPOINTTOHOSTACTION:   ACCEPT
      FELIX_HEALTHENABLED:                 true
      FELIX_HEALTHPORT:                    9099
      NODENAME:                             (v1:spec.nodeName)
      NAMESPACE:                            (v1:metadata.namespace)
      FELIX_TYPHAK8SNAMESPACE:             calico-system
      FELIX_TYPHAK8SSERVICENAME:           calico-typha
      FELIX_TYPHACAFILE:                   /etc/pki/tls/certs/tigera-ca-bundle.crt
      FELIX_TYPHACERTFILE:                 /node-certs/tls.crt
      FELIX_TYPHAKEYFILE:                  /node-certs/tls.key
      FIPS_MODE_ENABLED:                   false
      FELIX_TYPHACN:                       typha-server
      CALICO_MANAGE_CNI:                   true
      CALICO_IPV4POOL_CIDR:                192.168.0.0/16
      CALICO_IPV4POOL_VXLAN:               CrossSubnet
      CALICO_IPV4POOL_BLOCK_SIZE:          26
      CALICO_IPV4POOL_NODE_SELECTOR:       all()
      CALICO_IPV4POOL_DISABLE_BGP_EXPORT:  false
      CALICO_NETWORKING_BACKEND:           bird
      IP:                                  autodetect
      IP_AUTODETECTION_METHOD:             first-found
      IP6:                                 none
      FELIX_IPV6SUPPORT:                   false
      KUBERNETES_SERVICE_HOST:             10.96.0.1
      KUBERNETES_SERVICE_PORT:             443
    Mounts:
```

Wann calicoctl (Stand 2024/01 calico 3.27)

Für Informationen über die Nodes (z.B. BGP) - direkt auf Node ausführen

- calicoctl get nodes

Um Zusatzinformationen abzufragen, die nur in calicoctl zur Verfügung stehen

```
## namespace in command needs to be written at then end
calicoctl get wep -n namespace-der-application

## get version
calicoctl version

## show cidr / the ippool
```

```
calicoctl ipam show
calicoctl ipam check
```

Calico - only on one of nodes (e.g. controlplane - need to login with ssh)

```
## .kube/config does not need to be configured
calicoctl node status
calicoctl ipam status
```

Calico Default Routing Mode BGP & vxlancrossnet

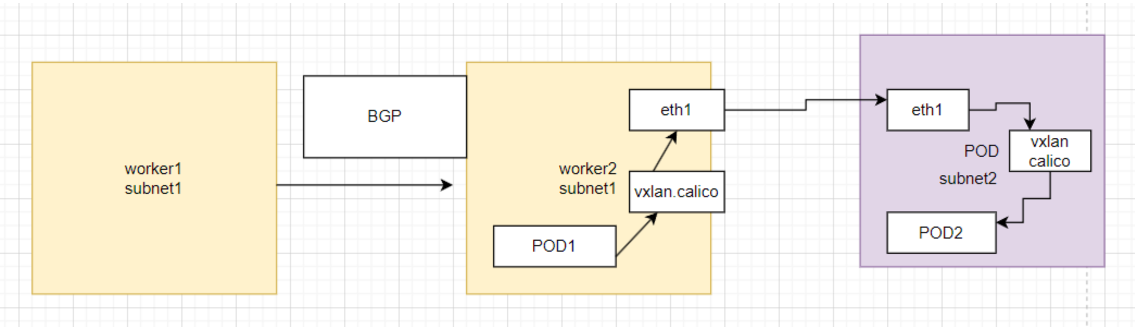
Mode:

- vxlan: crossnet

What does it do ?

- BGP is used, when other node is directly reachable (no restriction through firewalling / networking)
- vxlan is used, when worker node is not directly reachable (using BGP)

Grafics



Detailed Explanation

Was ist `vxlancrossnet` in Calico?

`vxlancrossnet` ist ein **Overlay-Modus von Calico**, der es ermöglicht, Pods auf verschiedenen Layer-3-Netzwerken (z. B. in getrennten Subnetzen oder Clouds) miteinander zu verbinden – **selbst wenn es keine direkte IP-Routen zwischen den Nodes gibt**.

Es funktioniert, indem **VXLAN-Tunnel** aufgebaut werden, um Pod-Verkehr zu kapseln und übers Netzwerk zu transportieren.

Warum braucht man `vxlancrossnet` ?

Standardmäßig funktioniert Calico im **BGP-Modus oder mit IP-IP/VXLAN Overlays**, wenn die Nodes sich gegenseitig routen können.

Aber:

- In **Cloud-Umgebungen**, wo Nodes in **verschiedenen VPCs/Subnetzen** liegen,
- oder in hybriden Szenarien (on-prem + cloud),
- ist manchmal **kein direktes Routing** zwischen den Nodes möglich.

`vxlancrossnet` löst das, indem es Pods trotzdem über ein **VXLAN-Overlay** miteinander kommunizieren lässt.

Beispiel: Cluster mit zwei Nodes in getrennten Subnetzen

Scenario:

- Node A: `10.0.0.10` (Subnetz 10.0.0.0/24)
- Node B: `10.1.0.20` (Subnetz 10.1.0.0/24)
- Kein Routing zwischen den Subnetzen möglich (Security Groups, Firewalls, etc.)
- Ziel: Pod auf Node A soll mit Pod auf Node B kommunizieren

Mit `vxlancrossnet` kapselt Calico den Pod-Traffic in ein VXLAN-Paket und überträgt es zwischen den Nodes – auch ohne Routing zwischen den Subnetzen.

Beispielhafte Konfiguration

Du kannst `vxlancrossnet` in Calico so aktivieren:

```
apiVersion: crd.projectcalico.org/v1
kind: IPPool
metadata:
  name: default-pool
spec:
  cidr: 192.168.0.0/16
```

```
vxlanMode: CrossSubnet  # oder Always
encapsulation: VXLAN
blockSize: 26
natOutgoing: true
nodeSelector: all()
```

How to find out, if this node is used

```
kubectl -n calico-system get ippool -o yaml | grep vxlan
```

Calico cni setting mtu 0

Explanation

In the **CNI (Container Network Interface) config for Calico**, the `"mtu": 0` setting might look confusing at first. Here's what it means:

What does `"mtu": 0` mean?

Setting `"mtu": 0` **tells Calico to automatically detect the MTU** (Maximum Transmission Unit) based on the underlying network interface.

Why would you do that?

Instead of hardcoding a specific MTU (like 1500 or 1450), using `0` delegates the responsibility to Calico's CNI plugin, which will:

- Discover the MTU of the primary network interface on the node.
- Subtract necessary overhead (like VXLAN, WireGuard, IP-in-IP headers, etc., depending on your encapsulation mode).
- Use the appropriate MTU for the workload interfaces to avoid fragmentation issues.

Example Use Case

In a hybrid cluster with nodes running on different types of networks (cloud, on-prem), hardcoding MTU might lead to:

- **Packet fragmentation**
- **Network errors**
- **Unpredictable performance**

By using `"mtu": 0`, Calico adapts automatically.

✅ Summary

Key	Value
`"mtu": 0`	Auto-detect MTU based on host interface
Good for	Avoiding manual config, supporting diverse environments
When to use	Most of the time unless you have a specific MTU requirement

---

Alright! Here's how you can **check what MTU Calico is actually using** on your nodes:

Option 1: Inspect a Calico CNI-created interface inside a pod

Run a pod and check the interface MTU inside it:

```
kubectl run -it --rm test-pod --image=alpine -- sh
```

Then inside the pod:

```
ip link
```

Look for the `eth0` interface — it's usually the one used for communication — and you'll see something like:

```
3: eth0@if5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1440 ...
```

✅ This MTU (`1440` here) is what Calico assigned to the pod's virtual interface, based on the host MTU and overhead.

Option 2: Check on the host what MTU Calico chose

SSH into one of your nodes and look for Calico interfaces:

```
ip link | grep cali
```

You might see:

```
15: cali12345678@if16: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1440 ...
```

That shows the MTU used for the host-side veth of the pod.

Option 3: Look at the actual auto-detected MTU logic in the logs

You can grep the logs of the Calico CNI plugin or Felix:

```
kubectl -n kube-system logs -l k8s-app=calico-node | grep -i mtu
```

Or on some setups:

```
kubectl -n kube-system logs daemonset/calico-node | grep -i mtu
```

You may see lines like:

```
Auto-detected interface eth0 with MTU 1500
Calico CNI MTU set to 1440 (1500 - 60)
```

That confirms both the detected MTU and how it subtracts encapsulation overhead.

If you'd like, I can help you write a little automated script to grab MTU values across all nodes or pods. Want that?

Calico cni setting policy_setup_timeout_seconds

Explanation

In the Calico CNI (Container Network Interface) configuration, the setting:

```
"policy_setup_timeout_seconds": 0
```

refers to the **timeout period (in seconds)** that Calico waits for network policy setup to complete during pod creation.

Here's what it specifically means:

- `policy_setup_timeout_seconds` is a configuration parameter used in Calico's CNI plugin to **control how long the plugin will wait for policy enforcement components to be ready** before allowing the pod networking to proceed.

- When it's set to `0`, it **disables the wait** entirely. In other words, **Calico will not wait** for policy enforcement to be fully ready before returning control to Kubernetes.

Why would you use `0` ?

- **Faster Pod Startup:** This can help speed up pod startup time, especially in scenarios where you don't want pod creation to be blocked by policy setup.
- **Risk Tradeoff:** You might sacrifice some initial enforcement of network policies — the pod might be up and running **before the policies are applied**.

Typical Use Case:

You might set this to `0` :

- In test environments where speed is prioritized over strict policy enforcement.
- If you have other mechanisms ensuring policies are consistently applied.
- To avoid delays during cluster bootstrap or upgrades.

TL;DR:

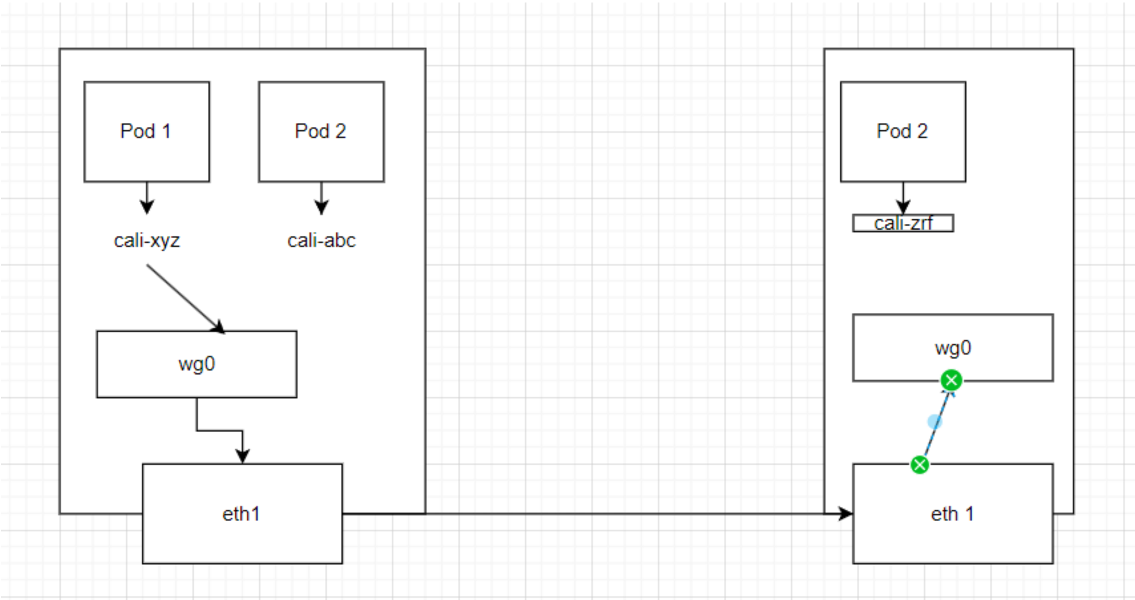
Setting `"policy_setup_timeout_seconds": 0` in the Calico CNI config tells Calico:

> "Don't wait for policies to be set up before finishing pod networking — just go ahead immediately."

Let me know if you want to explore when it might be risky to use this or best practices around it.

Kubernetes calico (CNI-Plugin) - Part 4 -encryption

calico and wirguard



How to activate

Calico + WireGuard

Enable WireGuard

```
kubectl patch felixconfiguration default --type='merge' -p
'{"spec":{"wireguardEnabled":true}}'
```

Fetch the node, see the key...

```
calicoctl get node -o yaml

...
 status:
   ..
   wireguardPublicKey: jlkVyQYooZYzI2wFfNhSZez5eWh44yfq1wKVjLvSXgY=
   ...
```

Reference (Restrictions)

https://docs.tigera.io/calico/latest/network-policy/encrypt-cluster-pod-traffic

Kubernetes multus (Meta-CNI - Plugin)

Multus Überblick

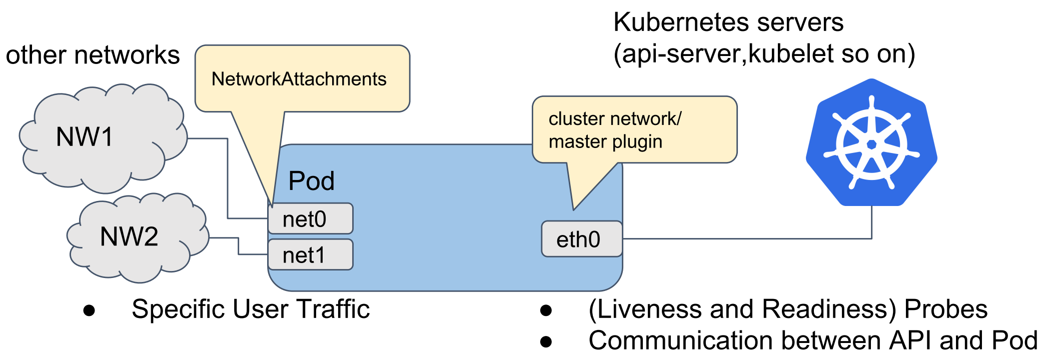
Problem, Warum multus ?

- Aktuell kann seitens kubernetes nur ein Interface verwaltet werden, weil der CNI-Call nur 1x ausgeführt wird. (eigentlich 2x wenn man localhost mit einbezieht)

Prerequisites

- a CNI must be installed beforehand, that manages the network (Calico, Cilium)

Graphics



General

- Multus is a meta-plugin, which makes it possible to attach additional networks to your pod (multi - homing)

macvlan plugin

Example macvlan

- https://github.com/k8snetworkplumbingwg/multus-cni/blob/master/examples/macvlan-pod.yml

```
mkdir -p manifests/multus-example
cd manifests/multus-example
nano network-attachment.yaml
```

```
---
## This net-attach-def defines macvlan-conf with
##   + ips capabilities to specify ip in pod annotation and
##   + mac capabilities to specify mac address in pod annotation
## default gateway is defined as well
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: macvlan-conf
spec:
  config: '{
      "cniVersion": "0.3.1",
      "plugins": [
        {
          "type": "macvlan",
          "capabilities": { "ips": true },
          "master": "eth1",
          "mode": "bridge",
          "ipam": {
            "type": "static",
            "routes": [
              {
                "dst": "0.0.0.0/0",
                "gw": "10.1.1.1"
              }
            ]
          }
        }, {
          "capabilities": { "mac": true },
          "type": "tuning"
        }
      ]
    }'
```

```
kubectl apply -f .
```

```
nano pod.yaml
```

```
## Define a pod with macvlan-conf, defined above, with ip address and mac, and
## "gateway" overrides default gateway to use macvlan-conf's one.
## without "gateway" in k8s.v1.cni.cncf.io/networks, default route will be cluster
## network interface, eth0, even tough macvlan-conf has default gateway config.
apiVersion: v1
```

```
kind: Pod
metadata:
  name: samplepod
  annotations:
    k8s.v1.cni.cncf.io/networks: '[
            { "name": "macvlan-conf",
              "ips": [ "10.1.1.101/24" ],
              "mac": "c2:b0:57:49:47:f1",
              "gateway": [ "10.1.1.1" ]
            }]'
spec:
  containers:
  - name: samplepod
    command: ["/bin/bash", "-c", "trap : TERM INT; sleep infinity & wait"]
    image: dougbtv/centos-network
    ports:
    - containerPort: 80
```

```
kubectl apply -f .
kubectl exec -it samplepod -- sh
```

```
## in der shell
ip a
```

Multus Installation

Schritt 1: Optional: View settings of cni before installing multus

```
kubectl debug -it node/worker1 --image=alpine
```

```
## in der bash
apk add jq
cat /host/etc/cni/net.d/10-calico.conflist | jq
```

```
## Ausgabe

/ # cat /host/etc/cni/net.d/10-calico.conflist | jq
{
  "name": "k8s-pod-network",
  "cniVersion": "0.3.1",
  "plugins": [
    {
      "container_settings": {
        "allow_ip_forwarding": false
      },
      "datastore_type": "kubernetes",
      "endpoint_status_dir": "/var/run/calico/endpoint-status",
      "ipam": {
        "assign_ipv4": "true",
        "assign_ipv6": "false",
        "type": "calico-ipam"
      },
      "kubernetes": {
        "k8s_api_root": "https://10.96.0.1:443",
        "kubeconfig": "/etc/cni/net.d/calico-kubeconfig"
      },
      "log_file_max_age": 30,
      "log_file_max_count": 10,
      "log_file_max_size": 100,
      "log_file_path": "/var/log/calico/cni/cni.log",
      "log_level": "Info",
      "mtu": 0,
      "nodename_file_optional": false,
      "policy": {
        "type": "k8s"
      },
      "policy_setup_timeout_seconds": 0,
      "type": "calico"
    },
    {
      "capabilities": {
        "bandwidth": true
      },
      "type": "bandwidth"
    },
    {
      "capabilities": {
```

```
        "portMappings": true
      },
      "snat": true,
      "type": "portmap"
    }
  ]
}
```

Schritt 2: Installation mit Helm

```
kubectl apply -f https://raw.githubusercontent.com/k8snetworkplumbingwg/multus-cni/master/deployments/multus-daemonset-thick.yml
```

```
## Überprüfung
kubectl debug -it node/worker1 --image=alpine
```

```
## in der bash
apk add jq

## das wird als erstes geladen
cat /host/etc/cni/net.d/00-multus.conf | jq
```

sr-iov mit multus

Voraussetzung: Multus:

Konzept SR-IOV

- Direkte Hardwareanbindung der Netzwerkkarte
- Offload wird auf Netzwerkkarte gemacht (nicht im Kernel)
- bessere Performance

Generell

- Erweiterung des PCI-Express Standards
- Eine Netzwerkkarte wird mehrmals angeboten und Kommunikation erfolgt direkt und nicht über den Umweg Kernel

Vorbereitung

- https://github.com/k8snetworkplumbingwg/sriov-network-device-plugin
- https://github.com/k8snetworkplumbingwg/sriov-network-device-plugin/tree/db98d96cc0d6ad3fff917ba238bd1cc5cc3f7e82#config-parameters

Einbindung

- https://github.com/k8snetworkplumbingwg/sriov-network-device-plugin#example-deployments
- https://github.com/k8snetworkplumbingwg/multus-cni/blob/master/examples/sriov-pod.yml

Zufällige Auswahl von Network-Attachements-Definition für srv-iov

Yes, it's technically possible to **autodetect SR-IOV NICs** on a Kubernetes node and dynamically configure a **Multus `NetworkAttachmentDefinition`** to attach a pod to a randomly selected one — **but not entirely natively** or dynamically at runtime without help.

Let's break it down:

SR-IOV Interface Detection

You can detect all SR-IOV capable NICs on a node using tools like:

```
lspci | grep -i ether
```

Or more specifically:

```
ls /sys/class/net | while read iface; do
  if [ -d "/sys/class/net/$iface/device/sriov_totalvfs" ]; then
    echo "$iface is SR-IOV capable"
  fi
done
```

The `sriov-device-plugin` used with Kubernetes typically discovers and advertises these VFs to the Kubelet.

Multus + SR-IOV

Multus uses `NetworkAttachmentDefinition` (NAD) with SR-IOV by referencing a resource (like `intel.com/sriov_netdevice`). This ties into the `sriov-network-device-plugin`, which assigns Virtual Functions (VFs) to pods based on availability.

But **you can't directly say "randomly pick an SR-IOV interface" in the NAD JSON**. It needs a defined resource name and configuration — unless you automate generating NADs.

So how do you make this dynamic?

Option 1: Dynamic NAD Generator (Custom Controller or Admission Webhook)

- Create a custom mutating webhook or initContainer that:

- Detects all available SR-IOV devices on the node.
    - Randomly picks one.
    - Generates a NAD dynamically (with unique name) or patches the pod spec to refer to the chosen one.

Option 2: Predefine Multiple NADs

- Define multiple NADs manually, each bound to a specific SR-IOV interface.
- In your app logic or scheduling controller, randomly select one when creating the pod.

Option 3: Use SR-IOV device plugin with resource pool

- Let the `sriov-network-operator` manage pools (e.g., via `SriovNetworkNodePolicy`) and abstract the devices.
- Pods just request a resource (e.g., `intel.com/sriov_netdevice`) and the device plugin picks from the available pool.

This is not *truly random per pod*, but it's **automatically distributed** if you're okay with that.

✅ TL;DR

- **Auto-detecting SR-IOV NICs** = easy via sysfs or `sriov-network-device-plugin`.
- **Random assignment in Multus NAD** = not directly supported.
- **Solutions**:
    - Use device plugin + pool (simpler).
    - Generate NADs dynamically via custom logic (more flexible).

Perfect — you're aiming for **random selection of a specific SR-IOV VF per pod**, which is a bit outside of what Multus and the SR-IOV device plugin do by default. But with a bit of Kubernetes magic, we can make it work.

Here's a practical **design pattern** to achieve this:

Strategy for Random SR-IOV NIC Selection Per Pod

1. Predefine NADs for Each SR-IOV Interface

- You create one `NetworkAttachmentDefinition` per physical interface (e.g., `sriov-net-ens1f0`, `sriov-net-ens2f0`, etc.)
- Each NAD refers to a specific `resourceName` exposed by the SR-IOV device plugin.

Example NAD for `ens1f0`:

```
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  name: sriov-net-ens1f0
spec:
  config: '{
    "type": "sriov",
    "name": "sriov-net-ens1f0",
    "ipam": {
      "type": "static"
    },
    "deviceID": "0000:01:00.1"  # optional if resource pool used
  }'
```

---

2. Custom Init Container or Admission Webhook to Inject a Random NAD

Since Kubernetes doesn't support "pick a random NAD" natively, your workaround is:

Option A: Init Container + Downward API + Host Mount

1. Mount a host path (e.g., `/etc/cni/net.d`) into the pod as read-only.
2. Have an **init container** (or even an entrypoint script) that:
    - Lists available NADs (e.g., via Kubernetes API).
    - Picks one at random.
    - Patches the pod's annotations to include it (if running in privileged mode + with necessary permissions).
    - Or writes the choice to an environment file used by the main container.

Option B: Mutating Admission Webhook

This is more elegant but more complex:

- Write a small webhook service (e.g., in Go or Python) that intercepts Pod creation.
- At admission time, it:
    - Queries available NADs matching a pattern (e.g., label or name prefix).
    - Picks one at random.
    - Adds the `k8s.v1.cni.cncf.io/networks` annotation with the chosen NAD.

This works seamlessly and is transparent to your apps.

---

3. Deploy Your Pods With a Generic NAD Label

You only need to request something like this:

```
metadata:
  annotations:
    k8s.v1.cni.cncf.io/networks: sriov-net-<RANDOM>
```

The actual `<RANDOM>` part gets injected by your webhook or entrypoint logic.

Tools You'll Need

- Kubernetes client (kubectl or client library)
- Permissions to list NADs (CRD)
- Optional: controller/webhook framework like Kubebuilder or Kopf (Python)
- Custom script or binary to pick the NAD at runtime

✅ Benefits

- Full control and randomness
- Works even with static SR-IOV assignments
- Extensible for other policies (e.g., "least used")

⚠ Caveats

- You'll need a controller/webhook or privileged init logic.
- NADs must be kept in sync with actual node hardware and VF pools.
- If you're using the `sriov-network-operator`, you may have to manually break up your `SriovNetworkNodePolicy` into per-interface pools.

Want a ready-made Python or Bash script to randomly pick a NAD and patch the pod? Or a small webhook template in Go/Python?

Awesome! Here's a **ready-made Bash script** that randomly selects a `NetworkAttachmentDefinition` (NAD) from a specific namespace and outputs it in a way your Pod or init container can use.

✅ What this script does:

- Lists all NADs in a namespace (with optional prefix filter).
- Randomly picks one.
- Writes it to a file or echoes it (depending on how you want to use it).

`pick-random-nad.sh`

```
##!/bin/bash

## CONFIGURABLE VARIABLES
NAMESPACE="${NAD_NAMESPACE:-default}"        # Namespace where NADs are defined
PREFIX="${NAD_PREFIX:-sriov-net-}"           # Filter NADs with a certain prefix
OUTPUT_FILE="${OUTPUT_FILE:-/tmp/selected_nad}"  # File to store result

## REQUIRE: kubectl installed and available in container
if ! command -v kubectl &> /dev/null; then
    echo "kubectl not found in PATH"
    exit 1
fi

## Fetch list of matching NADs
NADS=$(kubectl get network-attachment-definitions -n "$NAMESPACE" -o jsonpath="{.items[*].metadata.name}" | tr ' ' '\n' | grep
"^${PREFIX}")

## Check if we found anything
if [ -z "$NADS" ]; then
    echo "No NADs found in namespace '$NAMESPACE' with prefix '$PREFIX'"
    exit 1
fi

## Pick one at random
SELECTED_NAD=$(echo "$NADS" | shuf -n1)

echo "Selected NAD: $SELECTED_NAD"

## Save to file or stdout
echo "$SELECTED_NAD" > "$OUTPUT_FILE"
```

Example Usage in an Init Container (conceptual)

```
initContainers:
  - name: pick-nad
    image: bitnami/kubectl:latest
    command: ["/bin/bash", "-c"]
    args:
      - |
        chmod +x /scripts/pick-random-nad.sh
        /scripts/pick-random-nad.sh
    env:
      - name: NAD_NAMESPACE
```

```yaml
          value: "default"
        - name: NAD_PREFIX
          value: "sriov-net-"
      volumeMounts:
        - name: scripts
          mountPath: /scripts
        - name: shared
          mountPath: /nad

  containers:
    - name: main-app
      image: your-app
      env:
        - name: NAD_NAME
          valueFrom:
            configMapKeyRef:
              name: nad-config
              key: selected_nad  # Or mount shared volume if using file
      volumeMounts:
        - name: shared
          mountPath: /nad
  volumes:
    - name: scripts
      configMap:
        name: pick-nad-script
    - name: shared
      emptyDir: {}
```

Permissions Needed

Your service account (or init container) needs permission to:

```yaml
apiGroups: ["k8s.cni.cncf.io"]
resources: ["network-attachment-definitions"]
verbs: ["list"]
```

Awesome — let's build a **Go-based mutating admission webhook** that adds a random SR-IOV NAD to pod annotations at creation time.

Overview

- **Language**: Go
- **Framework**: `controller-runtime` (standard for webhooks/controllers)
- **Function**: Inject `k8s.v1.cni.cncf.io/networks: sriov-net-<random>` into pods missing that annotation
- **Behavior**: Only mutate pods that match a label (e.g., `inject-sriov: "true"`)
- **Deployment**: Webhook service + `MutatingWebhookConfiguration` + TLS certs

Project Structure

```
sriov-webhook/
├── main.go
├── webhook.go
├── Dockerfile
├── config/
│   ├── webhook.yaml
│   └── certs/
```

Core Go Code

`main.go`

```go
package main

import (
    "flag"
    "fmt"
    "log"
    "net/http"
    "os"

    "sriov-webhook/webhook"
)

func main() {
    var certFile, keyFile string
    flag.StringVar(&certFile, "tls-cert-file", "/certs/tls.crt", "Path to TLS certificate file")
```

```go
    flag.StringVar(&keyFile, "tls-key-file", "/certs/tls.key", "Path to TLS key file")
    flag.Parse()

    mux := http.NewServeMux()
    mux.HandleFunc("/mutate", webhook.HandlePodMutate)

    log.Println("  Starting webhook server on port 8443...")
    server := &http.Server{
        Addr:    ":8443",
        Handler: mux,
    }
    err := server.ListenAndServeTLS(certFile, keyFile)
    if err != nil {
        log.Fatalf("✕ Failed to start server: %v", err)
    }
}
```

`webhook.go`

```go
package webhook

import (
    "encoding/json"
    "fmt"
    "io/ioutil"
    "math/rand"
    "net/http"
    "time"

    admissionv1 "k8s.io/api/admission/v1"
    corev1 "k8s.io/api/core/v1"
)

var nadList = []string{
    "sriov-net-ens1f0", "sriov-net-ens2f0", "sriov-net-ens3f0",
}

func HandlePodMutate(w http.ResponseWriter, r *http.Request) {
    body, err := ioutil.ReadAll(r.Body)
    if err != nil {
        http.Error(w, "unable to read request", http.StatusBadRequest)
        return
    }

    review := admissionv1.AdmissionReview{}
    if err := json.Unmarshal(body, &review); err != nil {
        http.Error(w, "unable to parse admission review", http.StatusBadRequest)
        return
    }

    pod := corev1.Pod{}
    if err := json.Unmarshal(review.Request.Object.Raw, &pod); err != nil {
        http.Error(w, "unable to unmarshal pod", http.StatusBadRequest)
        return
    }

    if pod.ObjectMeta.Annotations["k8s.v1.cni.cncf.io/networks"] != "" {
        respond(w, &review, nil) // already has NAD
        return
    }

    if pod.ObjectMeta.Labels["inject-sriov"] != "true" {
        respond(w, &review, nil) // label not matched
        return
    }

    // Pick random NAD
    rand.Seed(time.Now().UnixNano())
    randomNAD := nadList[rand.Intn(len(nadList))]

    patch := []map[string]interface{}{
        {
            "op":    "add",
            "path":  "/metadata/annotations/k8s.v1.cni.cncf.io~1networks",
            "value": randomNAD,
        },
    }
```

```go
    patchBytes, _ := json.Marshal(patch)

    review.Response = &admissionv1.AdmissionResponse{
        UID:     review.Request.UID,
        Allowed: true,
        Patch:   patchBytes,
        PatchType: func() *admissionv1.PatchType {
            pt := admissionv1.PatchTypeJSONPatch
            return &pt
        }(),
    }

    respBytes, _ := json.Marshal(review)
    w.Header().Set("Content-Type", "application/json")
    w.Write(respBytes)
}

func respond(w http.ResponseWriter, review *admissionv1.AdmissionReview, patch []byte) {
    review.Response = &admissionv1.AdmissionResponse{
        UID:     review.Request.UID,
        Allowed: true,
    }
    respBytes, _ := json.Marshal(review)
    w.Header().Set("Content-Type", "application/json")
    w.Write(respBytes)
}
```

Dockerfile

```dockerfile
FROM golang:1.21 as builder
WORKDIR /app
COPY . .
RUN go build -o sriov-webhook .

FROM alpine
WORKDIR /root/
COPY --from=builder /app/sriov-webhook .
COPY config/certs /certs
CMD ["./sriov-webhook", "-tls-cert-file=/certs/tls.crt", "-tls-key-file=/certs/tls.key"]
```

`MutatingWebhookConfiguration` (snippet)

```
mkdir -p manifests/webhook-sr-iov
cd manifests /web-sr-iov
nano 01-webhook.yaml
```

```yaml
apiVersion: admissionregistration.k8s.io/v1
kind: MutatingWebhookConfiguration
metadata:
  name: sriov-nad-injector
webhooks:
  - name: nad-injector.sriov.svc.cluster.local
    rules:
      - apiGroups: [""]
        apiVersions: ["v1"]
        operations: ["CREATE"]
        resources: ["pods"]
    clientConfig:
      service:
        name: sriov-webhook
        namespace: your-namespace
        path: /mutate
      caBundle: <base64-cabundle>
    admissionReviewVersions: ["v1"]
    sideEffects: None
```

```
kubectl apply -f .
```

Deploying

You'll need:

- TLS certs (can use `cert-manager` or Kubernetes CA)

- A `Deployment` + `Service` for the webhook
- RBAC (if needed to query real NADs from the API)

```
Perfect! Here's a helper script that:

1. Generates a **self-signed TLS certificate** for the webhook service.
2. Base64-encodes the **CA bundle**.
3. Prepares it for inclusion in the `MutatingWebhookConfiguration`.

---

###   `generate-tls.sh`

```bash
##!/bin/bash

## Configurable values
NAMESPACE="sriov-webhook"
SERVICE="sriov-webhook"
SECRET_NAME="sriov-webhook-tls"

## Create certs directory
mkdir -p certs && cd certs

## Generate CA
openssl genrsa -out ca.key 2048
openssl req -x509 -new -nodes -key ca.key -subj "/CN=SRIOV Webhook CA" -days 365 -out ca.crt

## Generate server key and CSR
openssl genrsa -out tls.key 2048
openssl req -new -key tls.key -subj "/CN=${SERVICE}.${NAMESPACE}.svc" -out server.csr

## Generate server certificate signed by our CA
cat > csr.conf <<EOF
[ v3_ext ]
authorityKeyIdentifier=keyid,issuer
basicConstraints=CA:FALSE
keyUsage = digitalSignature, nonRepudiation, keyEncipherment
extendedKeyUsage = serverAuth
subjectAltName = @alt_names

[ alt_names ]
DNS.1 = ${SERVICE}
DNS.2 = ${SERVICE}.${NAMESPACE}
DNS.3 = ${SERVICE}.${NAMESPACE}.svc
EOF

openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key -CAcreateserial \
    -out tls.crt -days 365 -extensions v3_ext -extfile csr.conf

## Create Kubernetes TLS secret
kubectl create secret tls ${SECRET_NAME} \
  --cert=tls.crt --key=tls.key \
  -n ${NAMESPACE}

## Output base64-encoded CA bundle for webhook
echo
echo "  Copy this base64-encoded CA bundle into your MutatingWebhookConfiguration:"
echo
base64 -w 0 ca.crt
echo
```

How to Use

1. Save it as `generate-tls.sh`
2. Run: `chmod +x generate-tls.sh && ./generate-tls.sh`
3. Copy the printed base64 `caBundle` into your `MutatingWebhookConfiguration.yaml` here:

```yaml
clientConfig:
  caBundle: <PASTE_CA_BUNDLE_HERE>
```

```
## Kubernetes coil (egress - gateway)
```
```

```
### coil


### Opt-In egress-gateway (NAT-Service)
```

apiVersion: coil.cybozu.com/v2 kind: Egress metadata: namespace: internet-egress name: nat spec: replicas: 2 destinations:

- 0.0.0.0/0
- ::/0

```

  * Not all Pods become the client of Egress. To become a client, Pods need to have special annotations like this:
```

apiVersion: v1 kind: Pod metadata: namespace: default name: nat-client annotations: egress.coil.cybozu.com/internet-egress: nat spec:

```

### Reference

  * Refs: https://blog.kintone.io/entry/coilv2
  * https://github.com/cybozu-go/coil


## Kubernetes NetworkPolicy

### Einfache Übung Network Policy


### Schritt 1: Deployment und Service erstellen
```

KURZ=jm kubectl create ns policy-demo-$KURZ

```
```

cd mkdir -p manifests cd manifests mkdir -p np cd np

```
```

nano 01-deployment.yml

```
```

apiVersion: apps/v1 kind: Deployment metadata: name: nginx-deployment spec: selector: matchLabels: app: nginx replicas: 1 template: metadata: labels: app: nginx spec: containers: - name: nginx image: nginx:1.23 ports: - containerPort: 80

```
```

kubectl -n policy-demo-$KURZ apply -f .

```
```

nano 02-service.yml

```
```

apiVersion: v1 kind: Service metadata: name: nginx spec: type: ClusterIP # Default Wert ports:

- port: 80 protocol: TCP selector: app: nginx

```
```

kubectl -n policy-demo-$KURZ apply -f .

```

### Schritt 2: Zugriff testen ohne Regeln
```

lassen einen 2. pod laufen mit dem auf den nginx zugreifen

kubectl run --namespace=policy-demo-$KURZ access --rm -ti --image busybox

```
```

innerhalb der shell

wget -q nginx -O -

```
```

Optional: Pod anzeigen in 2. ssh-session zu jump-host

kubectl -n policy-demo-$KURZ get pods --show-labels

```
### Schritt 3: Policy festlegen, dass kein Zugriff erlaubt ist.
```

nano 03-default-deny.yaml

```
```

Schritt 2: Policy festlegen, dass kein Ingress-Traffic erlaubt

in diesem namespace: policy-demo-$KURZ

kind: NetworkPolicy apiVersion: networking.k8s.io/v1 metadata: name: default-deny spec: podSelector: matchLabels: {}

```
```

kubectl -n policy-demo-$KURZ apply -f .

```
### Schritt 3.5: Verbindung mit deny all Regeln testen
```

kubectl run --namespace=policy-demo-$KURZ access --rm -ti --image busybox

```
```

innerhalb der shell

wget -q nginx -O -

```
### Schritt 4: Zugriff erlauben von pods mit dem Label run=access (alle mit run gestarteten pods mit namen access haben dieses label per default)
```

nano 04-access-nginx.yaml

```
```

apiVersion: networking.k8s.io/v1 kind: NetworkPolicy metadata: name: access-nginx spec: podSelector: matchLabels: app: nginx ingress: - from: - podSelector: matchLabels: run: access

```
```

kubectl -n policy-demo-$KURZ apply -f .

```
### Schritt 5: Testen (zugriff sollte funktionieren)
```

lassen einen 2. pod laufen mit dem auf den nginx zugreifen

pod hat durch run -> access automatisch das label run:access zugewiesen

kubectl run --namespace=policy-demo-$KURZ access --rm -ti --image busybox

```
```

innerhalb der shell

wget -q nginx -O -

```
### Schritt 6: Pod mit label run=no-access - da sollte es nicht gehen
```

kubectl run --namespace=policy-demo-$KURZ no-access --rm -ti --image busybox

```
```

in der shell

wget -q nginx -O -

```
### Schritt 7: Aufräumen
```

kubectl delete ns policy-demo-$KURZ

```
### Ref:

  * https://projectcalico.docs.tigera.io/security/tutorials/kubernetes-policy-basic

## Calico NetworkPolicy

### Protecting Services


### Example
```

apiVersion: projectcalico.org/v3 kind: GlobalNetworkPolicy metadata: name: allow-cluster-ips spec: selector: k8s-role == 'node' types:

- Ingress applyOnForward: true preDNAT: true ingress:

Allow 50.60.0.0/16 to access Cluster IP A

- action: Allow source: nets:
  - 50.60.0.0/16 destination: nets:
  - 10.20.30.40/32 Cluster IP A

Allow 70.80.90.0/24 to access Cluster IP B

- action: Allow source: nets:
  - 70.80.90.0/24 destination: nets:
  - 10.20.30.41/32 Cluster IP B

```
### Referenz

  * https://docs.tigera.io/calico/latest/network-policy/services/services-cluster-ips

### Calico Logging Firewall Rules


### General

 * NetworkPolicy of Kubernetes does not provide possibility to track

### Solutions

  * Use NetworkPolicy from calico (to apply it with kubectl - the calico api server needs to be installed) / or use calicoctl
  * Enable Tracing
  * Use: https://kubernetes.io/blog/2019/04/19/introducing-kube-iptables-tailer/

### Solution 1: NetworkPolicy calico

  * https://github.com/projectcalico/calico/issues/4344


### Logs
```

Normally you should see it with (on the right kubernetes node)

cat /var/log/syslog | grep calico-packet

This is how a syslog entry looks like

Here is a example (default) Log: Apr 3 10:12:30 aks-workerpool1-13987120-vmss000000 kernel: [10821.860593] calico-packet: IN=calic440f455693 OUT=eth0 MAC=ee:ee:ee:ee:ee:ee:f2:f8:09:3d:97:03:08:00 SRC=10.244.2.7 DST=8.8.8.8 LEN=84 TOS=0x00 PREC=0x00 TTL=63 ID=33536 DF PROTO=ICMP TYPE=8 CODE=0 ID=32113 SEQ=43

```
### Walkthrough
```

cd mkdir -p manifests cd manifests mkdir pol2 cd pol2 vi 01-pod.yaml

```

```

apiVersion: v1 kind: Pod metadata: name: static-web labels: app: web spec: containers: - name: web image: nginx ports: - name: web containerPort: 80 protocol: TCP vi 02-pol.yaml

apiVersion: projectcalico.org/v3 kind: NetworkPolicy metadata: name: log spec: selector: app == 'web' types:

- Ingress
- Egress ingress:
- action: Log egress:
- action: Log
- action: Deny

kubectl apply -f .

find the node, where it runs on

kubectl get pods -o wide

login to that node with ssh (kubernetes node)

e.g. ssh user@node

switch to root: sudo su -

tail -f /var/log/syslog | grep calico-packet

or

journalctl -f | grep calico-packet

now open a debug pod

kubectl debug -it static-web --image=busybox

in pod ping - this will not work, because we cannot retrieve dns

ping www.google.de

watch output from other node in the meanwhile

```
### Reference

  * Eventually set a prefix for logging:
  * https://docs.tigera.io/calico-cloud/visibility/iptables

### Exercise calico Network Policy


### Step 1: Set global policy
```

apiVersion: crd.projectcalico.org/v1 kind: GlobalNetworkPolicy metadata: name: default-deny spec: namespaceSelector: kubernetes.io/metadata.name != "kube-system" types:

- Ingress
- Egress egress:

allow all namespaces to communicate to DNS pods

- action: Allow protocol: UDP destination: selector: 'k8s-app == "kube-dns"' ports:
  - 53
- action: Allow protocol: TCP destination: selector: 'k8s-app == "kube-dns"' ports:
  - 53

kubectl apply -f .

```
### Step 2: nginx ausrollen aus manifests/04-service und testen
```

cd mkdir -p manifests cd manifests mkdir 04-service cd 04-service

nano deploy.yml

apiVersion: apps/v1 kind: Deployment metadata: name: web-nginx spec: selector: matchLabels: web: my-nginx replicas: 2 template: metadata: labels: web: my-nginx spec: containers: - name: cont-nginx image: nginx ports: - containerPort: 80

nano service.yml

apiVersion: v1 kind: Service metadata: name: svc-nginx labels: run: svc-my-nginx spec: type: ClusterIP ports:

- port: 80 protocol: TCP selector: web: my-nginx

kubectl apply -f .

kubectl run -it --rm access --image=busybox

In der Bbusybox

wget -O - http://svc-nginx

```
### Step 3: Traffic erlauben egress von busybox
```

cd cd manifests mkdir cnp cd cnp

vi 02-egress-allow-busybox.yml

apiVersion: crd.projectcalico.org/v1 kind: NetworkPolicy metadata: name: allow-busybox-egress spec: selector: run == 'access' types:

- Egress egress:
- action: Allow

kubectl apply -f .

kubectl run -it --rm access --image=busybox

sollte gehen

wget -O - http://www.google.de

sollte nicht funktionieren

wget -O - http://my-nginx

```
### Step 4: Traffic erlauben für nginx
```

03-allow-ingress-my-nginx.yml

apiVersion: crd.projectcalico.org/v1 kind: NetworkPolicy metadata: name: allow-nginx-ingress spec: selector: run == 'my-nginx' types:

- Ingress ingress:
- action: Allow source: selector: run == 'access'

```

```

kubectl apply -f .

```

```

kubectl run -it --rm access --image=busybox

```

```

In der Bbusybox

wget -O - http://my-nginx

```
## Kubernetes Scheduler

### Koordinator scheduler, that also network bandwith into account


### Explanation

Yes, Koordinator supports that.

To configure **Koordinator** for **network-aware scheduling** in your Kubernetes cluster, you can follow these steps:

---

#### 1. **Install Koordinator**

Ensure that Koordinator is installed in your cluster. If it's not already installed, you can add the Koordinator Helm repository
and install it using Helm:


```bash
helm repo add koordinator https://koordinator-sh.github.io/charts/
helm repo update
helm install koordinator koordinator/koordinator --namespace koordinator-system --create-namespace
```

---

2. Enable Load-Aware Scheduling

Koordinator's **Load-Aware Scheduling** feature allows the scheduler to consider real-time node metrics, including CPU and memory usage, to make informed scheduling decisions. This feature is enabled by default. However, you can customize its behavior by modifying the `koord-scheduler-config` ConfigMap.

To customize the configuration:

- Edit the `koord-scheduler-config` ConfigMap:

  ```
  kubectl edit configmap koord-scheduler-config -n koordinator-system
  ```

- In the `data.koord-scheduler-config` section, you can configure thresholds and weights for resources, influencing how the scheduler balances loads across nodes. For example:

  ```yaml
  apiVersion: kubescheduler.config.k8s.io/v1beta2
  kind: KubeSchedulerConfiguration
  profiles:
    - schedulerName: koord-scheduler
      plugins:
        filter:
          enabled:
            - name: LoadAwareScheduling
        score:
          enabled:
            - name: LoadAwareScheduling
              weight: 1
        reserve:
          enabled:
            - name: LoadAwareScheduling
      pluginConfig:
        - name: LoadAwareScheduling
          args:
            # Configure thresholds and weights here
  ```
  ```

This configuration enables the `LoadAwareScheduling` plugin and allows you to set specific thresholds and weights. For more detailed information on configuring these parameters, refer to the Koordinator Load-Aware Scheduling Documentation.

---

3. Annotate Nodes with Custom Usage Thresholds (Optional)

For more granular control, you can set specific load thresholds on individual nodes using annotations. By adding the `scheduling.koordinator.sh/usage-thresholds` annotation to a node, you can define custom utilization thresholds that Koordinator will consider during scheduling.

For example, to set a CPU usage threshold of 80% and a memory usage threshold of 70% on a node:

```
kubectl annotate node <node-name> scheduling.koordinator.sh/usage-thresholds='{"usageThresholds":{"cpu":80,"memory":70}}'
```

Replace `<node-name>` with the name of your node. This annotation helps Koordinator make more informed scheduling decisions based on the real-time load of each node. Detailed instructions can be found in the Koordinator Load-Aware Scheduling Documentation.

---

4. Deploy Pods Using Koordinator's Scheduler

When deploying your applications, specify `koord-scheduler` as the scheduler in your Pod or Deployment manifests to utilize Koordinator's scheduling capabilities.

Here's an example of a Deployment manifest:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: example
  template:
    metadata:
      labels:
        app: example
    spec:
      schedulerName: koord-scheduler
      containers:
        - name: example-container
          image: example-image
          ports:
            - containerPort: 80
```

In this manifest:

- The `schedulerName: koord-scheduler` line directs Kubernetes to use Koordinator's scheduler for this Deployment.
- The rest of the manifest defines a typical Deployment with three replicas of a container named `example-container`.

By setting the `schedulerName` to `koord-scheduler`, Koordinator will handle the scheduling of these Pods, taking into account network traffic and other factors to optimize performance. For more information on configuring multiple schedulers, refer to the Kubernetes Documentation.

Why is CPU pinning not used as default in kubernetes scheduler anymore ?

The basics

CPU pinning (or **CPU affinity**) is a technique where specific processes or threads are bound to run on specific CPUs (cores), rather than being scheduled on any available CPU by the OS scheduler. This can improve **performance and cache locality** by keeping workloads on the same CPU and reducing context switching or cache misses.

How CPU Pinning Works

When CPU pinning is enabled:

- The OS scheduler is told to **restrict** a process/thread to a specific CPU (or a set of CPUs).
- For example: process A → only runs on CPU 0, process B → only on CPU 1.
- In Linux, tools like `taskset`, `cset`, or `cpuset cgroups` can enforce CPU pinning.
- In containers, it's handled via cgroups and `cpuset` settings, often via the container runtime or orchestration platform like Kubernetes.

Why Was CPU Pinning Used?

- **Real-time systems**: where latency and predictability are critical.
- **Performance tuning**: for applications that are CPU-bound and benefit from avoiding cache thrashing or migration.
- **NUMA-aware scheduling**: pinning threads to specific NUMA nodes can help optimize memory access patterns.

Why Kubernetes Doesn't Use CPU Pinning by Default Anymore

Kubernetes used to support pinning via the `cpuManagerPolicy=static`, which could align containers to specific cores when `Guaranteed` QoS was used.

However, **it's not the default**, and here's why:

1. **Complexity vs. Benefit**:
    - Managing CPU topology at scale is **complex and brittle**.

- In dynamic workloads (which Kubernetes excels at), strict pinning can lead to **resource fragmentation** and underutilization.

2. **Reduced Flexibility**:
   - Pinning reduces the OS scheduler's ability to optimize CPU time across workloads.
   - It limits Kubernetes' ability to reschedule or balance pods dynamically.
3. **Overhead in Mixed Workloads**:
   - For non-latency-critical workloads, the gains are often negligible.
   - Kubernetes prefers **isolation** via cgroups (`cpu shares` , `quota` , etc.), which is more flexible.
4. **Not Portable**:
   - CPU pinning is **highly node-specific**.
   - Kubernetes prioritizes portability, so default behavior avoids assuming specific CPU layouts.

When You *Should* Use CPU Pinning in Kubernetes

CPU pinning *is* still possible in Kubernetes with the right settings:

- Use `Guaranteed` QoS class (i.e., requests = limits for CPU).
- Enable `--cpu-manager-policy=static` on the kubelet.
- Optionally configure `TopologyManager` and `NUMA` alignment.

✅ Recommended for:

- Real-time or latency-sensitive apps (e.g., telco 5G workloads)
- HPC (high-performance computing)
- Some AI/ML inference workloads

Let's break that down — it's a key point in understanding *why* Kubernetes doesn't use strict CPU pinning by default.

What are Dynamic Workloads?

In Kubernetes, **dynamic workloads** means:

- Pods **come and go frequently** (auto-scaling, job runners, CI/CD pipelines, etc.).
- Cluster **resources are shared** among many teams or services.
- Kubernetes constantly tries to **optimize usage**: bin-packing, rescheduling, scaling up/down.

What Does "Strict CPU Pinning" Do?

Strict CPU pinning means:

- A container gets locked to **specific CPU cores**.
- Those cores are reserved **exclusively** for that pod — no sharing.
- Even if the pod is doing **nothing**, no one else can use those cores.

Now: What is Resource Fragmentation?

Imagine you have a 4-core node:

Core	Available?	Pinned to
0	No	Pod A
1	No	Pod B
2	Yes	Free
3	Yes	Free

Now another pod (Pod C) comes in and wants 2 CPUs — but **only 2 unpinned cores are available**.

✅ Kubernetes sees 2 free CPUs overall
But because cores 0 & 1 are pinned (and not shared), Pod C **can't be scheduled here**, even though the node is only 50% utilized!

That's **resource fragmentation**: the **available CPU is split into unusable chunks**.

What is Underutilization?

Same scenario:
Let's say Pod A and Pod B are mostly idle — using only ~5% CPU each.

But because they've *pinned* their CPUs, no other pod is allowed to use cores 0 or 1.

Result?

- Cores 0 and 1 are **mostly idle**.
- The node looks **"full"** to the scheduler.
- Meanwhile, other nodes may be **overloaded**.

That's **underutilization** — you're **not getting the most out of your hardware**.

⚖️ Why Kubernetes Avoids This

Kubernetes prefers **soft isolation** via cgroups:

- Containers get CPU *shares* or *quotas*, not full core reservations.
- The OS can dynamically move workloads around.
- Kubernetes can fit more workloads per node and reschedule things efficiently.

So unless you're in a **real-time** or **deterministic** use case (like telco or industrial systems), strict pinning hurts the **efficiency and elasticity** that Kubernetes is designed for.

Helm (Kubernetes Paketmanager)

Helm Grundlagen

Wo ?

```
artifacts helm
```

- https://artifacthub.io/

Komponenten

```
Chart - beeinhaltet Beschreibung und Komponenten
tar.gz - Format
oder Verzeichnis

Wenn wir ein Chart ausführen wird eine Release erstellen
(parallel: image -> container, analog: chart -> release)
```

Installation

```
## Beispiel ubuntu
## snap install --classic helm

## Cluster muss vorhanden, aber nicht notwendig wo helm installiert

## Voraussetzung auf dem Client-Rechner (helm ist nichts als anderes als ein Client-Programm)
Ein lauffähiges kubectl auf dem lokalen System (welches sich mit dem Cluster verbinden kann).
-> saubere -> .kube/config

## Test
kubectl cluster-info
```

Helm Warum ?

```
Ein Paket für alle Komponenten
Einfaches Installieren, Updaten und deinstallieren
Feststehende Struktur
```

Helm Example

Prerequisites

- kubectl needs to be installed and configured to access cluster
- Good: helm works as unprivileged user as well - Good for our setup
- install helm on ubuntu (client) as root: snap install --classic helm
  - this installs helm3
- Please only use: helm3. No server-side components needed (in cluster)
  - Get away from examples using helm2 (hint: helm init) - uses tiller

Simple Walkthrough (Example 0)

```
## Repo hinzufpgen
helm repo add bitnami https://charts.bitnami.com/bitnami
## gecachte Informationen aktualieren
helm repo update

helm search repo bitnami
## helm install release-name bitnami/mysql
helm install my-mysql bitnami/mysql
## Chart runterziehen ohne installieren
## helm pull bitnami/mysql

## Release anzeigen zu lassen
helm list

## Status einer Release / Achtung, heisst nicht unbedingt nicht, dass pod läuft
helm status my-mysql
```

```
## weitere release installieren
## helm install neuer-release-name  bitnami/mysql
```

Under the hood

```
## Helm speichert Informationen über die Releases in den Secrets
kubectl get secrets | grep helm
```

Example 1: - To get know the structure

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm search repo bitnami
helm repo update
helm pull bitnami/mysql
tar xzvf mysql-9.0.0.tgz
```

Example 2: We will setup mysql without persistent storage (not helpful in production ;o()

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm search repo bitnami
helm repo update

helm install my-mysql bitnami/mysql
```

Example 2 - continue - fehlerbehebung

```
helm uninstall my-mysql
## Install with persistentStorage disabled - Setting a specific value
helm install my-mysql --set primary.persistence.enabled=false bitnami/mysql

## just as notice
## helm uninstall my-mysql
```

Example 2b: using a values file

```
## mkdir helm-mysql
## cd helm-mysql
## vi values.yml
primary:
  persistence:
    enabled: false
```

```
helm uninstall my-mysql
helm install my-mysql bitnami/mysql -f values.yml
```

Example 3: Install wordpress

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm install my-wordpress \
  --set wordpressUsername=admin \
  --set wordpressPassword=password \
  --set mariadb.auth.rootPassword=secretpassword \
    bitnami/wordpress
```

Example 4: Install Wordpress with values and auth

```

## mkdir helm-mysql
## cd helm-mysql
## vi values.yml
persistence:
  enabled: false



wordpressUsername: admin
wordpressPassword: password
mariadb:
  primary:
    persistence:
      enabled: false
```

```
  auth:
    rootPassword: secretpassword

```

```
helm uninstall my-wordpress
helm install my-wordpress bitnami/wordpress -f values
```

Referenced

- https://github.com/bitnami/charts/tree/master/bitnami/mysql/#installing-the-chart
- https://helm.sh/docs/intro/quickstart/

Kubernetes - RBAC

Nutzer einrichten microk8s ab kubernetes 1.25

Schritt 1: Nutzer-Account auf Server anlegen und secret anlegen / in Client

```
cd
mkdir -p manifests/rbac
cd manifests/rbac
```

Mini-Schritt 1: Definition für Nutzer

```
nano 01-service-account.yml
```

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: training
  namespace: default
```

```
kubectl apply -f .
```

Mini-Schritt 1.5: Secret erstellen

- From Kubernetes 1.25 tokens are not created automatically when creating a service account (sa)
- You have to create them manually with annotation attached
- https://kubernetes.io/docs/reference/access-authn-authz/service-accounts-admin/#create-token

```
## vi 02-secret.yml
apiVersion: v1
kind: Secret
type: kubernetes.io/service-account-token
metadata:
  name: trainingtoken
  namespace: default
  annotations:
    kubernetes.io/service-account.name: training
```

```
kubectl apply -f .
```

Mini-Schritt 2: ClusterRolle festlegen - Dies gilt für alle namespaces, muss aber noch zugewiesen werden

```
nano 03-pods-clusterrole.yml
```

```
### Bevor sie zugewiesen ist, funktioniert sie nicht - da sie keinem Nutzer zugewiesen ist
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pods-clusterrole
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

```
kubectl apply -f .
```

Mini-Schritt 3: Die ClusterRolle den entsprechenden Nutzern über RoleBinding zu ordnen

```
## vi 04-rb-training-ns-default-pods.yml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: rolebinding-ns-default-pods
```

```
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: pods-clusterrole
subjects:
- kind: ServiceAccount
  name: training
  namespace: default
```

```
kubectl apply -f .
```

Mini-Schritt 4: Testen (klappt der Zugang)

```
kubectl auth can-i get pods -n default --as system:serviceaccount:default:training
## yes
kubectl auth can-i get deployment -n default --as system:serviceaccount:default:training
## no
kubectl auth can-i --list --as system:serviceaccount:default:training
```

Schritt 2: Context anlegen / Credentials auslesen und in kubeconfig hinterlegen (bis Version 1.25.)

Mini-Schritt 1: kubeconfig setzen

```
kubectl config set-context training-ctx --cluster kubernetes --user training

## extract name of the token from here

TOKEN=`kubectl get secret trainingtoken -o jsonpath='{.data.token}' | base64 --decode`
echo $TOKEN
kubectl config set-credentials training --token=$TOKEN
kubectl config use-context training-ctx

## Hier reichen die Rechte nicht aus
kubectl get deploy
## Error from server (Forbidden): pods is forbidden: User "system:serviceaccount:kube-system:training" cannot list # resource
"pods" in API group "" in the namespace "default"
```

Mini-Schritt 2:

```
kubectl config use-context training-ctx
kubectl get pods
```

Mini-Schritt 3: Zurück zum alten Default-Context

```
kubectl config get-contexts
```

```
CURRENT   NAME                          CLUSTER      AUTHINFO           NAMESPACE
          kubernetes-admin@kubernetes   kubernetes   kubernetes-admin
*         training-ctx                  kubernetes   training
```

```
kubectl config use-context kubernetes-admin@kubernetes
```

Refs:

- https://docs.oracle.com/en-us/iaas/Content/ContEng/Tasks/contengaddingserviceaccttoken.htm
- https://microk8s.io/docs/multi-user
- https://faun.pub/kubernetes-rbac-use-one-role-in-multiple-namespaces-d1d08bb08286

Ref: Create Service Account Token

- https://kubernetes.io/docs/reference/access-authn-authz/service-accounts-admin/#create-token

Besser: Nutzer einrichten mit Zertifikat

Step 0: create an new rolebinding for the group (we want to use)

```
kubectl create rolebinding developers --clusterrole=view --group=developers
```

Step 1: on your client: create private certificate

```
cd
mkdir -p certs
## create your private key
openssl genrsa -out ~/certs/jochen.key 4096
```

Step 2: on your client: create csr (certificate signing request)

```
nano ~/certs/jochen.csr.cnf
```

```
[ req ]
default_bits = 2048
prompt = no
default_md = sha256
distinguished_name = dn
[ dn ]
CN = jochen
O = developers
[ v3_ext ]
authorityKeyIdentifier=keyid,issuer:always
basicConstraints=CA:FALSE
keyUsage=keyEncipherment,dataEncipherment
extendedKeyUsage=serverAuth,clientAuth
```

```
## Create Certificate Signing Request
openssl req -config ~/certs/jochen.csr.cnf -new -key ~/certs/jochen.key -nodes -out ~/certs/jochen.csr
openssl req -in certs/jochen.csr --noout -text
```

Step 3: Send approval request to server

```
## get csr (base64 decoded)
cat ~/certs/jochen.csr | base64 | tr -d '\n'
```

```
cd certs
nano jochen-csr.yaml
```

```
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: jochen-authentication
spec:
  signerName: kubernetes.io/kube-apiserver-client
  groups:
    - system:authenticated
  request:
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSBSRVFVRVNULS0tLS0KTUlJRWF6Q0NBbE1DQVFBd0pqRVBNQTBHQTFVRUF3d0dhbTlqYUdWdU1STXdFUVlEVlFRS0RBcGtaWFpsYkc5

  usages:
  - client auth
```

```
kubectl apply -f jochen-csr.yaml
kubectl get -f jochen-csr.yaml
## show me the current state -> pending
kubectl describe -f jochen-csr.yaml
```

Step 4: approve signing request

```
kubectl certificate approve jochen-authentication
## or:
kubectl certificate approve -f jochen-csr.yaml

## see, that it is approved
kubectl describe -f jochen-csr.yaml
```

Step 5: get the approved certificate to be used

```
kubectl get csr jochen-authentication -o jsonpath='{.status.certificate}' | base64 --decode > ~/certs/jochen.crt
```

Step 6: construct kubeconfig for new user

```
cd
cd certs
```

```
## create new user
kubectl config set-credentials jochen --client-certificate=jochen.crt --client-key=jochen.key
```

```
## add a new context
kubectl config set-context jochen --user=jochen --cluster=kubernetes
```

Step 7: Use and test the new context

```
kubectl config use-context jochen
kubectl get pods
```

Ref:

- https://kb.leaseweb.com/kb/users-roles-and-permissions-on-kubernetes-rbac/kubernetes-users-roles-and-permissions-on-kubernetes-rbac-create-a-certificate-based-kubeconfig/

Kubernetes Advanced

Curl api-server kubernetes aus pod heraus

https://nieldw.medium.com/curling-the-kubernetes-api-server-d7675cfc398c

Kubernetes Load Balancer / metallb (on premise)

Kubernetes Load Balancer

General

- Supports bgp and arp
- Divided into controller, speaker

Installation Ways

- helm
- manifests

Step 1: install metallb

```
## Just to show some basics
## Page from metallb says that digitalocean is not really supported well
## So we will not install the speaker .

helm repo add metallb https://metallb.github.io/metallb
```

```
## Eventually disabling speaker
## vi values.yml
```

```
helm install metallb metallb/metallb --namespace=metallb-system --create-namespace --version 0.14.8
```

Step 2: addresspool und Propagation-type (config)

```
cd
mkdir -p manifests
cd manifests
mkdir lb
cd lb
nano 01-addresspool.yml
```

```
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: first-pool
  namespace: metallb-system
spec:
  addresses:
  # we will use our external ip here
  - 134.209.231.154-134.209.231.154
  # both notations are possible
  - 157.230.113.124/32
```

```
kubectl apply -f .
```

```
nano 02-advertisement.yml
```

```
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: example
  namespace: metallb-system
```

```
kubectl apply -f .
```

Schritt 4: Test do i get an external ip

```
nano 03-deploy.yml
```

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

```
nano 04-service.yml
```

```
apiVersion: v1
kind: Service
metadata:
  name: svc-nginx
  labels:
    svc: nginx
spec:
  type: LoadBalancer
  ports:
  - port: 80
    protocol: TCP
  selector:
    run: web-nginx
```

```
kubectl apply -f .
kubectl get pods
kubectl get svc
```

```
## auf dem client
curl http://<ip aus get svc>
```

```
kubectl delete -f 03-deploy.yml 04-service.yml
```

Schritt 5: Referenz:

- https://metallb.io/installation/#installation-with-helm

Kubernetes API Reference

API-Reference

- https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.29/

OpenAPI

- https://raw.githubusercontent.com/kubernetes/kubernetes/master/api/openapi-spec/swagger.json

Swagger Editor Online

- https://editor.swagger.io/

Kubernetes Istio

Why istio and benefits

What is a service mesh ?

```
A service mesh is an infrastructure layer
that gives applications capabilities
like zero-trust security, observability,
and advanced traffic management, without code changes.
```

Advantages / Features

1. Observability & monitoring
2. Traffic management
3. Resilience & Reliability

4. Security
5. Service Discovery

Observability & monitoring

- Service mesh offers:
  - valuable insights into the communication between services
  - effective monitoring to help in troubleshooting application errors.

Traffic management

- Service mesh offers:
  - intelligent request distribution
  - load balancing,
  - support for canary deployments.
  - These capabilities enhance resource utilization and enable efficient traffic management

Resilience & Reliability

- By handling retries, timeouts, and failures,
  - service mesh contributes to the overall stability and resilience of services
  - reducing the impact of potential disruptions.

Security

- Service mesh enforces security policies, and handles authentication, authorization, and encryption
  - ensuring secure communication between services and eventually, strengthening the overall security posture of the application.

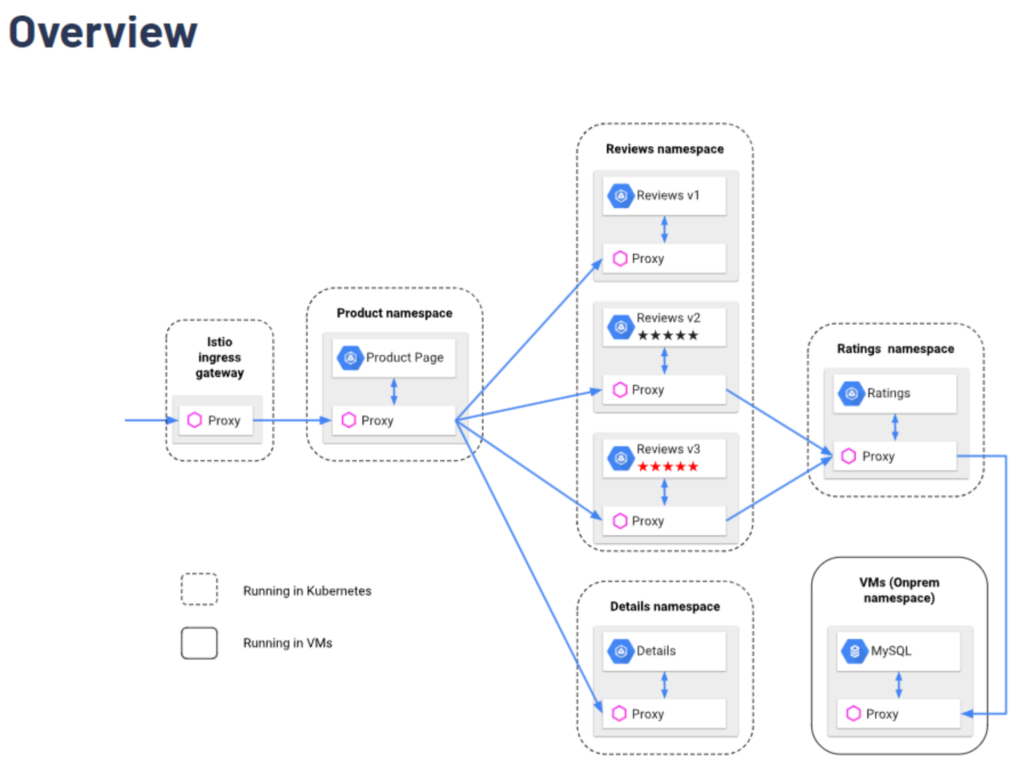
Service Discovery

- With service discovery features, service mesh can simplify the process of locating and routing services dynamically
- adapting to system changes seamlessly. This enables easier management and interaction between services.

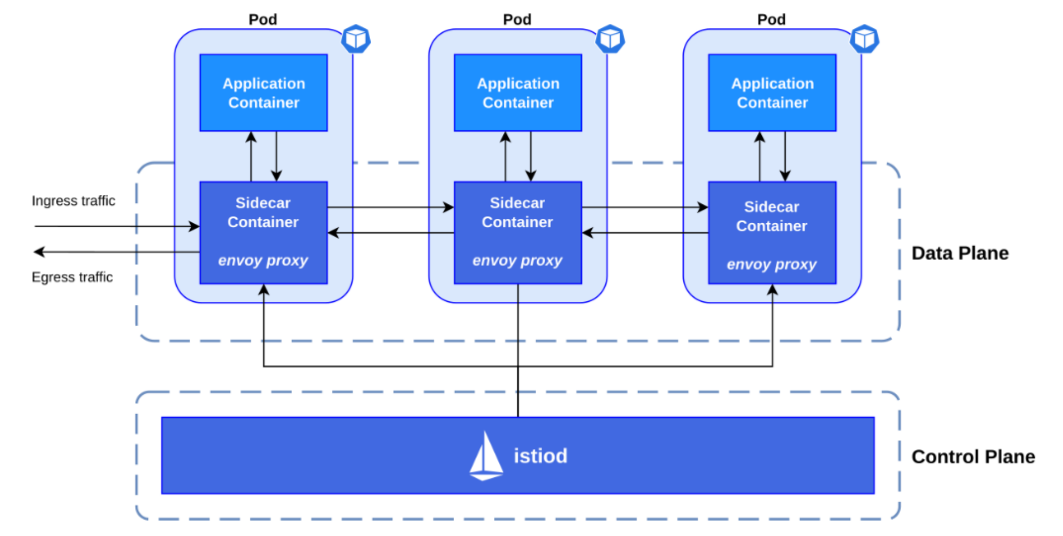
Overall benefits

```
Microservices communication:
Adopting a service mesh can simplify the implementation of a microservices architecture by abstracting away infrastructure
complexities.
It provides a standardized approach to manage and orchestrate communication within the microservices ecosystem.
```

Overview sidecar pattern



Istio control plane and data plane



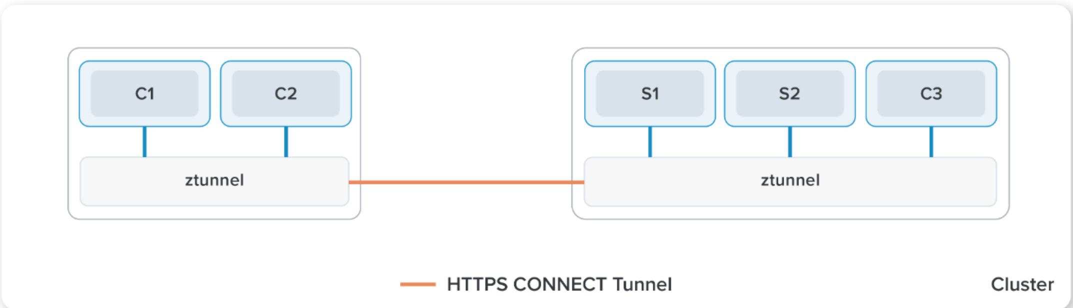
- Source: kubebyexample.com

Ambient Mode istio

Light: Only Layer 4 per node (ztunnel)

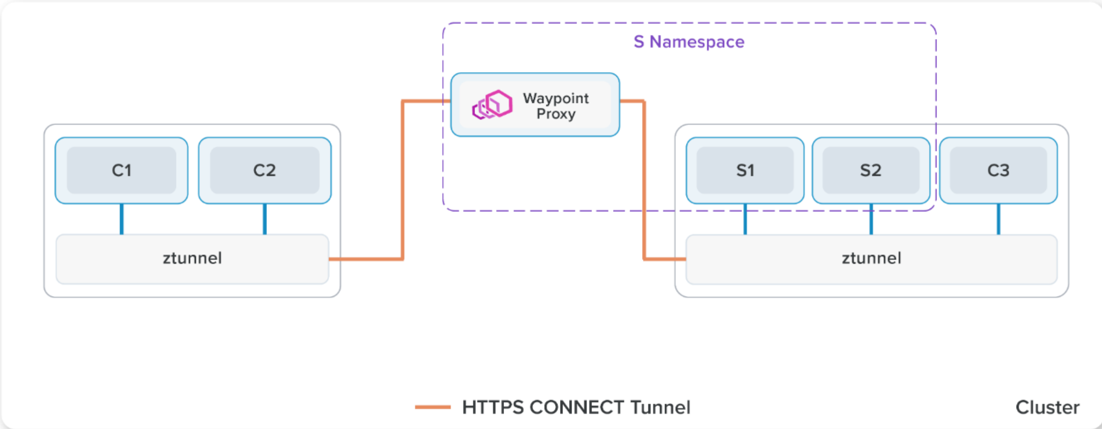
- No sidecar (envoy-proxy) per Pod, but one ztunnel agent per Node (Layer 4)
- Enables security features (mtls, traffic encryption)

Like so:



Full fledged: Layer 4 (ztunnel) per Node & Layer 7 per Namespace (

- One waypoint - proxy is rolled out per Namespace, which connects to the ztunnel agents



When additional features are needed, ambient mesh deploys waypoint proxies, which ztunnels connect through for policy enforcement

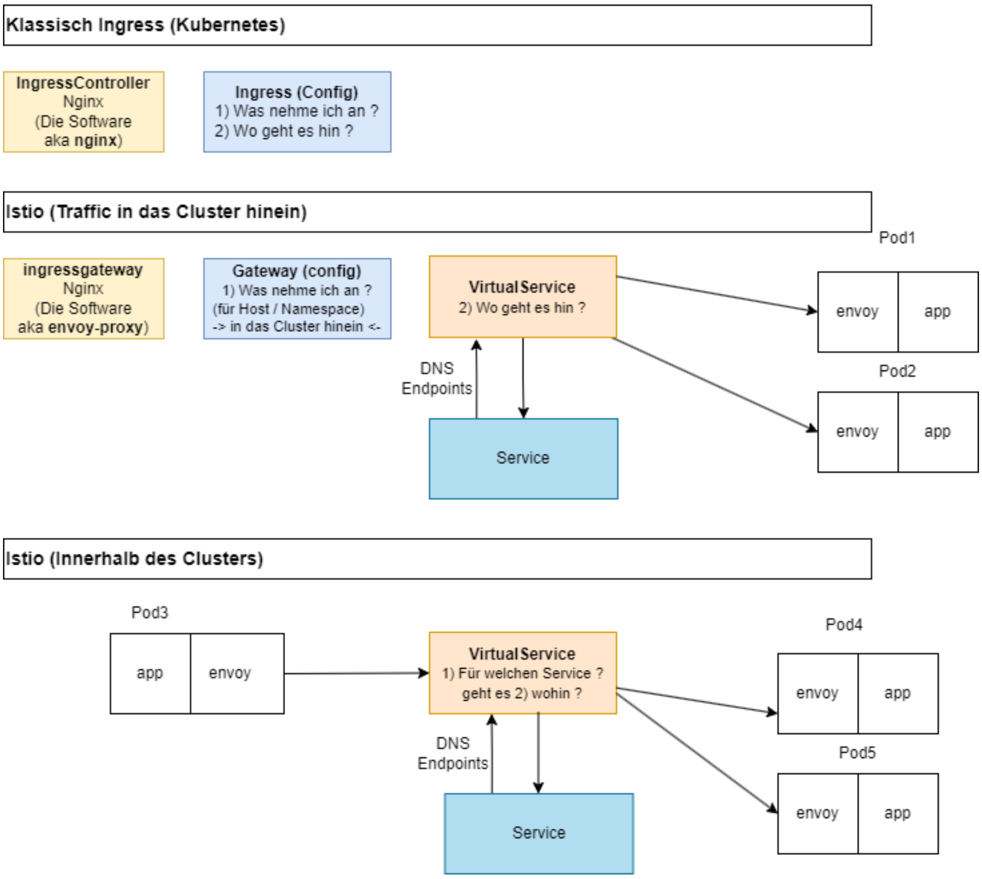
Features in "fully-fledged" - ambient - mode

Application deployment use case	Ambient mode configuration
Zero Trust networking via mutual-TLS, encrypted and tunneled data transport of client application traffic, L4 authorization, L4 telemetry	ztunnel only (default)
As above, plus advanced Istio traffic management features (including L7 authorization, telemetry and VirtualService routing)	ztunnel and waypoint proxies

Advantages:

- Less overhead
- One can start step-by-step moving towards a mesh (Layer 4 firstly and if wanted Layer 7 for specicfic namespaces)
- Old pattern: sidecar and new pattern: ambient can live side by side.

Istio vs. Ingress Überblick



Istio installieren und Addons bereitsstellen

On the client (where you also use kubectl)

Steps 1: Download install and run

```
## as tlnx - user
## find a decent where to run the installation
## not perfect, but better than to put it in home-folder
cd
mkdir -p manifests/istio
cd manifests/istio
```

```
## now download the install an run the shell
curl -L https://istio.io/downloadIstio | sh -
```

Step 2: Run istioctl - commands (version-check, precheck and install)

```
## This istioctl will be under istio-1.20.2/bin
## but TRAINER has already installed it under /usr/bin/istioctl
## So we can use that one !!
```

```
## cd istio-1.23.1/bin
istioctl version
istioctl x precheck
istioctl install --set profile=demo -y
```

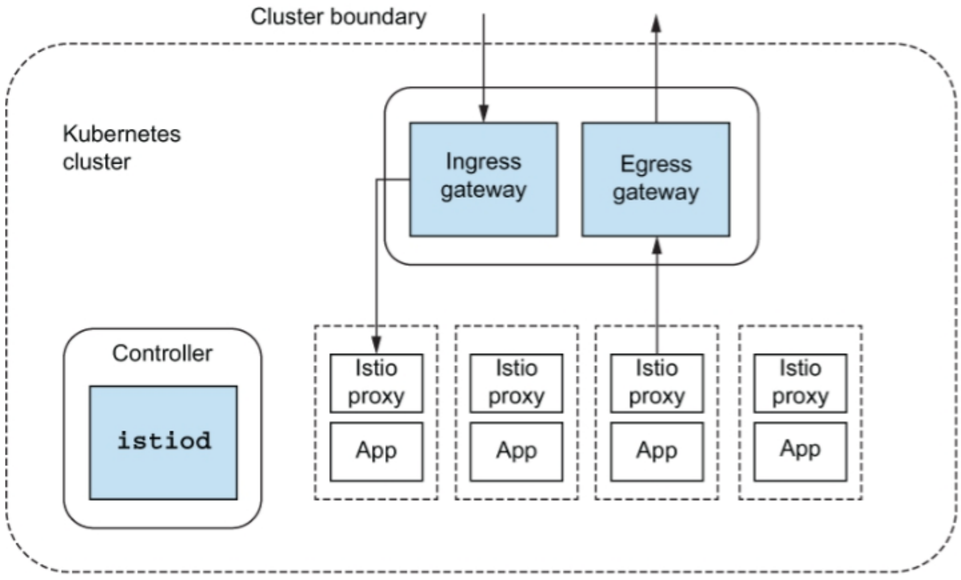
Step 3: Install the addons

```
## Install Add-Ons
## Ask trainer for directory
cd /usr/src/
kubectl apply -f istio-1.23.1/samples/addons/
```

Step 4: Check if all the corresponding container (from istio and addons) are running

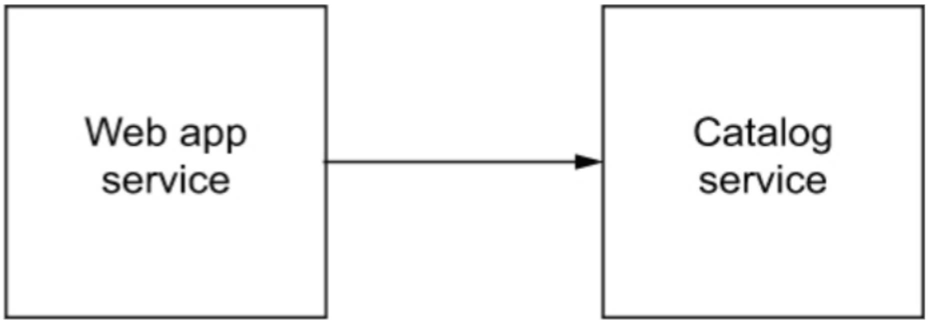
```
kubectl -n istio-system get pods
```

Istion Überblick - egress und ingress - gateway



Istio - Deployment of simple application

Overview (what we want to do)



- Catalog Service is reachable through api

Step 1: Vorbereitung - repo mit beispielen klonen

```
cd
git clone https://github.com/jmetzger/istio-exercises/
cd istio-exercises
```

Step 2: Eigenen Namespace erstellen

```
## Jeder Teilnehmer erstellt seinen eigenen Namespace
## z.B. istioapp-tlnx
## d.h. für Teilnehmer 5 (tln5) -> istioapp-tln5
kubectl create ns istioapp-tln5
## Context so einstellen, dass dieser namespace verwendet
kubectl config set-context --current --namespace istioapp-tln5
```

Step 3: Anwendung untersuchen / istioctl kube-inject

- Ihr könnt unten direkt den Pfad nehmen, das ist einfacher ;o)

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: catalog
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: catalog
  name: catalog
spec:
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: 3000
  selector:
    app: catalog
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: catalog
    version: v1
  name: catalog
spec:
  replicas: 1
  selector:
    matchLabels:
      app: catalog
      version: v1
  template:
    metadata:
      labels:
        app: catalog
        version: v1
    spec:
      serviceAccountName: catalog
      containers:
      - env:
        - name: KUBERNETES_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        image: istioinaction/catalog:latest
        imagePullPolicy: IfNotPresent
        name: catalog
        ports:
        - containerPort: 3000
          name: http
          protocol: TCP
        securityContext:
          privileged: false
```

```
## schauen wir uns das mal mit injection an
istioctl kube-inject -f services/catalog/kubernetes/catalog.yaml | less
```

Step 4: Automatische Injection einrichten.

```
## kubectl label namespace istioapp-tlnx istio-injection=enabled
## z.B
kubectl label namespace istioapp-tln5 istio-injection=enabled
```

Step 5: catalog ausrollen

```
kubectl apply -f services/catalog/kubernetes/catalog.yaml
```

```
## Prüfen, ob wirklich 2 container in einem pod laufen,
## dann funktioniert die Injection
## WORKS, Yeah !
kubectl get pods
```

Step 6: Wir wollen den Catalog jetzt erreichen

```
## do it from your namespace, e.g. tlnx
## z.B.
kubectl run -it --rm curly --image=curlimages/curl -- sh
```

```
## within shell of that pod
## catalog.yourappnamespace/items/1
curl http://catalog.istioapp-tln5/items/1
exit
```

Step 7: Jetzt deployen wir die webapp

```
## Wir schauen uns das manifest für die webapp an
## und ändern die env-variablen CATALOG_SERVICE_HOST
## tlnx durch Eure Teilnehmernummer ersetzen
catalog.istioapp-tlnx
```

```
kubectl apply -f services/webapp/kubernetes/webapp.yaml
kubectl get pod
```

Step 8: Verbindung zu webapp testen

```
## tlnx
## kubectl -n tlnx run -it --rm curly --image=curlimages/curl -- sh
## z.B.
kubectl run -it --rm curly --image=curlimages/curl -- sh
```

```
## Within shell connect to webapp
curl -s http://webapp.istioapp-tln5/api/catalog/items/1
exit
```

Step 8.1: Verbindung zu webapp von aussen

```
## Wir können es aber auch visualisieren
kubectl port-forward deploy/webapp 8001:8080
## z.B. Teilnehmer tln1 -> 8001:8080

## WICHTIG Jeder Teilneher sollte hier einen abweichenden Port nehmen
## Jetzt lokal noch einen Tunnel aufbauen
## s. Anleitung Putty
## Source Port: 8080 # das ist der auf dem Rechner
## Destination: localhost:8001
## Add
## Achtung -> danach noch Session speichern
```

```
## Jetzt im Browser http://localhost:8080
## aufrufen
```

Step 9: Ingress - Gateway konfigurieren (ähnlich wie Ingress-Objekt)

```
## wir schauen uns das vorher mal an
```

```
## namespace - fähig, d.h. ein Gateway mit gleichem Namen pro Namespace möglich
cat ingress-virtualservice/ingress-gateway.yaml
## hier bitte bei Hosts hostname eintragen, der für t3isp.de verwendet, und zwar
## jeder Teilnehmer eine eigene Subdomain:  z.B. jochen.istio.t3isp.de
kubectl apply -f ingress-virtualservice/ingress-gateway.yaml
```

Step 10: Reach it from outside

```
## We need to find the loadbalancer IP
kubectl -n istio-system get svc
## in unserem Fall
146.190.177.12
## Das trägt Jochen dns t3isp.de ein.

## Wir können jetzt also das System von extern erreichen
## vomn client aus, oder direkt über den Browser
##curl -i 146.190.177.12/api/catalog/items/1
## Hier hostname statt ip einträgen
curl -i http://tlnx.istio.t3isp.de/api/catalog/items/1
```

```
## Wir können auch über istioctl direkt überprüfen, ob es einen Routen-Config gibt
istioctl proxy-config routes deploy/istio-ingressgateway.istio-system
```

```
## Falls das nicht funktioniert, können wir auch überprüfen ob ein gateway und ein virtualservice installiert wurde
kubectl get gateway
kubectl get virtualservice
## Kurzform des Services reicht, weil im gleichen namespace
## Wo soll es hingehen -> == -> Upstream
## route -> destination -> host -> webapp
kubectl get virtualservice -o yaml
```

```
### Wichtiger Hinweis, auf beiden Seiten ingressgateway und vor dem Pod des Dienstes Webapp
### Sitzt ein envoy-proxy und kann Telemetrie-Daten und Insight sammeln was zwischen den
### applicationen passiert -> das passiert über ein sidecar in jeder Applikation

### Wichtig: Das passiert alles ausserhalb der Applikation
### Nicht wie früher z.B. bei Netflix innerhalb z.B. für die Sprache Java
```

Istio - Grafana Dashboard

Status

- Wir haben bereits mit den Addons Grafana ausgerollt,
- Dieses wollen wir jetzt aktivieren

Schritt 1: Dashboard aktivieren -> achtung jeder nimmt seinen eigenen Port

```
## um Grunde macht das auch nur ein port - forward
## Das macht der Trainer nur 1x, dann können alle dort zugreifen
istioctl dashboard grafana --port=3000 --browser=false
```

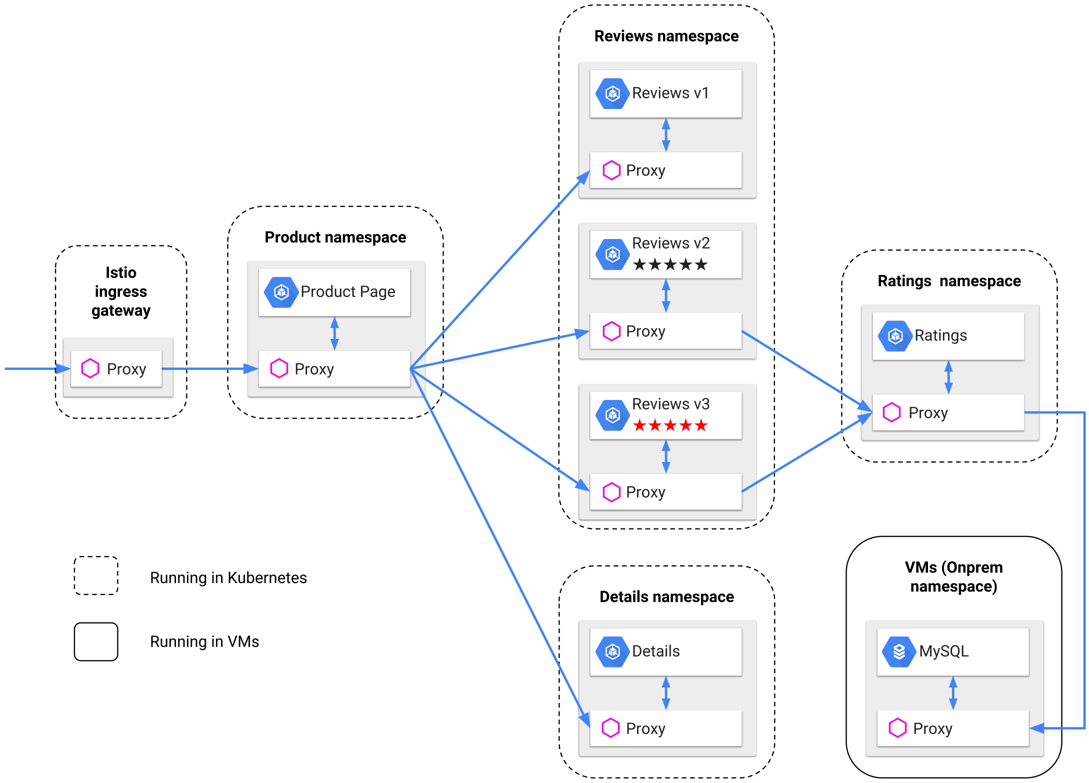
```
## Jetzt über den Browser öffnen
http://localhost:3000
## Dann Dashboard -> istio -> istio services
```

```
## Lass uns mal Traffic hinschicken vom Client aus
## ip vom ingressgateway from loadBalancer
while true; do curl http://jochen.istio.t3isp.de/api/catalog; sleep .5; done

## Und das das Dashboard nochmal refreshend
##-> General ausklappen
```

Mesh / istio

Schaubild



Istio

```
## Visualization
## with kiali (included in istio)
https://istio.io/latest/docs/tasks/observability/kiali/kiali-graph.png

## Example
## https://istio.io/latest/docs/examples/bookinfo/
The sidecars are injected in all pods within the namespace by labeling the namespace like so:
kubectl label namespace default istio-injection=enabled

## Gateway (like Ingress in vanilla Kubernetes)
kubectl label namespace default istio-injection=enabled
```

istio tls

- https://istio.io/latest/docs/ops/configuration/traffic-management/tls-configuration/

istio - the next generation without sidecar

- https://istio.io/latest/blog/2022/introducing-ambient-mesh/

Benchmarks istio ambient,sidecar und baseline

- https://livewyer.io/blog/2024/06/06/comparison-of-service-meshes-part-two/
- https://github.com/livewyer-ops/poc-servicemesh2024/blob/main/docs/test-report-istio.md

Kubernetes Deployment Scenarios

Deployment green/blue,canary,rolling update

Canary Deployment

```
A small group of the user base will see the new application
(e.g. 1000 out of 100.000), all the others will still see the old version

From: a canary was used to test if the air was good in the mine
(like a test balloon)
```

Blue / Green Deployment

```
The current version is the Blue one
The new version is the Green one

New Version (GREEN) will be tested and if it works
the traffic will be switch completey to the new version (GREEN)

Old version can either be deleted or will function as fallback
```

A/B Deployment/Testing

```
2 Different versions are online, e.g. to test a new design / new feature
You can configure the weight (how much traffic to one or the other)
by the number of pods
```

Example Calculation

```
e.g. Deployment1: 10 pods
Deployment2: 5 pods

Both have a common label,
The service will access them through this label
```

Service Blue/Green

Step 1: Deployment + Service

```
## vi blue.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-version-blue
spec:
  selector:
    matchLabels:
      version: blue
  replicas: 10 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
        version: blue
    spec:
      containers:
      - name: nginx
        image: nginx:1.21
        ports:
        - containerPort: 80
```

```
## vi green.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-version-green
spec:
  selector:
    matchLabels:
      version: green
  replicas: 1 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
        version: green
    spec:
      containers:
      - name: nginx
        image: nginx:1.22
        ports:
        - containerPort: 80
```

```
## svc.yml
apiVersion: v1
kind: Service
metadata:
  name: svc-nginx
spec:
```

```
  ports:
  - port: 80
    protocol: TCP
  selector:
    app: nginx
```

Step 2: Ingress

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-config
  annotations:
    ingress.kubernetes.io/rewrite-target: /
    # with the ingress controller from helm, you need to set an annotation
    # old version useClassName instead
    # otherwice it does not know, which controller to use
    # kubernetes.io/ingress.class: nginx
spec:
  ingressClassName: nginx
  rules:
  - host: "app.lab1.t3isp.de"
    http:
      paths:
        - path: /
          pathType: Prefix
          backend:
            service:
              name: svc-nginx
              port:
                number: 80
```

```
kubectl apply -f .
```

Praxis-Übung A/B Deployment

Walkthrough

```
cd
cd manifests
mkdir ab
cd ab
```

```
## vi 01-cm-version1.yml
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-version-1
data:
  index.html: |
    <html>
    <h1>Welcome to Version 1</h1>
    </br>
    <h1>Hi! This is a configmap Index file Version 1 </h1>
    </html>
```

```
## vi 02-deployment-v1.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deploy-v1
spec:
  selector:
    matchLabels:
      version: v1
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
        version: v1
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
```

```
        - containerPort: 80
        volumeMounts:
            - name: nginx-index-file
              mountPath: /usr/share/nginx/html/
      volumes:
      - name: nginx-index-file
        configMap:
          name: nginx-version-1
```

```
## vi 03-cm-version2.yml
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-version-2
data:
  index.html: |
    <html>
    <h1>Welcome to Version 2</h1>
    </br>
    <h1>Hi! This is a configmap Index file Version 2 </h1>
    </html>
```

```
## vi 04-deployment-v2.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deploy-v2
spec:
  selector:
    matchLabels:
      version: v2
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
        version: v2
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
        volumeMounts:
            - name: nginx-index-file
              mountPath: /usr/share/nginx/html/
      volumes:
      - name: nginx-index-file
        configMap:
          name: nginx-version-2
```

```
## vi 05-svc.yml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    svc: nginx
spec:
  type: NodePort
  ports:
  - port: 80
    protocol: TCP
  selector:
    app: nginx
```

```
kubectl apply -f .
## get external ip
kubectl get nodes -o wide
## get port
kubectl get svc my-nginx -o wide
## test it with curl apply it multiple time (at least ten times)
curl <external-ip>:<node-port>
```

Kubernetes - Hilfreiche Tools / Networking /Debugging

NetworkPolicyEditor

- https://editor.networkpolicy.io

Debug on node with root-priviliges with kubectl debug

```
kubectl debug node/worker2 -it --image=nicolaka/netshoot --profile=netadmin
## in der shell
apk add iptables-legacy
```

Kubernetes Security

NodeShell vs. kubectl debug (for node)

Explanation

`node-shell` (e.g., from `kops` or other tools)

- **What it is**: A command that typically starts a privileged pod on the node you want to debug.
- **How it works**:
  - It schedules a **privileged pod** on the node using `hostPID`, `hostNetwork`, and often mounts the node's filesystem.
  - Once the pod is up, it execs into a shell inside that pod.
- **Main Use Case**: Used primarily to get "inside" a node when you need full access to the node's environment from within Kubernetes (e.g., inspecting logs, filesystems, or processes).
- **Tools**: Often provided by tools like `kops`, e.g., `kops toolbox node-shell <node-name>`.

✅ `kubectl debug` (with node targeting)

- **What it is**: A native `kubectl` command introduced in Kubernetes 1.18+ (stable around 1.23) for debugging pods and nodes.
- **How it works (node debug)**:
  - You run something like:

    ```
    kubectl debug node/<node-name> -it --image=busybox
    ```

  - It creates a **debug pod** on that node, using similar capabilities to `node-shell` (privileged, `hostPID`, etc.).
  - Offers more control (custom images, attach to namespaces, etc.) and is more **standardized** across clusters.

Key Differences

Feature	`node-shell`	`kubectl debug node`
Native to Kubernetes	❌ (usually via `kops`, scripts)	✅ Yes
Requires external tooling	✅ Often (`kops`, scripts)	❌ Just `kubectl`
Uses `kubectl`	✅ Yes (indirectly)	✅ Yes
Customizable image	❌ Usually fixed	✅ You can specify the image
Part of standard workflow	❌ More "legacy" or tool-specific	✅ Recommended upstream method
Works on any cluster	❌ Usually limited (e.g., to `kops`)	✅ Works on any compliant cluster

---

✅ Recommendation

- Prefer **`kubectl debug`** for node access if your Kubernetes version supports it and you're not tied to a specific tooling ecosystem.
- Use **`node-shell`** if you're in a `kops` environment or using automation/scripts that depend on it.

Want a sample command for either?

Security Concerns with `node-shell`

1. Privileged Access

- It runs a **privileged pod** with:
  - `hostPID: true` → can see/manipulate processes on the node.
  - `hostNetwork: true` → full network stack access.
  - Mounting the node's **entire root filesystem**.
- This essentially gives **root-level access** to the node, which can be risky if misused.

2. Cluster-Wide Risk

- If someone gains access to this mechanism and can launch `node-shell`, they could potentially:
  - Inspect secrets.
  - Tamper with system files.
  - Escalate privileges or break out of isolation.

3. Audit and Logging

- It's not always obvious who launched a `node-shell` session.
- If your environment **doesn't log pod creations properly**, this might fly under the radar.
- `kubectl debug`, by contrast, has more clear semantics and integrates better with Kubernetes auditing.

4. Non-Standard Behavior

- `node-shell` is often **tool-specific** (e.g., `kops`) and not portable or standardized across clusters.
- This can lead to confusion or inconsistent behavior in different environments (especially if you're collaborating across teams or orgs).

✅ **When Is `node-shell` Still Useful?**

- **Dev or staging clusters**, where quick debugging is more important than strict security controls.
- When using `kops`, and you want a simple, reliable way to jump into nodes.
- If your team already **audits or restricts RBAC access** to who can run it.

Recommendations

Practice	Suggestion
Use RBAC	Only let **trusted admins** run `node-shell` or create privileged pods.
Audit logs	Enable **audit logging** to track who creates pods like this.
Prefer `kubectl debug`	Especially in **regulated** or **production** environments.
Avoid running it long-term	Don't leave debug pods running — they could be misused.

Want a quick `kubectl debug` alternative to `node-shell` for a node? I can give you a drop-in command.

Kube Api Server - Watch Internals

Der `watch`-Mechanismus im Kubernetes API Server ist ein zentrales Feature für effiziente Reaktionen auf Änderungen im Clusterzustand – z. B. wenn Controller, Operatoren oder Tools wie `kubectl` Ressourcenänderungen beobachten. Hier ist eine technische Erklärung, wie das funktioniert:

Grundidee von `watch`

Statt regelmäßig (`polling`) den API-Server nach dem Zustand einer Ressource zu fragen, öffnet der Client eine **langlaufende HTTP-Verbindung**, bei der der API-Server **kontinuierlich Änderungen streamt**, sobald sie auftreten.

Technische Funktionsweise

```
GET /api/v1/pods?watch=true
```

...dann antwortet der API Server **nicht mit einer einmaligen Liste**, sondern beginnt, **ein Event-Stream (HTTP-Streaming)** zurückzugeben.

3. Events als Antwort

Der Client erhält ein Stream von JSON-Objekten wie:

```
{
  "type": "ADDED",
  "object": {
    "metadata": { ... }
  }
}
{
  "type": "MODIFIED",
  "object": { ... }
}
{
  "type": "DELETED",
  "object": { ... }
}
```

Jedes dieser Events beschreibt eine Änderung, die in etcd passiert ist.

⚡ Performance: Wie skaliert das?

- Der API Server cached Änderungen (Change history, genannt **watch cache**) für kurze Zeit im RAM.
- So muss er nicht ständig etcd abfragen.
- Das erhöht die Performance bei vielen parallelen Watches (z. B. in großen Clustern mit vielen Controllern).

Beispiel mit curl

```
curl -k -H "Authorization: Bearer $TOKEN" \
  "https://<apiserver>/api/v1/namespaces/default/pods?watch=true"
```

Du bekommst eine `chunked` HTTP-Antwort mit Live-Updates.

Super spannende Frage – hier tauchen wir richtig tief in die Kubernetes-Interna ein!

Kann man den Watch-Stream im API Server "sehen"?

Kurz gesagt:

Du kannst den Traffic **sehen**, aber **nicht direkt einsehen oder entschlüsseln**, wenn du nicht Teil des Traffics bist oder nicht Zugriff auf den TLS-Schlüssel hast.

Welches Protokoll wird verwendet?

✅ **HTTP/1.1 über TLS**

- Kubernetes API Calls, inklusive `watch`, laufen über **HTTPS** (also HTTP/1.1 über TLS).
- Der `watch` -Mechanismus verwendet **HTTP-Streaming** mit `Transfer-Encoding: chunked`.

Ist der Stream verschlüsselt?

Ja – immer!

- Selbst innerhalb des Clusters läuft die Kommunikation mit dem API-Server meist über HTTPS.
- Die Verbindung ist durch **TLS verschlüsselt**, also ohne den privaten Schlüssel **nicht entschlüsselbar**.

Was könntest du trotzdem beobachten?

Möglichkeit 1: Audit Logs aktivieren

- Du kannst in der API-Server-Konfiguration **Audit Logging** aktivieren.
- Dann siehst du, **welcher Client** wann einen `watch` auf welche Ressourcen geöffnet hat – aber nicht den Stream selbst.

Beispiel aus einem Audit Log:

```
{
  "verb": "watch",
  "user": { "username": "system:kube-controller-manager" },
  "objectRef": { "resource": "pods", "namespace": "default" },
  "stage": "ResponseStarted"
}
```

➡ Zeigt dir: Watch wurde geöffnet, aber nicht was "gestreamt" wurde.

Möglichkeit 2: Mitprotokollieren mit einem Proxy (z. B. mit MITM)

- Wenn du in einer Testumgebung bist, kannst du den Traffic über einen **TLS-terminierenden Proxy** wie [mitmproxy] oder Envoy umleiten.
- Dort kannst du dann den **klartext HTTP-Stream** beobachten.
- Das geht natürlich **nur**, wenn du den TLS-Traffic aufbrechen kannst, z. B. durch eigene Zertifikate oder Debug-Zugriff.

Möglichkeit 3: Einen eigenen Watch-Client schreiben

- Du kannst ein Tool oder Script schreiben (z. B. in Python oder Go), das einen `watch=true` -Request stellt.
- Dann **siehst du direkt**, was der API Server als Stream sendet.

Beispiel in Go (vereinfacht):

```
clientset.CoreV1().Pods("").Watch(context.TODO(), metav1.ListOptions{
    Watch: true,
})
```

Fazit

Aspekt	Antwort
Protokoll	HTTP/1.1 über TLS
Stream sichtbar?	Nein, verschlüsselt
Entschlüsselung möglich?	Nur mit MITM oder privatem TLS-Zertifikat
Auditierbar?	Ja, mit Audit Logs
Debugging möglich?	Ja, mit eigenen Clients oder Proxies

Wenn du willst, kann ich dir ein Beispiel-Setup machen, wie du z. B. mit `mitmproxy` oder Envoy den Watch-Traffic in einer Testumgebung sichtbar machst. Oder ein kleines Python-Tool, das einen Watch aufruft und ausgibt. Sag einfach Bescheid!

Sehr gute und tiefgehende Frage – du willst's wirklich wissen!
Schauen wir uns an, **welcher private Schlüssel bei der Verschlüsselung des Kubernetes API-Traffics verwendet wird**:

Welcher private Key wird verwendet?

Der des API Servers selbst, also:

> ***Der private Schlüssel des TLS-Zertifikats**, mit dem der Kubernetes API Server seine HTTPS-Verbindung absichert.*

Wo liegt dieser Key?

In einem typischen Kubernetes-Setup (z. B. kubeadm oder Vanilla):

- Zertifikat: `/etc/kubernetes/pki/apiserver.crt`
- Privater Schlüssel: `/etc/kubernetes/pki/apiserver.key`

Das sieht z. B. so aus:

```
$ ls -l /etc/kubernetes/pki/apiserver.*
-rw------- 1 root root 1675 apiserver.key
-rw-r--r-- 1 root root 1363 apiserver.crt
```

Nur **der API Server** (bzw. Root-User auf dem Host) hat Zugriff auf diesen Key. Er wird verwendet, um TLS-Handshake-Anfragen von Clients zu beantworten und die Verbindung zu verschlüsseln.

Kann ich den Traffic entschlüsseln, wenn ich den Key habe?

Ja – mit Tools wie `Wireshark`, `mitmproxy`, `ssldump`, oder `openssl` kannst du den TLS-Traffic **entschlüsseln**, **wenn** du Zugriff auf den Private Key des API Servers hast **und** Perfect Forward Secrecy (PFS) **deaktiviert** ist (was allerdings meist **nicht der Fall** ist – siehe unten).

⚠ ABER: Was ist mit Perfect Forward Secrecy (PFS)?

Die meisten Kubernetes API Server verwenden moderne TLS-Cipher wie:

```
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
```

Diese verwenden **Ephemeral Diffie-Hellman (ECDHE)** → bedeutet:

> *Selbst wenn du den privaten Schlüssel **nachträglich** bekommst, kannst du vergangene Verbindungen **nicht entschlüsseln**, weil die Sitzungsschlüssel **flüchtig erzeugt und nicht speicherbar** sind.*

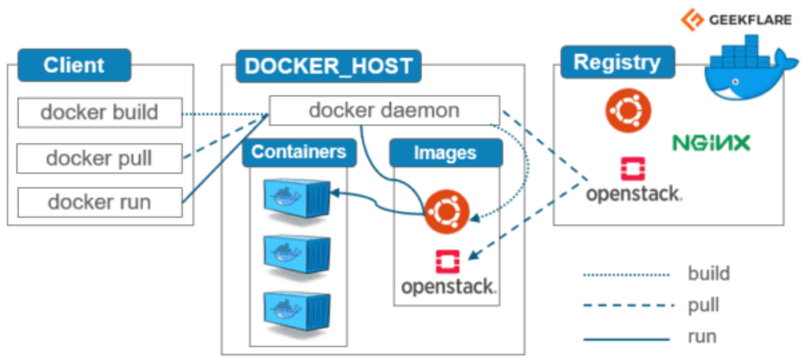
PFS schützt also effektiv gegen spätere Entschlüsselung, auch mit Key.

Fazit

Frage	Antwort
Wer hat den Key?	Der API Server (Datei: `apiserver.key`)
Womit wird verschlüsselt?	Mit dem TLS-Zertifikat des API Servers
Kann ich damit Traffic entschlüsseln?	Nur in Spezialfällen, wenn kein PFS aktiv ist
Ist PFS aktiv?	In >99 % aller modernen Kubernetes-Setups: **Ja**

Docker Überblick

Docker Aufbau



Was sind Container

```
- vereint in sich Software
- Bibliotheken
- Tools
- Konfigurationsdateien
- keinen eigenen Kernel
- gut zum Ausführen von Anwendungen auf verschiedenen Umgebungen

- Container sind entkoppelt
- Container sind voneinander unabhängig
- Können über wohldefinierte Kommunikationskanäle untereinander Informationen austauschen

- Durch Entkopplung von Containern:
  o Unverträglichkeiten von Bibliotheken, Tools oder Datenbank können umgangen werden, wenn diese von den Applikationen in unterschiedlichen Versionen benötigt werden.
```

Was sind container images

- Container Image benötigt, um zur Laufzeit Container-Instanzen zu erzeugen
- Bei Docker werden Docker Images zu Docker Containern, wenn Sie auf einer Docker Engine als Prozess ausgeführt werden.
- Man kann sich ein Docker Image als Kopiervorlage vorstellen.
  - Diese wird genutzt, um damit einen Docker Container als Kopie zu erstellen

Was ist ein Dockerfile

What is it ?

- Textdatei, die Linux - Kommandos enthält
  - die man auch auf der Kommandozeile ausführen könnte
  - Diese erledigen alle Aufgaben, die nötig sind, um ein Image zusammenzustellen
  - mit docker build wird dieses image erstellt

Exmaple

```
## syntax=docker/dockerfile:1
FROM ubuntu:18.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

Kubernetes - Überblick

Installation - Welche Komponenten from scratch

Step 1: Server 1 (manuell installiert -> microk8s)

```
## Installation Ubuntu - Server

## cloud-init script
## s.u. BASIS (keine Voraussetzung - nur zum Einrichten des Nutzers 11trainingdo per ssh)

## Server 1 - manuell
## Ubuntu 20.04 LTS - Grundinstallation

## minimal Netzwerk - öffentlichen IP
## nichts besonderes eingerichtet - Standard Digitalocean

## Standard vo Installation microk8s
lo               UNKNOWN        127.0.0.1/8 ::1/128
## public ip / interne
eth0             UP             164.92.255.234/20 10.19.0.6/16 fe80::c:66ff:fec4:cbce/64
## private ip
eth1             UP             10.135.0.3/16 fe80::8081:aaff:feaa:780/64

snap install microk8s --classic
## namensaufloesung fuer pods
microk8s enable dns
```

```
## Funktioniert microk8s
microk8s status
```

Steps 2: Server 2+3 (automatische Installation -> microk8s)

```
## Was macht das ?
## 1. Basisnutzer (11trainingdo) - keine Voraussetzung für microk8s
## 2. Installation von microk8s
##.>>>>>>> microk8s installiert <<<<<<<<
## - snap install --classic microk8s
## >>>>>>> Zuordnung zur Gruppe microk8s - notwendig für bestimmte plugins (z.B. helm)
## usermod -a -G microk8s root
## >>>>>>> Setzen des .kube - Verzeichnisses auf den Nutzer microk8s -> nicht zwingend erforderlich
## chown -r -R microk8s ~/.kube
## >>>>>>> REQUIRED .. DNS aktivieren, wichtig für Namensauflösungen innerhalb der PODS
## >>>>>>> sonst funktioniert das nicht !!!
## microk8s enable dns
## >>>>>>> kubectl alias gesetzt, damit man nicht immer microk8s kubectl eingeben muss
## - echo "alias kubectl='microk8s kubectl'" >> /root/.bashrc

## cloud-init script
## s.u. MITMICROK8S (keine Voraussetzung - nur zum Einrichten des Nutzers 11trainingdo per ssh)
##cloud-config
users:
```

```
  - name: 11trainingdo
    shell: /bin/bash

runcmd:
  - sed -i "s/PasswordAuthentication no/PasswordAuthentication yes/g" /etc/ssh/sshd_config
  - echo " " >> /etc/ssh/sshd_config
  - echo "AllowUsers 11trainingdo" >> /etc/ssh/sshd_config
  - echo "AllowUsers root" >> /etc/ssh/sshd_config
  - systemctl reload sshd
  - sed -i '/11trainingdo/c
11trainingdo:$6$HeLUJW3a$4xSfDFQjKWfAoGkZF3LFAxM4hgl3d6ATbr2kEu9zMOFwLxkYMO.AJF526mZONwdmsm9sg0tCBKl.SYbhS52u70:17476:0:99999:7:::'
 /etc/shadow
  - echo "11trainingdo ALL=(ALL) ALL" > /etc/sudoers.d/11trainingdo
  - chmod 0440 /etc/sudoers.d/11trainingdo

  - echo "Installing microk8s"
  - snap install --classic microk8s
  - usermod -a -G microk8s root
  - chown -f -R microk8s ~/.kube
  - microk8s enable dns
  - echo "alias kubectl='microk8s kubectl'" >> /root/.bashrc
```

```
## Prüfen ob microk8s - wird automatisch nach Installation gestartet
## kann eine Weile dauern
microk8s status
```

Step 3: Client - Maschine (wir sollten nicht auf control-plane oder cluster - node arbeiten

```
Weiteren Server hochgezogen.
Vanilla + BASIS

## Installation Ubuntu - Server

## cloud-init script
## s.u. BASIS (keine Voraussetzung - nur zum Einrichten des Nutzers 11trainingdo per ssh)

## Server 1 - manuell
## Ubuntu 20.04 LTS - Grundinstallation

## minimal Netzwerk - öffentlichen IP
## nichts besonderes eingerichtet - Standard Digitalocean

## Standard vo Installation microk8s
lo               UNKNOWN        127.0.0.1/8 ::1/128
## public ip / interne
eth0             UP             164.92.255.232/20 10.19.0.6/16 fe80::c:66ff:fec4:cbce/64
## private ip
eth1             UP             10.135.0.5/16 fe80::8081:aaff:feaa:780/64
```

```
##### Installation von kubectl aus dem snap
## NICHT .. keine microk8s - keine control-plane / worker-node
## NUR Client zum Arbeiten
snap install kubectl --classic

##### .kube/config
## Damit ein Zugriff auf die kube-server-api möglich
## d.h. REST-API Interface, um das Cluster verwalten.
## Hier haben uns für den ersten Control-Node entschieden
## Alternativ wäre round-robin per dns möglich

## Mini-Schritt 1:
## Auf dem Server 1: kubeconfig ausspielen
microk8s config > /root/kube-config
## auf das Zielsystem gebracht (client 1)
scp /root/kubeconfig 11trainingdo@10.135.0.5:/home/11trainingdo

## Mini-Schritt 2:
## Auf dem Client 1 (diese Maschine) kubeconfig an die richtige Stelle bringen
## Standardmäßig der Client nach eine Konfigurationsdatei sucht in ~/.kube/config
sudo su -
cd
mkdir .kube
cd .kube
mv /home/11trainingdo/kube-config config

## Verbindungstest gemacht
```

```
## Damit feststellen ob das funktioniert.
kubectl cluster-info
```

Schritt 4: Auf allen Servern IP's hinterlegen und richtigen Hostnamen überprüfen

```
## Auf jedem Server
hostnamectl
## evtl. hostname setzen
## z.B. - auf jedem Server eindeutig
hostnamectl set-hostname n1.training.local

## Gleiche hosts auf allen server einrichten.
## Wichtig, um Traffic zu minimieren verwenden, die interne (private) IP

/etc/hosts
10.135.0.3 n1.training.local n1
10.135.0.4 n2.training.local n2
10.135.0.5 n3.training.local n3
```

Schritt 5: Cluster aufbauen

```
## Mini-Schritt 1:
## Server 1: connection - string (token)
microk8s add-node
## Zeigt Liste und wir nehmen den Eintrag mit der lokalen / öffentlichen ip
## Dieser Token kann nur 1x verwendet werden und wir auf dem ANDEREN node ausgeführt
## microk8s join 10.135.0.3:25000/e9cdaa11b5d6d24461c8643cdf107837/bcad1949221a

## Mini-Schritt 2:
## Dauert eine Weile, bis das durch ist.
## Server 2: Den Node hinzufügen durch den JOIN - Befehl
microk8s join 10.135.0.3:25000/e9cdaa11b5d6d24461c8643cdf107837/bcad1949221a

## Mini-Schritt 3:
## Server 1: token besorgen für node 3
microk8s add-node

## Mini-Schritt 4:
## Server 3: Den Node hinzufügen durch den JOIN-Befehl
microk8s join 10.135.0.3:25000/09c96e57ec12af45b2752fb45450530c/bcad1949221a

## Mini-Schritt 5: Überprüfen ob HA-Cluster läuft
Server 1: (es kann auf jedem der 3 Server überprüft werden, auf einem reicht
microk8s status | grep high-availability
high-availability: yes
```

Ergänzend nicht notwendige Scripte

```
## cloud-init script
## s.u. BASIS (keine Voraussetzung - nur zum Einrichten des Nutzers 11trainingdo per ssh)

## Digitalocean - unter user_data reingepastet beim Einrichten

##cloud-config
users:
  - name: 11trainingdo
    shell: /bin/bash

runcmd:
  - sed -i "s/PasswordAuthentication no/PasswordAuthentication yes/g" /etc/ssh/sshd_config
  - echo " " >> /etc/ssh/sshd_config
  - echo "AllowUsers 11trainingdo" >> /etc/ssh/sshd_config
  - echo "AllowUsers root" >> /etc/ssh/sshd_config
  - systemctl reload sshd
  - sed -i '/11trainingdo/c
11trainingdo:$6$HeLUJW3a$4xSfDFQjKWfAoGkZF3LFAxM4hgl3d6ATbr2kEu9zMOFwLxkYMO.AJF526mZONwdmsm9sg0tCBKl.SYbhS52u70:17476:0:99999:7:::'
 /etc/shadow
  - echo "11trainingdo ALL=(ALL) ALL" > /etc/sudoers.d/11trainingdo
  - chmod 0440 /etc/sudoers.d/11trainingdo
```

Kubernetes - microk8s (Installation und Management)

kubectl unter windows - Remote-Verbindung zu Kuberenets (microk8s) einrichten

Walkthrough (Installation)

```
## Step 1
chocolatry installiert.
(powershell als Administrator ausführen)
## https://docs.chocolatey.org/en-us/choco/setup
Set-ExecutionPolicy Bypass -Scope Process -Force; [System.Net.ServicePointManager]::SecurityProtocol =
[System.Net.ServicePointManager]::SecurityProtocol -bor 3072; iex ((New-Object
System.Net.WebClient).DownloadString('https://community.chocolatey.org/install.ps1'))

## Step 2
choco install kubernetes-cli

## Step 3
testen:
kubectl version --client

## Step 4:
## powershell als normaler benutzer öffnen
```

Walkthrough (autocompletion)

```
in powershell (normaler Benutzer)
kubectl completion powershell | Out-String | Invoke-Expression
```

kubectl - config - Struktur vorbereiten

```
## in powershell im heimatordner des Benutzers .kube - ordnern anlegen
## C:\Users\<dein-name>\
mkdir .kube
cd .kube
```

IP von Cluster-Node bekommen

```
## auf virtualbox - maschine per ssh einloggen
## öffentliche ip herausfinden - z.B. enp0s8 bei HostOnly - Adapter
ip -br a
```

config für kubectl aus Cluster-Node auslesen (microk8s)

```
## auf virtualbox - maschine per ssh einloggen / zum root wechseln
## abfragen
microk8s config

## Alle Zeilen ins clipboard kopieren
## und mit notepad++ in die Datei \Users\<dein-name>\.kube\config
## schreiben

## Wichtig: Zeile cluster -> clusters / server
## Hier ip von letztem Schritt eintragen:
## z.B.
Server: https://192.168.56.106/......
```

Testen

```
## in powershell
## kann ich eine Verbindung zum Cluster aufbauen ?
kubectl cluster-info
```

- https://kubernetes.io/docs/tasks/tools/install-kubectl-windows/

Arbeiten mit der Registry

Installation Kubernetes Dashboard

Reference:

- https://blog.tippybits.com/installing-kubernetes-in-virtualbox-3d49f666b4d6

Remote-Verbindung zu Kubernetes (microk8s) einrichten

```
## on CLIENT install kubectl
sudo snap install kubectl --classic

## On MASTER -server get config
## als root
cd
microk8s config > /home/kurs/remote_config
```

```
## Download (scp config file) and store in .kube - folder
cd ~
mkdir .kube
cd .kube  # Wichtig: config muss nachher im verzeichnis .kube liegen
## scp kurs@master_server:/path/to/remote_config config
## z.B.
scp kurs@192.168.56.102:/home/kurs/remote_config config
## oder benutzer 11trainingdo
scp 11trainingdo@192.168.56.102:/home/11trainingdo/remote_config config

##### Evtl. IP-Adresse in config zum Server aendern

## Ultimative 1. Test auf CLIENT
kubectl cluster-info

## or if using kubectl or alias
kubectl get pods

## if you want to use a different kube config file, you can do like so
kubectl --kubeconfig /home/myuser/.kube/myconfig
```

Kubernetes - Ingress

Ingress controller in microk8s aktivieren

Aktivieren

```
microk8s enable ingress
```

Referenz

- https://microk8s.io/docs/addon-ingress

ingress mit ssl absichern

Kubernetes API-Objekte

Pod manifest

Walkthrough

```
cd
mkdir -p manifests
cd manifests
mkdir -p web
cd web
```

```
## vi nginx-static.yml

apiVersion: v1
kind: Pod
metadata:
  name: nginx-static-web
  labels:
    webserver: nginx
spec:
  containers:
  - name: web
    image: nginx
```

```
kubectl apply -f nginx-static.yml
kubectl describe pod nginx-static-web
## show config
kubectl get pod/nginx-static-web -o yaml
kubectl get pod/nginx-static-web -o wide
```

Replicasets

```
cd
mkdir -p manifests
cd manifests
mkdir 02-rs
cd 02-rs
## vi rs.yml
```

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-replica-set
spec:
  replicas: 2
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      name: template-nginx-replica-set
      labels:
        tier: frontend
    spec:
      containers:
        - name: nginx
          image: nginx:1.21
          ports:
            - containerPort: 80
```

```
kubectl apply -f rs.yml
```

Hintergrund Ingress

Ref. / Dokumentation

- https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-ingress-guide-nginx-example.html

Documentation for default ingress nginx

- https://kubernetes.github.io/ingress-nginx/user-guide/nginx-configuration/configmap/

Beispiel Ingress

Prerequisits

```
## Ingress Controller muss aktiviert sein
microk8s enable ingress
```

Walkthrough

Schritt 1:

```
cd
mkdir -p manifests
cd manifests
mkdir abi
cd abi
```

```
## apple.yml
## vi apple.yml
kind: Pod
apiVersion: v1
metadata:
  name: apple-app
  labels:
    app: apple
spec:
  containers:
    - name: apple-app
      image: hashicorp/http-echo
      args:
        - "-text=apple"
---

kind: Service
apiVersion: v1
metadata:
  name: apple-service
spec:
  selector:
    app: apple
  ports:
    - protocol: TCP
```

```
      port: 80
      targetPort: 5678 # Default port for image
```

```
kubectl apply -f apple.yml
```

```
## banana
## vi banana.yml
kind: Pod
apiVersion: v1
metadata:
  name: banana-app
  labels:
    app: banana
spec:
  containers:
    - name: banana-app
      image: hashicorp/http-echo
      args:
        - "-text=banana"

---

kind: Service
apiVersion: v1
metadata:
  name: banana-service
spec:
  selector:
    app: banana
  ports:
    - port: 80
      targetPort: 5678 # Default port for image
```

```
kubectl apply -f banana.yml
```

Schritt 2:

```
## Ingress
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: example-ingress
  annotations:
    ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
  - http:
      paths:
        - path: /apple
          backend:
            serviceName: apple-service
            servicePort: 80
        - path: /banana
          backend:
            serviceName: banana-service
            servicePort: 80
```

```
## ingress
kubectl apply -f ingress.yml
kubectl get ing
```

Reference

- https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-ingress-guide-nginx-example.html

Find the problem

```
## Hints

## 1. Which resources does our version of kubectl support
## Can we find Ingress as "Kind" here.
kubectl api-ressources

## 2. Let's see, how the configuration works
kubectl explain --api-version=networking.k8s.io/v1 ingress.spec.rules.http.paths.backend.service
```

```
## now we can adjust our config
```

Solution

```
## in kubernetes 1.22.2 - ingress.yml needs to be modified like so.
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
  annotations:
    ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
  - http:
      paths:
        - path: /apple
          pathType: Prefix
          backend:
            service:
              name: apple-service
              port:
                number: 80
        - path: /banana
          pathType: Prefix
          backend:
            service:
              name: banana-service
              port:
                number: 80
```

Install Ingress On Digitalocean DOKS

Achtung: Ingress mit Helm - annotations

Permanente Weiterleitung mit Ingress

Example

```
## redirect.yml
apiVersion: v1
kind: Namespace
metadata:
  name: my-namespace

---

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    nginx.ingress.kubernetes.io/permanent-redirect: https://www.google.de
    nginx.ingress.kubernetes.io/permanent-redirect-code: "308"
  creationTimestamp: null
  name: destination-home
  namespace: my-namespace
spec:
  rules:
  - host: web.training.local
    http:
      paths:
      - backend:
          service:
            name: http-svc
            port:
              number: 80
        path: /source
        pathType: ImplementationSpecific
```

```
Achtung: host-eintrag auf Rechner machen, von dem aus man zugreift

/etc/hosts
45.23.12.12 web.training.local
```

```
curl -I  http://web.training.local/source
HTTP/1.1 308
```

```
Permanent Redirect
```

Umbauen zu google ;o)

```
This annotation allows to return a permanent redirect instead of sending data to the upstream. For example
nginx.ingress.kubernetes.io/permanent-redirect: https://www.google.com would redirect everything to Google.
```

Refs:

- https://github.com/kubernetes/ingress-nginx/blob/main/docs/user-guide/nginx-configuration/annotations.md#permanent-redirect
-

ConfigMap Example

Schritt 1: configmap vorbereiten

```
cd
mkdir -p manifests
cd manifests
mkdir configmaptests
cd configmaptests
nano 01-configmap.yml
```

```
### 01-configmap.yml
kind: ConfigMap
apiVersion: v1
metadata:
  name: example-configmap
data:
  # als Wertepaare
  database: mongodb
  database_uri: mongodb://localhost:27017
```

```
kubectl apply -f 01-configmap.yml
kubectl get cm
kubectl get cm -o yaml
```

Schrit 2: Beispiel als Datei

```
nano 02-pod.yml
```

```
kind: Pod
apiVersion: v1
metadata:
  name: pod-mit-configmap

spec:
  # Add the ConfigMap as a volume to the Pod
  volumes:
    # `name` here must match the name
    # specified in the volume mount
    - name: example-configmap-volume
      # Populate the volume with config map data
      configMap:
        # `name` here must match the name
        # specified in the ConfigMap's YAML
        name: example-configmap

  containers:
    - name: container-configmap
      image: nginx:latest
      # Mount the volume that contains the configuration data
      # into your container filesystem
      volumeMounts:
        # `name` here must match the name
        # from the volumes section of this pod
        - name: example-configmap-volume
          mountPath: /etc/config

```

```
kubectl apply -f 02-pod.yml
```

```
##Jetzt schauen wir uns den Container/Pod mal an
kubectl exec pod-mit-configmap -- ls -la /etc/config
kubectl exec -it pod-mit-configmap --  bash
## ls -la /etc/config
```

Schritt 3: Beispiel. ConfigMap als env-variablen

```
nano 03-pod-mit-env.yml
```

```
## 03-pod-mit-env.yml
kind: Pod
apiVersion: v1
metadata:
  name: pod-env-var
spec:
  containers:
    - name: env-var-configmap
      image: nginx:latest
      envFrom:
        - configMapRef:
            name: example-configmap
```

```
kubectl apply -f 03-pod-mit-env.yml
```

```
## und wir schauen uns das an
##Jetzt schauen wir uns den Container/Pod mal an
kubectl exec pod-env-var -- env
kubectl exec -it pod-env-var --  bash
## env
```

Reference:

- https://matthewpalmer.net/kubernetes-app-developer/articles/ultimate-configmap-guide-kubernetes.html

Configmap MariaDB - Example

Schritt 1: configmap

```
cd
mkdir -p manifests
cd manifests
mkdir cftest
cd cftest
nano 01-configmap.yml
```

```
### 01-configmap.yml
kind: ConfigMap
apiVersion: v1
metadata:
  name: mariadb-configmap
data:
  # als Wertepaare
  MARIADB_ROOT_PASSWORD: 11abc432
```

```
kubectl apply -f .
kubectl get cm
kubectl get cm mariadb-configmap -o yaml
```

Schritt 2: Deployment

```
nano 02-deploy.yml
```

```
##deploy.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mariadb-deployment
spec:
  selector:
    matchLabels:
      app: mariadb
  replicas: 1
  template:
    metadata:
      labels:
        app: mariadb
    spec:
      containers:
      - name: mariadb-cont
        image: mariadb:latest
        envFrom:
```

```
        - configMapRef:
            name: mariadb-configmap
```

```
kubectl apply -f .
```

Important Sidenode

- If configmap changes, deployment does not know
- So kubectl apply -f deploy.yml will not have any effect
- to fix, use stakater/reloader: https://github.com/stakater/Reloader

Configmap MariaDB my.cnf

configmap zu fuss

```
vi mariadb-config2.yml
```

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: example-configmap
data:
  # als Wertepaare
  database: mongodb
  my.cnf: |
 [mysqld]
 slow_query_log = 1
 innodb_buffer_pool_size = 1G
```

```
kubectl apply -f .
```

```
##deploy.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mariadb-deployment
spec:
  selector:
    matchLabels:
      app: mariadb
  replicas: 1
  template:
    metadata:
      labels:
        app: mariadb
    spec:
      containers:
      - name: mariadb-cont
        image: mariadb:latest
        envFrom:
        - configMapRef:
            name: mariadb-configmap

        volumeMounts:
          - name: example-configmap-volume
            mountPath: /etc/my

      volumes:
      - name: example-configmap-volume
        configMap:
          name: example-configmap
```

```
kubectl apply -f .
```

Kubernetes Wartung & Debugging

kubectl drain/uncordon

```
## Achtung, bitte keine pods verwenden, dies können "ge"-drained (ausgetrocknet) werden
kubectl drain <node-name>
z.B.
## Daemonsets ignorieren, da diese nicht gelöscht werden
kubectl drain n17 --ignore-daemonsets

## Alle pods von replicasets werden jetzt auf andere nodes verschoben
```

```
## Ich kann jetzt wartungsarbeiten durchführen

## Wenn fertig bin:
kubectl uncordon n17

## Achtung: deployments werden nicht neu ausgerollt, dass muss ich anstossen.
## z.B.
kubectl rollout restart deploy/webserver
```

Alte manifeste konvertieren mit convert plugin

What is about?

- Plugins needs to be installed seperately on Client (or where you have your manifests)

Walkthrough

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl-convert"
## Validate the checksum
curl -LO "https://dl.k8s.io/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl-convert.sha256"
echo "$(<kubectl-convert.sha256) kubectl-convert" | sha256sum --check
## install
sudo install -o root -g root -m 0755 kubectl-convert /usr/local/bin/kubectl-convert

## Does it work
kubectl convert --help

## Works like so
## Convert to the newest version
## kubectl convert -f pod.yaml
```

Reference

- https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/#install-kubectl-convert-plugin

Kubernetes - RBAC

Nutzer einrichten - kubernetes bis 1.24

Enable RBAC in microk8s

```
## This is important, if not enable every user on the system is allowed to do everything
microk8s enable rbac
```

Schritt 1: Nutzer-Account auf Server anlegen / in Client

```
cd
mkdir -p manifests/rbac
cd manifests/rbac
```

Mini-Schritt 1: Definition für Nutzer

```
## vi service-account.yml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: training
  namespace: default
```

```
kubectl apply -f service-account.yml
```

Mini-Schritt 2: ClusterRolle festlegen - Dies gilt für alle namespaces, muss aber noch zugewiesen werden

```
### Bevor sie zugewiesen ist, funktioniert sie nicht - da sie keinem Nutzer zugewiesen ist

## vi pods-clusterrole.yml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pods-clusterrole
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

```
kubectl apply -f pods-clusterrole.yml
```

Mini-Schritt 3: Die ClusterRolle den entsprechenden Nutzern über RoleBinding zu ordnen

```
## vi rb-training-ns-default-pods.yml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: rolebinding-ns-default-pods
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: pods-clusterrole
subjects:
- kind: ServiceAccount
  name: training
  namespace: default
```

```
kubectl apply -f rb-training-ns-default-pods.yml
```

Mini-Schritt 4: Testen (klappt der Zugang)

```
kubectl auth can-i get pods -n default --as system:serviceaccount:default:training
```

Schritt 2: Context anlegen / Credentials auslesen und in kubeconfig hinterlegen (bis Version 1.25.)

Mini-Schritt 1: kubeconfig setzen

```
kubectl config set-context training-ctx --cluster microk8s-cluster --user training

## extract name of the token from here

TOKEN=`kubectl get secret trainingtoken -o jsonpath='{.data.token}' | base64 --decode`
echo $TOKEN
kubectl config set-credentials training --token=$TOKEN
kubectl config use-context training-ctx

## Hier reichen die Rechte nicht aus
kubectl get deploy
## Error from server (Forbidden): pods is forbidden: User "system:serviceaccount:kube-system:training" cannot list # resource
"pods" in API group "" in the namespace "default"
```

Mini-Schritt 2:

```
kubectl config use-context training-ctx
kubectl get pods
```

Refs:

- https://docs.oracle.com/en-us/iaas/Content/ContEng/Tasks/contengaddingserviceaccttoken.htm
- https://microk8s.io/docs/multi-user
- https://faun.pub/kubernetes-rbac-use-one-role-in-multiple-namespaces-d1d08bb08286

Ref: Create Service Account Token

- https://kubernetes.io/docs/reference/access-authn-authz/service-accounts-admin/#create-token

kubectl

Tipps&Tricks zu Deploymnent - Rollout

Warum

```
Rückgängig machen von deploys, Deploys neu unstossen.
(Das sind die wichtigsten Fähigkeiten
```

Beispiele

```
## Deployment nochmal durchführen
## z.B. nach kubectl uncordon n12.training.local
kubectl rollout restart deploy nginx-deployment

## Rollout rückgängig machen
kubectl rollout undo deploy nginx-deployment
```

Kubernetes - Monitoring (microk8s und vanilla)

metrics-server aktivieren (microk8s und vanilla)

Warum ? Was macht er ?

```
Der Metrics-Server sammelt Informationen von den einzelnen Nodes und Pods
Er bietet mit

kubectl top pods
kubectl top nodes

ein einfaches Interface, um einen ersten Eindruck über die Auslastung zu bekommen.
```

Walktrough

```
## Auf einem der Nodes im Cluster (HA-Cluster)
microk8s enable metrics-server

## Es dauert jetzt einen Moment bis dieser aktiv ist auch nach der Installation
## Auf dem Client
kubectl top nodes
kubectl top pods
```

Kubernetes

- https://kubernetes-sigs.github.io/metrics-server/
- kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml

Kubernetes - Backups

Kubernetes Aware Cloud Backup - kasten.io

Background

- Belongs to veeam (one of the major companies for backup software)

What does Kubernetes Native Backup mean ?

- It is tight into the control plane, so it knows about the objects
- Uses the api to find out about Kubernetes

Setup a storage class (Where to store backup)

- https://docs.kasten.io/latest/install/storage.html#direct-provider-integration

Inject backup into a namespace to be used by app

- https://docs.kasten.io/latest/install/generic.html#using-sidecars

Restore:

```
Restore is done on the K10 - Interface
```

Creating MYSQL - Backup / Restore with Kasten

- TODO: maybe move this to a seperate page
- https://blog.kasten.io/kubernetes-backup-and-restore-for-mysql

Ref:

- https://www.kasten.io
- Installation DigitalOcean
- Installation Kubernetes (Other distributions)

Kubernetes calico (CNI-Plugin)

Debug Container

Walkthrough Debug Container

```
kubectl run ephemeral-demo --image=registry.k8s.io/pause:3.1 --restart=Never
kubectl exec -it ephemeral-demo -- sh

kubectl debug -it ephemeral-demo --image=busybox
```

Example with nginx

```
kubectl run --image=nginx nginx
### debug this container
kubectl debug -it nginx --image=busybox
```

Walkthrough Debug Node

```
kubectl get nodes
kubectl debug node/mynode -it --image=ubuntu
```

Reference

- https://kubernetes.io/docs/tasks/debug/debug-application/debug-running-pod/#ephemeral-container

Kubernetes antrea (CNI-Plugin)

Unterschiede Dokus vmware (antrea mit nsx-t) und OpenSource Antrea

- OpenSource - Version has less features than closed version

Antrea (OpenSource) - Version

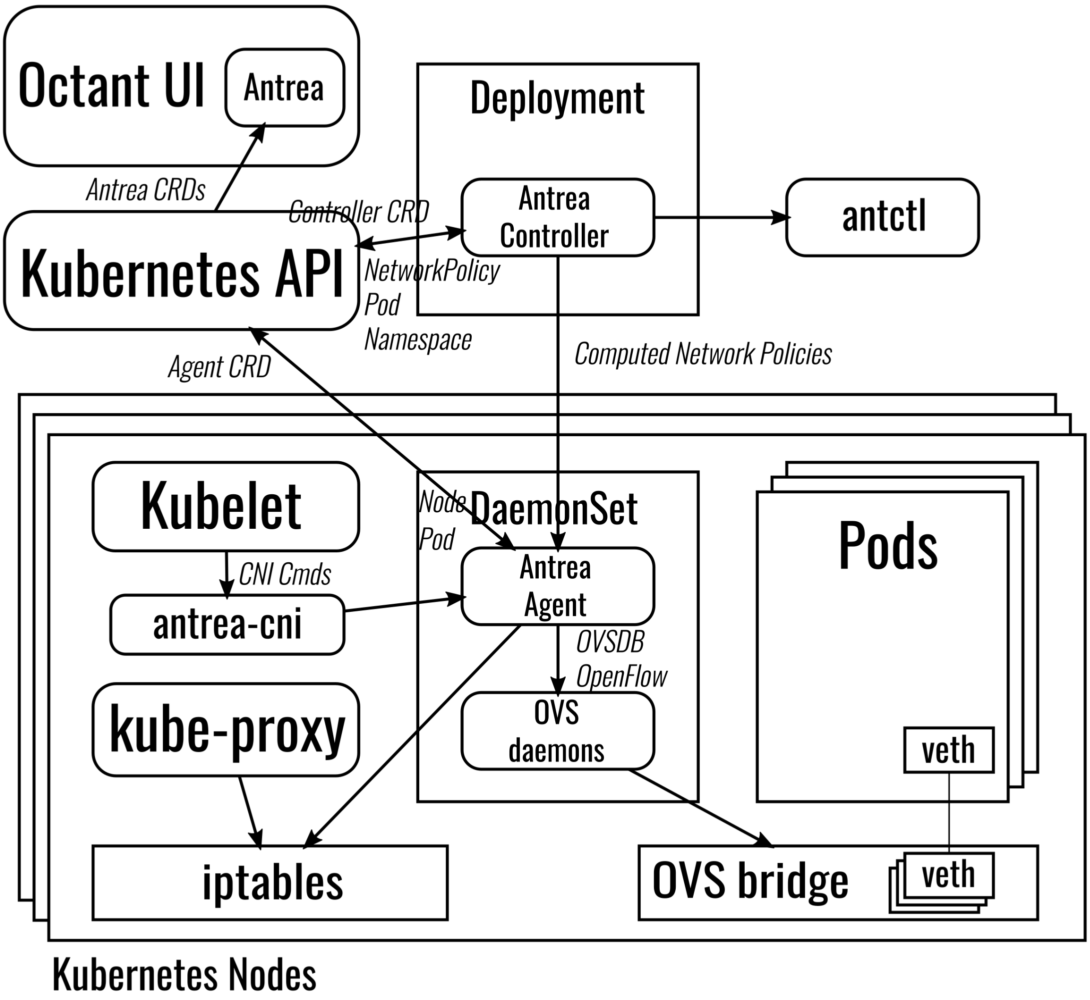
- https://antrea.io/docs/v1.13.2/

vmware - spread across tanzu (AFAIK)

- https://docs.vmware.com/en/VMware-Tanzu-Kubernetes-Grid/2.4/tkg-deploy-mc/mgmt-reqs-network-antrea-tiering.html

Overview Kubernetes Antrea CNI-Plugin

Overview



Basics

- Created by vmware
- Uses Open VShift (virtuell Switches)
- Kernel-Modul openswitch.ko takes care of traffic (performant)

Components

antrea-controller (+api)

- Watches kube-api-server for changes on
  - pod
  - namespaces
  - NetworkPolicy
- Implementation of Controller - API-Server
- Reachable over kube-api-server by implementation https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/apiserver-aggregation/
- Currently only 1 replica is supported
- computes NetworkPolicies and distributes them to the Antrea agents

antrea controller api - part (how authentication works)

- The Controller API server delegates authentication and authorization to the Kubernetes API
- the Antrea Agent uses a Kubernetes ServiceAccount token to authenticate to the Controller,
- the Controller API server validates the token and whether the ServiceAccount is authorized for the API request with the Kubernetes API.

antrea-agent

- Runs on every pod, deployed by Daemonset
- has an endpoint running gRPC which the controller connects to
- Agents connect to controller api by ClusterIP - wit a service Account
- Authentication is done through the kubernetes api server

antctl

- cli for some debugging
- controller-mode on controller (accessing from within controller pod)
- agent-mode on agent (accessing from within agent-pod)
- external also possible - uses kubeconfig to connect
  - Connection is done through kube-api-server

Important antctl commands

```
## on kube-system
kubectl -n kube-system get üpods

antctl get featuregates
```

Reference

- https://antrea.io/docs/v1.3.0/docs/design/architecture/

Antctl

Install (externally as tool (not in pod)): uses .kube/config (Done by trainer)

```
## as root
cd /usr/local/sbin
curl -Lo ./antctl "https://github.com/antrea-io/antrea/releases/download/v1.13.2/antctl-$(uname)-x86_64"
chmod +x ./antctl
```

```
## run as unprivileged user having a .kube/config in homedir
antctl version
```

Shows feature-gates for controller and agent (using antctl client externally)

- Shows both (for controller and for agent), when you do it externally as client-tool from outside pod

```
antctl get featuregates
```

```
Antrea Agent Feature Gates
FEATUREGATE                 STATUS          VERSION
Egress                      Enabled         BETA
EndpointSlice               Enabled         BETA
NetworkPolicyStats          Enabled         BETA
NodePortLocal               Enabled         BETA
Traceflow                   Enabled         BETA
AntreaIPAM                  Disabled        ALPHA
ServiceExternalIP           Disabled        ALPHA
AntreaProxy                 Enabled         BETA
FlowExporter                Disabled        ALPHA
Multicluster                Disabled        ALPHA
AntreaPolicy                Enabled         BETA
Multicast                   Enabled         BETA

Antrea Controller Feature Gates
FEATUREGATE                 STATUS          VERSION
Multicluster                Disabled        ALPHA
AntreaPolicy                Enabled         BETA
Multicast                   Enabled         BETA
Egress                      Enabled         BETA
NetworkPolicyStats          Enabled         BETA
ServiceExternalIP           Disabled        ALPHA
Traceflow                   Enabled         BETA
NodeIPAM                    Enabled         BETA
```

Use antctl from within agent

```
kubectl -n kube-system exec -it ANTREA-AGENT_POD_NAME -n kube-system -c antrea-agent -- bash
```

- or

```
kubectl -n kube-system exec -it daemonset/antrea-agent -n kube-system -c antrea-agent -- bash
```

```
antctl help
antctl log-level
antctl get featuregates
```

Antrea view bridge and config

Finding the bridge

- ovs-vsctl - utility for querying and configuring ovs-vswitchd

```
## How to see the bridge
kubectl -n kube-system exec -it antrea-agent-79bx2 -c antrea-agent -- ovs-vsctl show

## or: always shows the first pod it finds
kubectl -n kube-system exec -it daemonset/antrea-agent -c antrea-agent -- ovs-vsctl show
```

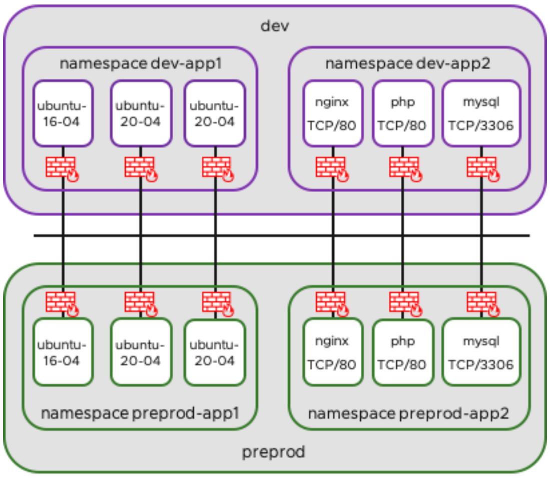
```
708fb906-48b3-4b9e-8508-b4b862389d58
    Bridge br-int
        datapath_type: system
        Port antrea-tun0
            Interface antrea-tun0
                type: geneve
                options: {key=flow, remote_ip=flow}
        Port antrea-gw0
            Interface antrea-gw0
                type: internal
    ovs_version: "2.17.7"
```

Show the configuraton settings of antrea (configmap)

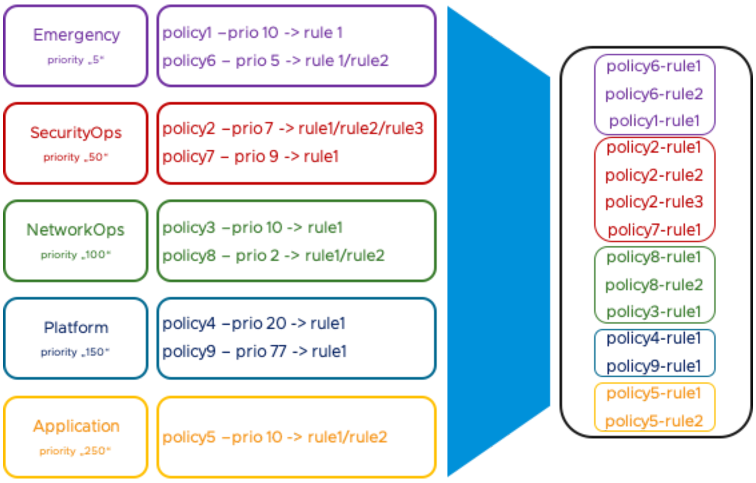
```
kubectl -n kube-system get cm antrea-config -o yaml
```

Antrea Exercise

Our Goal



How the order of priorities work

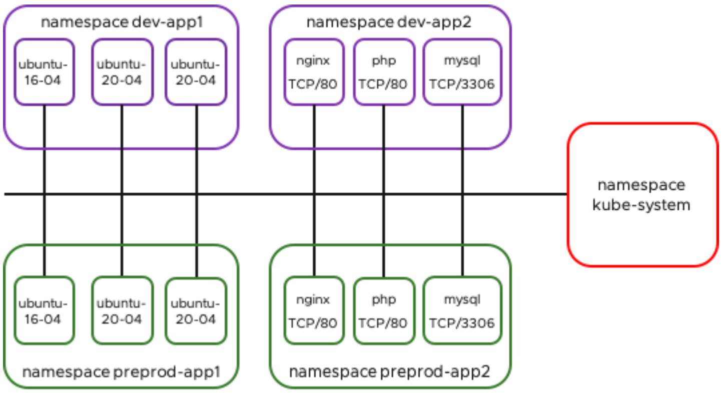


Our Setup

```
In app1 are some Ubuntu Servers for Testing: dev-app1 / preprod-app1

1x Ubuntu Server 16.04
2x Ubuntu Server 20.04
In app2 is a simple 3 Tier-App (WEB-APP-DB): dev-app2 / preprod-app2 (3tier-app)

1x nginx TCP/80 (NodePort)
1x php TCP/80 (ClusterIP)
1x mysql TCP/3306 (ClusterIP)
```



Step 1: Rollout the pods (dev-app1)

- Important - you need to adjust the namespaces as follows:
  - dev-app1- -> z.B. dev-app1-jjm (Deine Initialien)

```
cd
mkdir -p manifests
cd manifests
mkdir 10-antrea
cd 10-antrea
```

```
## nano 01-deployment-dev-app1.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: dev-app1-<name-kurz>
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ubuntu-16-04
  labels:
    app: ubuntu-16-04
  namespace: dev-app1-<name-kurz>
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ubuntu-16-04
  template:
    metadata:
      labels:
        app: ubuntu-16-04
    spec:
      containers:
      - name: ubuntu-16-04
        image: ubuntu:16.04
        imagePullPolicy: IfNotPresent
        command: [ "/bin/bash", "-c" ]
        args:
          - apt-get update;
            apt-get install iputils-ping -y;
            apt-get install net-tools;
            apt-get install curl -y;
            sleep infinity;
---
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: ubuntu-20-04
  labels:
    app: ubuntu-20-04
  namespace: dev-app1-<name-kurz>
spec:
  replicas: 2
  selector:
    matchLabels:
      app: ubuntu-20-04
  template:
    metadata:
      labels:
        app: ubuntu-20-04
    spec:
      containers:
      - name: ubuntu-20-04
        image: ubuntu:20.04
        imagePullPolicy: IfNotPresent
        command: [ "/bin/bash", "-c" ]
        args:
          - apt-get update;
            apt-get install tcpdump -y;
            apt-get install telnet -y;
            apt-get install iputils-ping -y;
            apt-get install nmap -y;
            apt-get install net-tools;
            apt-get install netdiscover -y;
            apt-get install mysql-client -y;
            apt-get install curl -y;
            apt-get install dsniff -y;
            sleep infinity;
```

```
## check if we have replaced all the kurz entries
cat 01-deployment-dev-app1.yaml | grep kurz

kubectl apply -f .
## kubectl -n dev-app1-<name-kurz> get pods
## z.B. kubectl -n dev-app1-jjm get pods
```

Step 2: Rollout the pods (dev-app2)

```
## nano 02-deployment-dev-app2.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: dev-app2-<name-kurz>
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: default-conf
  namespace: dev-app2-<name-kurz>
data:
  default.conf: |
    server {
    listen 80 default_server;

    location / {
      proxy_pass http://app-service;
      proxy_http_version 1.1;
    }

    error_page   500 502 503 504  /50x.html;
    location = /50x.html {
        root   /usr/share/nginx/html;
    }

    }
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  namespace: dev-app2-<name-kurz>
spec:
  replicas: 1
```

```yaml
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
        service: web
        kind: dev
        type: internal
    spec:
      containers:
      - name: nginx
        image: nginx
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
        volumeMounts:
        - mountPath: /etc/nginx/conf.d # mount nginx-conf volumn to /etc/nginx
          readOnly: true
          name: default-conf
        - mountPath: /var/log/nginx
          name: log
      volumes:
      - name: default-conf
        configMap:
          name: default-conf # place ConfigMap `nginx-conf` on /etc/nginx
          items:
            - key: default.conf
              path: default.conf
      - name: log
        emptyDir: {}
---
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: dev-app2-<name-kurz>
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 80
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: appserver
  labels:
    app: app
  namespace: dev-app2-<name-kurz>
spec:
  replicas: 1
  selector:
    matchLabels:
      app: app
  template:
    metadata:
      labels:
        app: app
        kind: dev
        type: internal
    spec:
      containers:
      - name: php-apache
        image: derstich/miserver:006
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: app-service
  labels:
    app: app
```

```
  namespace: dev-app2-<name-kurz>
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    app: app
---
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: mysql
  namespace: dev-app2-<name-kurz>
spec:
  selector:
    matchLabels:
      app: mysql8
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mysql8
        service: db
        kind: dev
        type: internal
    spec:
      containers:
      - image: mysql:5.6
        name: mysql
        imagePullPolicy: IfNotPresent
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: .sweetpwd.
        - name: MYSQL_DATABASE
          value: my_db
        - name: MYSQL_USER
          value: db_user
        - name: MYSQL_PASSWORD
          value: .mypwd
        args: ["--default-authentication-plugin=mysql_native_password"]
        ports:
        - containerPort: 3306
          name: mysql8
---
apiVersion: v1
kind: Service
metadata:
  name: mysql8-service
  labels:
    app: mysql8
  namespace: dev-app2-<name-kurz>
spec:
  type: ClusterIP
  ports:
  - port: 3306
    protocol: TCP
  selector:
    app: mysql8
```

```
kubectl apply -f .
kubectl -n dev-app2-<name-kurz> get all
```

Schritt 3: rollout preprod-app1

```
## nano 03-deployment-preprod-app1.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: preprod-app1-<name-kurz>
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ubuntu-16-04
  labels:
    app: ubuntu-16-04
```

```
  namespace: preprod-app1-<name-kurz>
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ubuntu-16-04
  template:
    metadata:
      labels:
        app: ubuntu-16-04
    spec:
      containers:
      - name: ubuntu-16-04
        image: ubuntu:16.04
        imagePullPolicy: IfNotPresent
        command: [ "/bin/bash", "-c" ]
        args:
          - apt-get update;
            apt-get install iputils-ping -y;
            apt-get install net-tools;
            apt-get install curl -y;
            sleep infinity;
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ubuntu-20-04
  labels:
    app: ubuntu-20-04
  namespace: preprod-app1-<name-kurz>
spec:
  replicas: 2
  selector:
    matchLabels:
      app: ubuntu-20-04
  template:
    metadata:
      labels:
        app: ubuntu-20-04
    spec:
      containers:
      - name: ubuntu-20-04
        image: ubuntu:20.04
        imagePullPolicy: IfNotPresent
        command: [ "/bin/bash", "-c" ]
        args:
          - apt-get update;
            apt-get install tcpdump -y;
            apt-get install telnet -y;
            apt-get install iputils-ping -y;
            apt-get install nmap -y;
            apt-get install net-tools;
            apt-get install netdiscover -y;
            apt-get install mysql-client -y;
            apt-get install curl -y;
            apt-get install dsniff -y;
            sleep infinity;
```

```
kubectl apply -f .
```

Schritt 4: Deploy preprod-app2

```
## nano 04-deployment-preprod-app2.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: preprod-app2-<name-kurz>
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: default-conf
  namespace: preprod-app2-<name-kurz>
data:
  default.conf: |
    server {
    listen 80 default_server;
```

```
    location / {
      proxy_pass http://app-service;
      proxy_http_version 1.1;
    }

    error_page   500 502 503 504  /50x.html;
    location = /50x.html {
        root   /usr/share/nginx/html;
    }

    }
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  namespace: preprod-app2-<name-kurz>
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
        service: web
        kind: dev
        type: internal
    spec:
      containers:
      - name: nginx
        image: nginx
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
        volumeMounts:
        - mountPath: /etc/nginx/conf.d # mount nginx-conf volumn to /etc/nginx
          readOnly: true
          name: default-conf
        - mountPath: /var/log/nginx
          name: log
      volumes:
      - name: default-conf
        configMap:
          name: default-conf # place ConfigMap `nginx-conf` on /etc/nginx
          items:
            - key: default.conf
              path: default.conf
      - name: log
        emptyDir: {}
---
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: preprod-app2-<name-kurz>
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 80
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: appserver
  labels:
    app: app
  namespace: preprod-app2-<name-kurz>
spec:
  replicas: 1
  selector:
    matchLabels:
      app: app
```

```yaml
  template:
    metadata:
      labels:
        app: app
        kind: dev
        type: internal
    spec:
      containers:
      - name: php-apache
        image: derstich/miserver:005
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: app-service
  labels:
    app: app
  namespace: preprod-app2-<name-kurz>
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    app: app
---
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: mysql
  namespace: preprod-app2-<name-kurz>
spec:
  selector:
    matchLabels:
      app: mysql8
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mysql8
        service: db
        kind: dev
        type: internal
    spec:
      containers:
      - image: mysql:5.6
        name: mysql
        imagePullPolicy: IfNotPresent
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: .sweetpwd.
        - name: MYSQL_DATABASE
          value: my_db
        - name: MYSQL_USER
          value: db_user
        - name: MYSQL_PASSWORD
          value: .mypwd
        args: ["--default-authentication-plugin=mysql_native_password"]
        ports:
        - containerPort: 3306
          name: mysql8
---
apiVersion: v1
kind: Service
metadata:
  name: mysql8-service
  labels:
    app: mysql8
  namespace: preprod-app2-<name-kurz>
spec:
  type: ClusterIP
  ports:
  - port: 3306
    protocol: TCP
```

```
  selector:
    app: mysql8
```

```
kubectl apply -f .
```

Schritt 5: Daten auslesen

```
## Das bitte anpassen
KURZ=jm
```

```

## dev-app1
kubectl -n dev-app1-$KURZ get pods -o=custom-columns=NAMESPACE:.metadata.namespace,NAME:.metadata.name,STATUS:.status.phase,IP:.status.podIP,NODE:.spec.nodeName

## dev-app2
kubectl -n dev-app2-$KURZ get pods -o=custom-columns=NAMESPACE:.metadata.namespace,NAME:.metadata.name,STATUS:.status.phase,IP:.status.podIP,NODE:.spec.nodeName

## preprod-app1
kubectl -n preprod-app1-$KURZ get pods -o=custom-columns=NAMESPACE:.metadata.namespace,NAME:.metadata.name,STATUS:.status.phase,IP:.status.podIP,NODE:.spec.nodeName

## preprod-app2
kubectl -n preprod-app2-$KURZ get pods -o=custom-columns=NAMESPACE:.metadata.namespace,NAME:.metadata.name,STATUS:.status.phase,IP:.status.podIP,NODE:.spec.nodeName
```

```
## BITTE die Infos zwischen speichern oder Screenshot machen
```

Schritt 6: Zugriff auf dev-app2 klären

```
## Das ändern
KURZ=jm
```

```
kubectl get svc -n dev-app2-$KURZ nginx
```

```
tln1@k8s-client:~/manifests/10-antrea$ kubectl get svc -n dev-app2-$KURZ nginx
NAME    TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
nginx   NodePort   10.101.253.56   <none>        80:32767/TCP   25m
```

```
curl -i http://10.135.0.5:32767
## oder im Browser mit Public - IP
```

Schritt 7: Zugriff auf preprod-app klären

```
## Das ändern
KURZ=jm
```

```
kubectl get svc -n preprod-app2-$KURZ nginx
```

```
NAME    TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
nginx   NodePort   10.106.173.151   <none>        80:31836/TCP   14m
```

```
curl -i http://10.135.0.5:31836
```

Schritt 8: Zugriff ohne antrea policy testen

```
KURZ=jm
kubectl exec -it -n dev-app1-$KURZ deployment/ubuntu-20-04 -- /bin/bash
```

```
## scannen des netzes
nmap 10.244.0.0/22
```

```
Nmap scan report for 10.244.3.18
Host is up (0.0038s latency).
All 1000 scanned ports on 10.244.3.18 are closed

Nmap scan report for 10-244-3-19.nginx.preprod-app2-jm.svc.cluster.local (10.244.3.19)
Host is up (0.0032s latency).
Not shown: 999 closed ports
PORT   STATE SERVICE
80/tcp open  http

Nmap scan report for 10-244-3-20.mysql8-service.preprod-app2-jm.svc.cluster.local (10.244.3.20)
Host is up (0.0031s latency).
Not shown: 999 closed ports
PORT     STATE SERVICE
3306/tcp open  mysql

Nmap done: 1024 IP addresses (44 hosts up) scanned in 15.46 seconds
```

- Namen werden aufgelöst (rückwärtig)
- alle ports sind einsehbar
- Verbindung funktioniert nach überall

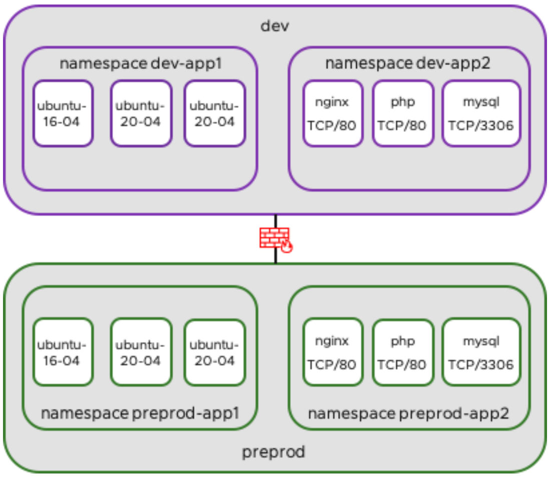
```
## mysql preprod herausfinden
nmap 10.244.0.0/22 | grep mysql | grep preprod
```

```
root@ubuntu-20-04-66598645fd-4gsjg:/# nmap 10.244.0.0/22 | grep mysql | grep preprod
Nmap scan report for 10-244-3-20.mysql8-service.preprod-app2-jm.svc.cluster.local (10.244.3.20)
```

```
## Oh, wir haben das Passwort herausgefunden (Social Engineering ;o))
.sweetpwd.
```

```
mysql -h 10-244-3-20.mysql8-service.preprod-app2-jm.svc.cluster.local -p
```

Schritt 9: Isolate dev and preprod




```
## entsprechend anpassen
KURZ=jm
```

```
## Namspaces labeln
kubectl label ns dev-app1-$KURZ env=dev-$KURZ ns=dev-app1-$KURZ
kubectl label ns dev-app2-$KURZ env=dev-$KURZ ns=dev-app2-$KURZ
kubectl label ns preprod-app1-$KURZ env=preprod-$KURZ ns=preprod-app1-$KURZ
kubectl label ns preprod-app2-$KURZ env=preprod-$KURZ ns=preprod-app2-$KURZ

kubectl describe ns dev-app1-$KURZ
```

```
## now create the policy
## nano 10-deny-dev-to-preprod.yaml
apiVersion: crd.antrea.io/v1beta1
kind: ClusterNetworkPolicy
metadata:
  name: deny-dev-to-preprod-<name-kurz>
```

```
spec:
    priority: 100
    tier: SecurityOps
    appliedTo:
      - namespaceSelector:
          matchLabels:
            env: preprod-<name-kurz>
    ingress:
      - action: Drop
        from:
          - namespaceSelector:
              matchLabels:
                env: dev-<name-kurz>
```

```
KURZ=jm
## Test ob ping von preprod nach dev funktioniert
## Hier ein POD-IP raussuchen
kubectl -n dev-app1-$KURZ get pods -o wide
kubectl -n preprod-app1-$KURZ exec deployments/ubuntu-20-04 -- ping 10.244.3.15

## Test ob ping von dev nach preprod funktioniert - der sollte nicht funktionieren
## Hier eine POD-IP rausschen
kubectl -n preprod-app1-$KURZ get pods -o wide
kubectl -n dev-app1-$KURZ exec deployments/ubuntu-20-04 -- ping 10.244.2.25
```

```
## ClusterNetworkPolicy anwenden
kubectl apply -f .
```

```
## Jetzt nochmal die Pings testen von oben
## ---> Ping ist immer noch möglich --> da keine Firewall - Regel
kubectl -n preprod-app1-$KURZ exec deployments/ubuntu-20-04 -- ping 10.244.3.15

## in die andere Richtung geht es aber nicht !!
kubectl -n dev-app1-$KURZ exec deployments/ubuntu-20-04 -- ping 10.244.2.25
```

```
## ok jetzt in die andere richtung
## nano 15-deny-preprod-to-dev.yaml
apiVersion: crd.antrea.io/v1beta1
kind: ClusterNetworkPolicy
metadata:
  name: deny-preprod-to-dev-<kurz-name>
spec:
    priority: 101
    tier: SecurityOps
    appliedTo:
      - namespaceSelector:
          matchLabels:
            env: dev-<name-kurz>
    ingress:
      - action: Drop
        from:
          - namespaceSelector:
              matchLabels:
                env: preprod-<name-kurz>
```

```
kubectl apply -f .
kubectl get clusternetworkpolicies
```

```
## Only output
NAME                     TIER          PRIORITY   DESIRED NODES   CURRENT NODES   AGE
deny-dev-to-preprod-jm   SecurityOps   100        2               2               16m
deny-preprod-to-dev      SecurityOps   101        2               2               3m15s
```

```
## und jetzt geht pingen in die andere Richtung auch nicht mehr
kubectl -n preprod-app1-$KURZ exec deployments/ubuntu-20-04 -- ping 10.244.3.15
```

Schritt 11: Isolate Pods (only within the namespaces)

- Aktuell ist das ping vom preprod-app1- zum preprod-app2- namespace noch möglich
- Das wollen wir einschränken
- Ausserdem von dev-app1- zu dev-app2-

```
## bei dir anpassen
KURZ=jm
```

```
## So sehen unsere Namespace - Labels aus
kubectl describe namespace dev-app1-$KURZ
```

```
## Ausgabe, z.B.
Name:         dev-app1-jm
Labels:       env=dev-jm
              ns=dev-app1-jm
```

```
## nano 20-allow-ns-dev-app1-dev-app1.yaml
## Traffic innerhalb des Namespaces erlaubt

apiVersion: crd.antrea.io/v1beta1
kind: ClusterNetworkPolicy
metadata:
  name: 20-allow-ns-dev-app1-dev-app1-<name-kurz>
spec:
    priority: 100
    tier: application
    appliedTo:
      - namespaceSelector:
          matchLabels:
            ns: dev-app1-<name-kurz>
    ingress:
      - action: Allow
        from:
          - namespaceSelector:
              matchLabels:
                ns: dev-app1-<name-kurz>
```

```
kubectl apply -f .
```

```
## nano 25-drop-any-ns-dev-app2.yaml
## allen anderen Traffic zum namespace app2 hin verbieten aus anderen namespaces
apiVersion: crd.antrea.io/v1beta1
kind: ClusterNetworkPolicy
metadata:
  name: 25-drop-any-ns-dev-app2-<name-kurz>
spec:
    priority: 110
    tier: application
    appliedTo:
      - namespaceSelector:
          matchLabels:
            ns: dev-app2-<name-kurz>
    ingress:
      - action: Drop
        from:
          - namespaceSelector: {}
```

```
kubectl apply -f .
```

```
## nano 30-allow-ns-preprod-app1-preprod-app1.yaml
## Same for preprod-app1
## Allow all traffic within namespace
apiVersion: crd.antrea.io/v1beta1
kind: ClusterNetworkPolicy
metadata:
  name: 30-allow-ns-preprod-app1-preprod-app1-<name-kurz>
spec:
    priority: 120
    tier: application
    appliedTo:
      - namespaceSelector:
          matchLabels:
            ns: preprod-app1-<name-kurz>
    ingress:
      - action: Allow
        from:
          - namespaceSelector:
              matchLabels:
                ns: preprod-app1-<name-kurz>
```

```
kubectl apply -f .
```

```
## disallow all traffic from other namespaces to prepr
## nano 35-drop-any-ns-preprod-app2.yaml
apiVersion: crd.antrea.io/v1beta1
kind: ClusterNetworkPolicy
metadata:
  name: 21-drop-any-ns-preprod-app2<name-kurz>
spec:
    priority: 130
    tier: application
    appliedTo:
      - namespaceSelector:
          matchLabels:
            ns: preprod-app2-<name-kurz>
    ingress:
      - action: Drop
        from:
          - namespaceSelector: {}
```

```
kubectl apply -f .
```

Schritt 12: Isolate traffic within app2 - namespaces (3-Tier-app) (Das kann leider nur er Trainer machen ;o() - wg der Labels

```
## For dev-app2-<name-kurz> we want
web->app (80)
app->db (3306)
drop everything else
```

```
KURZ=jm;
```

```
kubectl -n dev-app2-$KURZ describe pods | head -n 20
kubectl -n preprod-app2-$KURZ describe pods | head -n 20
```

```
Name:             appserver-8596ff696-14bpm
Namespace:        dev-app2-jm
Priority:         0
Service Account:  default
Node:             worker3/10.135.0.8
Start Time:       Wed, 29 Nov 2023 04:44:37 +0000
Labels:           app=app
                  kind=dev
                  pod-template-hash=8596ff696
```

- we are using the label app=xxx

```
## nano 40-allow-web-app.yaml
apiVersion: crd.antrea.io/v1beta1
kind: ClusterNetworkPolicy
metadata:
  name: 40-allow-web-app-<name-kurz>
spec:
    priority: 10
    tier: application
    appliedTo:
      - podSelector:
          matchLabels:
            app: app
    ingress:
      - action: Allow
        from:
          - podSelector:
              matchLabels:
                app: nginx
        ports:
          - protocol: TCP
            port: 80
```

```
kubectl apply -f  .
```

```
## nano 45-allow-app-db.yaml
apiVersion: crd.antrea.io/v1beta1
```

```
kind: ClusterNetworkPolicy
metadata:
  name: 02-allow-app-db-<name-kurz>
spec:
    priority: 20
    tier: application
    appliedTo:
      - podSelector:
          matchLabels:
            app: mysql8
    ingress:
      - action: Allow
        from:
          - podSelector:
              matchLabels:
                app: app
        ports:
          - protocol: TCP
            port: 3306
```

```
kubectl apply -f .
```

```
## nano 50-deny-any-to-app2.yaml
## Deny everything else
apiVersion: crd.antrea.io/v1beta1
kind: ClusterNetworkPolicy
metadata:
  name: 03-deny-any-to-app2-<name-kurz>
spec:
    priority: 30
    tier: application
    appliedTo:
      - namespaceSelector:
          matchLabels:
                  ns: dev-app2-<name-kurz>
      - namespaceSelector:
          matchLabels:
                  ns: preprod-app2-<name-kurz>
    ingress:
      - action: Drop
        from:
          - namespaceSelector: {}
```

```
kubectl apply -f .
```

Schritt 13: Usage of the Emergency Tier - e.g. Attack (only Trainer)

- We have problems with Ubuntu 16.04. an we want to isolate it.

```
kubectl get tiers
```

```
## nano 80-emergency.yaml
apiVersion: crd.antrea.io/v1beta1
kind: ClusterNetworkPolicy
metadata:
  name: 50-deny-any-pod-ubuntu16-<name-kurz>
spec:
    priority: 50
    tier: emergency
    appliedTo:
      - podSelector:
          matchLabels:
                  app: ubuntu-16-04
    ingress:
      - action: Drop
        from:
          - namespaceSelector: {}
```

```
kubectl apply -f .
```

- Because Emergency has the highest priority, the policy in application (allow any in ns-app1) has no Impact anymore.

Reference:

- https://www.vrealize.it/2020/09/28/securing-you-k8s-network-with-antrea-clusternetworkpolicy/

Kubernetes Interna / Misc.

OCI,Container,Images Standards

Schritt 1:

```
cd
mkdir bautest
cd bautest
```

Schritt 2:

```
## nano docker-compose.yml
version: "3.8"

services:
  myubuntu:
    build: ./myubuntu
    restart: always
```

Schritt 3:

```
mkdir myubuntu
cd myubuntu
```

```
nano hello.sh
```

```
##!/bin/bash
let i=0

while true
do
  let i=i+1
  echo $i:hello-docker
  sleep 5
done
```

```
## nano Dockerfile
FROM ubuntu:latest
RUN apt-get update; apt-get install -y inetutils-ping
COPY hello.sh .
RUN chmod u+x hello.sh
CMD ["/hello.sh"]
```

Schritt 4:

```
cd ../
## wichtig, im docker-compose - Ordner seiend
##pwd
##~/bautest
docker-compose up -d
## wird image gebaut und container gestartet

## Bei Veränderung vom Dockerfile, muss man den Parameter --build mitangeben
docker-compose up -d --build
```

Geolocation Kubernetes Cluster

- https://learnk8s.io/bite-sized/connecting-multiple-kubernetes-clusters

statische IP für Pod in calico

- https://docs.tigera.io/calico/latest/networking/ipam/use-specific-ip

yaml linting

- https://www.kubeval.com/installation/

ssl terminierung über proxy nginx

mit ssl

- https://jackiechen.blog/2019/01/24/nginx-sample-config-of-http-and-ldaps-reverse-proxy/

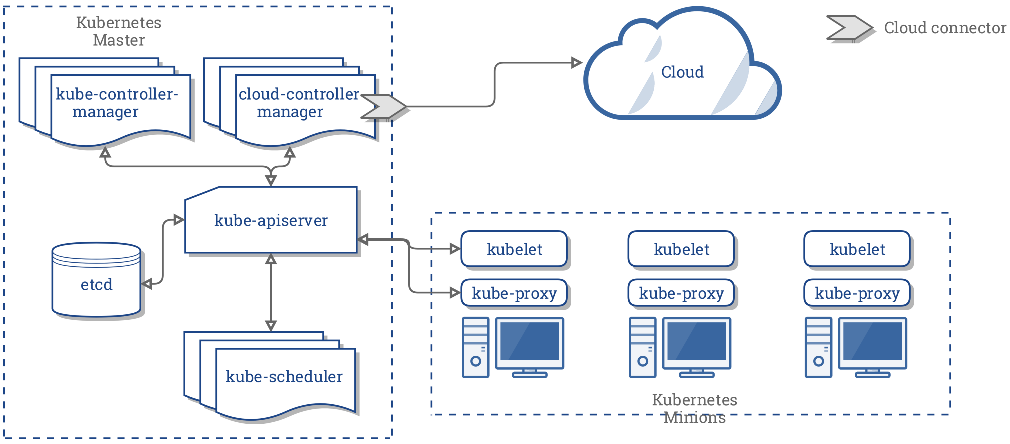
Ohne ssl

- https://kubernetes.github.io/ingress-nginx/user-guide/exposing-tcp-udp-services/

LoadBalancer / Cluster Controller Manager

Keypart: Cluster Controller Manager (CCM)

- was decoupled from Kube Controller Manager
    - to make it easier for cloud providers to implement their specific environment/workings (e.g. LoadBalancer)
- To do this a skeleton was provided.



Control Loops in the CCM

- Der CCM erbt seine Funktionen von Komponenten des Kubernetes, die von einem Cloud Provider abhängig sind.
- Die meisten Funktionen des CCM stammen aus dem KCM. Wie im vorherigen Abschnitt erwähnt, führt das CCM die folgenden Steuerschleifen durch:

```
Node Controller
Route Controller
Service Controller
```

Service Controller

```
Der Service Controller ist verantwortlich für das Abhören von Ereignissen zum Erstellen, Aktualisieren und Löschen von Diensten.
Basierend auf dem aktuellen Stand der Services in Kubernetes konfiguriert es Cloud Load Balancer (wie ELB, Google LB oder Oracle
Cloud Infrastructure LB), um den Zustand der Services in Kubernetes abzubilden. Darüber hinaus wird sichergestellt, dass die
Service Backends für Cloud Loadbalancer auf dem neuesten Stand sind.
```

Load Balancer Implementation in DigitalOcean (DO)

- https://github.com/digitalocean/digitalocean-cloud-controller-manager/tree/master
- https://github.com/digitalocean/digitalocean-cloud-controller-manager/blob/master/cloud-controller-manager/do/loadbalancers.go

api - domain is hardcoded in cloud controller manager for digitalocean

```
jmetzger@powerhouse:~$ grep -ir "api.digitalocean.com" .
./digitalocean-cloud-controller-manager/vendor/github.com/digitalocean/godo/godo.go:     defaultBaseURL = "https://api.di
gitalocean.com/"
```

References:

- Good explanation
- Zugrundeliegende Konzepte

Kubernetes - Shared Volumes

Shared Volumes with nfs

Create new server and install nfs-server

```
## on Ubuntu 20.04LTS
apt install nfs-kernel-server
systemctl status nfs-server

vi /etc/exports
## adjust ip's of kubernetes master and nodes
## kmaster
/var/nfs/ 192.168.56.101(rw,sync,no_root_squash,no_subtree_check)
## knode1
/var/nfs/ 192.168.56.103(rw,sync,no_root_squash,no_subtree_check)
## knode 2
/var/nfs/ 192.168.56.105(rw,sync,no_root_squash,no_subtree_check)
```

```
exportfs -av
```

On all nodes (needed for production)

```
##
apt install nfs-common
```

On all nodes (only for testing)

```
#### Please do this on all servers (if you have access by ssh)
### find out, if connection to nfs works !

## for testing
mkdir /mnt/nfs
## 10.135.0.18 is our nfs-server
mount -t nfs 10.135.0.18:/var/nfs /mnt/nfs
ls -la /mnt/nfs
umount /mnt/nfs
```

Persistent Storage-Step 1: Setup PersistentVolume in cluster

```
cd
cd manifests
mkdir -p nfs
cd nfs
nano 01-pv.yml
```

```
apiVersion: v1
kind: PersistentVolume
metadata:
  # any PV name
  name: pv-nfs-tln<nr>
  labels:
    volume: nfs-data-volume-tln<nr>
spec:
  capacity:
    # storage size
    storage: 1Gi
  accessModes:
    # ReadWriteMany(RW from multi nodes), ReadWriteOnce(RW from a node), ReadOnlyMany(R from multi nodes)
    - ReadWriteMany
  persistentVolumeReclaimPolicy:
    # retain even if pods terminate
    Retain
  nfs:
    # NFS server's definition
    path: /var/nfs/tln<nr>/nginx
    server: 10.135.0.18
    readOnly: false
  storageClassName: ""
```

```
kubectl apply -f 01-pv.yml
kubectl get pv
```

Persistent Storage-Step 2: Create Persistent Volume Claim

```
nano 02-pvc.yml
```

```
## vi 02-pvc.yml
## now we want to claim space
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pv-nfs-claim-tln<nr>
spec:
  storageClassName: ""
  volumeName: pv-nfs-tln<nr>
  accessModes:
  - ReadWriteMany
  resources:
     requests:
       storage: 1Gi
```

```
kubectl apply -f 02-pvc.yml
kubectl get pvc
```

Persistent Storage-Step 3: Deployment

```
## deployment including mount
## vi 03-deploy.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 4 # tells deployment to run 4 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:

      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80

        volumeMounts:
          - name: nfsvol
            mountPath: "/usr/share/nginx/html"

      volumes:
      - name: nfsvol
        persistentVolumeClaim:
          claimName: pv-nfs-claim-tln<tln>
```

```
kubectl apply -f 03-deploy.yml
```

Persistent Storage Step 4: service

```
## now testing it with a service
## cat 04-service.yml
apiVersion: v1
kind: Service
metadata:
  name: service-nginx
  labels:
    run: svc-my-nginx
spec:
  type: NodePort
  ports:
  - port: 80
    protocol: TCP
  selector:
    app: nginx
```

```
kubectl apply -f 04-service.yml
```

Persistent Storage Step 5: write data and test

```
## connect to the container and add index.html - data
kubectl exec -it deploy/nginx-deployment -- bash
## in container
echo "hello dear friend" > /usr/share/nginx/html/index.html
exit

## now try to connect
kubectl get svc

## connect with ip and port
kubectl run -it --rm curly --image=curlimages/curl -- /bin/sh
## curl http://<cluster-ip>
## exit
```

```
## now destroy deployment
kubectl delete -f 03-deploy.yml

## Try again - no connection
kubectl run -it --rm curly --image=curlimages/curl -- /bin/sh
## curl http://<cluster-ip>
## exit
```

Persistent Storage Step 6: retest after redeployment

```
## now start deployment again
kubectl apply -f 03-deploy.yml

## and try connection again
kubectl run -it --rm curly --image=curlimages/curl -- /bin/sh
## curl http://<cluster-ip>
## exit
```

Kubernetes QoS

Quality of Service - evict pods

Die Class wird auf Basis der Limits und Requests der Container vergeben

```
Request: Definiert wieviel ein Container mindestens braucht (CPU,memory)
Limit: Definiert, was ein Container maximal braucht.

in spec.containers.resources
kubectl explain pod.spec.containers.resources
```

Art der Typen:

- Guaranteed
- Burstable
- BestEffort

Guaranteed

```
Type: Guaranteed:
https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/#create-a-pod-that-gets-assigned-a-qos-class-of-guaranteed

set when limit equals request
(request: das braucht er,
limit: das braucht er maximal)

Garantied ist die höchste Stufe und diese werden bei fehlenden Ressourcen
als letztes "evicted"

apiVersion: v1

kind: Pod
metadata:
  name: qos-demo
  namespace: qos-example
spec:
  containers:
  - name: qos-demo-ctr
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "700m"

      requests:
        memory: "200Mi"
        cpu: "700m"
```

Kustomize

Kustomize Overlay Beispiel

Konzept Overlay

- Base + Overlay = Gepatchtes manifest
- Sachen patchen.

  * Die werden drübergelegt.

Example 1: Walkthrough

```
## Step 1:
## Create the structure
## kustomize-example1
## L base
## | - kustomization.yml
## L overlays
##.    L dev
##       - kustomization.yml
##.    L prod
##.      - kustomization.yml
cd; mkdir -p manifests/kustomize-example1/base; mkdir -p manifests/kustomize-example1/overlays/prod; cd manifests/kustomize-
example1
```

```
## Step 2: base dir with files
## now create the base kustomization file
## vi base/kustomization.yml
resources:
- service.yml
```

```
## Step 3: Create the service - file
## vi base/service.yml
kind: Service
apiVersion: v1
metadata:
  name: service-app
spec:
  type: ClusterIP
  selector:
    app: simple-app
  ports:
  - name: http
    port: 80
```

```
## See how it looks like
kubectl kustomize ./base
```

```
## Step 4: create the customization file accordingly
##vi overlays/prod/kustomization.yaml
bases:
- ../../base
patches:
- service-ports.yaml
```

```
## Step 5: create overlay (patch files)
## vi overlays/prod/service-ports.yaml
kind: Service
apiVersion: v1
metadata:
  #Name der zu patchenden Ressource
  name: service-app
spec:
  # Changed to Nodeport
  type: NodePort
  ports: #Die Porteinstellungen werden überschrieben
  - name: https
    port: 443
```

```
## Step 6:
kubectl kustomize overlays/prod

## or apply it directly
kubectl apply -k overlays/prod/
```

```
## Step 7:
## mkdir -p overlays/dev
## vi overlays/dev/kustomization
bases:
- ../../base
```

```
## Step 8:
## statt mit der base zu arbeiten
kubectl kustomize overlays/dev
```

Example 2: Advanced Patching with patchesJson6902 (You need to have done example 1 firstly)

```
## Schritt 1:
## Replace overlays/prod/kustomization.yml with the following syntax
bases:
- ../../base
patchesJson6902:
- target:
    version: v1
    kind: Service
    name: service-app
  path: service-patch.yaml
```

```
## Schritt 2:
## vi overlays/prod/service-patch.yaml
- op: remove
  path: /spec/ports
  value:
  - name: http
    port: 80
- op: add
  path: /spec/ports
  value:
  - name: https
    port: 443
```

```
## Schritt 3:
kubectl kustomize overlays/prod
```

Special Use Case: Change the metadata.name

```
## Same as Example 2, but patch-file is a bit different
## vi overlays/prod/service-patch.yaml
- op: remove
  path: /spec/ports
  value:
  - name: http
    port: 80

- op: add
  path: /spec/ports
  value:
  - name: https
    port: 443

- op: replace
  path: /metadata/name
  value: svc-app-test
```

```
kubectl kustomize overlays/prod
```

Ref:

- https://blog.ordix.de/kubernetes-anwendungen-mit-kustomize

Helm mit kustomize verheiraten

Option 1: helm chart entpacken und das helm chart patchen

```
helm add repo bitnami ....
helm template --base-directory=base bitnami/mysql
## patchen kustomize
kustomize build overlay/prod
kubectl apply -k overlay/prod
```

Option 2: packe helm chart aus

```
## pull
helm pull
tar xvf mysql-9.0.34.tgz
## templates werden
kubectl kustomize build overlay/prod
helm install mysql-release mysql # 2. mysql wäre das chart-verzeichnis lokal im filesystem
## Vorteile
```

```
## ich kann das ganze auch wieder so installieren
## ich kann ein update durch führen
```

Option 3: helm --post-renderer

```
## erst wird template erstellt und dann dann ein weiteres script ausgeführt
## und dann erst installiert.
helm install --post-renderer=./patch.sh

## im shell-script
## kubectl kustomize
## https://austindewey.com/2020/07/27/patch-any-helm-chart-template-using-a-kustomize-post-renderer/
```

Option 4: kustomize lädt helm - chart

```
## kustomization.yml
https://github.com/kubernetes-sigs/kustomize/blob/master/examples/chart.md
```

Kubernetes - Tipps & Tricks

Kubernetes Debuggen ClusterIP/PodIP

Situation

- Kein Zugriff auf die Nodes, zum Testen von Verbindungen zu Pods und Services über die PodIP/ClusterIP

Lösung

```
## Wir starten eine Busybox und fragen per wget und port ab
## busytester ist der name
## long version
kubectl run -it --rm --image=busybox busytester
## wget <pod-ip-des-ziels>
## exit


## quick and dirty
kubectl run -it --rm --image=busybox busytester -- wget <pod-ip-des-ziels>
```

Debugging pods

How ?

1. Which pod is in charge
2. Problems when starting: kubectl describe po mypod
3. Problems while running: kubectl logs mypod

Taints und Tolerations

Taints

```
Taints schliessen auf einer Node alle Pods aus, die nicht bestimmte taints haben:

Möglichkeiten:

o Sie werden nicht gescheduled - NoSchedule
o Sie werden nicht executed - NoExecute
o Sie werden möglichst nicht gescheduled. - PreferNoSchedule
```

Tolerations

```
Tolerations werden auf Pod-Ebene vergeben:
tolerations:

Ein Pod kann (wenn es auf einem Node taints gibt), nur
gescheduled bzw. ausgeführt werden, wenn er die
Labels hat, die auch als
Taints auf dem Node vergeben sind.
```

Walkthrough

Step 1: Cordon the other nodes - scheduling will not be possible there

```
## Cordon nodes n11 and n111
## You will see a taint here
kubectl cordon n11
```

```
kubectl cordon n111
kubectl describe n111 | grep -i taint
```

Step 2: Set taint on first node

```
kubectl taint nodes n1 gpu=true:NoSchedule
```

Step 3

```
cd
mkdir -p manifests
cd manifests
mkdir tainttest
cd tainttest
nano 01-no-tolerations.yml
```

```
##vi 01-no-tolerations.yml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-test-no-tol
  labels:
    env: test-env
spec:
  containers:
  - name: nginx
    image: nginx:1.21
```

```
kubectl apply -f .
kubectl get po nginx-test-no-tol
kubectl get describe nginx-test-no-tol
```

Step 4:

```
## vi 02-nginx-test-wrong-tol.yml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-test-wrong-tol
  labels:
    env: test-env
spec:
  containers:
  - name: nginx
    image: nginx:latest
  tolerations:
  - key: "cpu"
    operator: "Equal"
    value: "true"
    effect: "NoSchedule"
```

```
kubectl apply -f .
kubectl get po nginx-test-wrong-tol
kubectl describe po nginx-test-wrong-tol
```

Step 5:

```
## vi 03-good-tolerations.yml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-test-good-tol
  labels:
    env: test-env
spec:
  containers:
  - name: nginx
    image: nginx:latest
  tolerations:
  - key: "gpu"
    operator: "Equal"
    value: "true"
    effect: "NoSchedule"
```

```
kubectl apply -f .
kubectl get po nginx-test-good-tol
kubectl describe po nginx-test-good-tol
```

Taints rausnehmen

```
kubectl taint nodes n1 gpu:true:NoSchedule-
```

uncordon other nodes

```
kubectl uncordon n11
kubectl uncordon n111
```

References

- Doku Kubernetes Taints and Tolerations
- https://blog.kubecost.com/blog/kubernetes-taints/

Autoscaling Pods/Deployments

Example: newest version with autoscaling/v2 used to be hpa/v1

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello
spec:
  replicas: 3
  selector:
    matchLabels:
      app: hello
  template:
    metadata:
      labels:
        app: hello
    spec:
      containers:
      - name: hello
        image: k8s.gcr.io/hpa-example
        resources:
          requests:
            cpu: 100m
---
kind: Service
apiVersion: v1
metadata:
  name: hello
spec:
  selector:
    app: hello
  ports:
    - port: 80
      targetPort: 80
---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: hello
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: hello
  minReplicas: 2
  maxReplicas: 20
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 80
```

- https://docs.digitalocean.com/tutorials/cluster-autoscaling-ca-hpa/

Reference

- https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/#autoscaling-on-more-specific-metrics
- https://medium.com/expedia-group-tech/autoscaling-in-kubernetes-why-doesnt-the-horizontal-pod-autoscaler-work-for-me-5f0094694054

pod aus deployment bei config - Änderung neu ausrollen

- https://github.com/stakater/Reloader

Assigning Pods to Nodes

Walkthrough

```
## leave n3 as is
kubectl label nodes worker1 machine=worker1
kubectl label nodes worker2 machine=worker2
kubectl get nodes --show-labels
```

1. Deployment auf worker1

```
cd
mkdir -p manifests
cd manifests
mkdir calicotest
cd calicotest
```

```
nano 01-deploy.yml
```

```
## nginx-deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment-calicotest
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
      nodeSelector:
        machine: worker1
```

2. noch ein Pod auf worker1

```
nano 02-pod.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-calicotest
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    machine: worker1
```

```
kubectl apply -f .
```

Kubernetes - Hardening

Kubernetes Tipps Hardening

PSA (Pod Security Admission)

```
Policies defined by namespace.
e.g. not allowed to run container as root.

Will complain/deny when creating such a pod with that container type
```

Möglichkeiten in Pods und Containern

```
## für die Pods
kubectl explain pod.spec.securityContext
kubectl explain pod.spec.containers.securityContext
```

Example (seccomp / security context)

```
A. seccomp - profile
https://github.com/docker/docker/blob/master/profiles/seccomp/default.json
```

```
apiVersion: v1
kind: Pod
metadata:
  name: audit-pod
  labels:
    app: audit-pod
spec:
  securityContext:
    seccompProfile:
      type: Localhost
      localhostProfile: profiles/audit.json

  containers:

  - name: test-container
    image: hashicorp/http-echo:0.2.3
    args:
    - "-text=just made some syscalls!"
    securityContext:
      allowPrivilegeEscalation: false
```

SecurityContext (auf Pod Ebene)

```
kubectl explain pod.spec.containers.securityContext
```

NetworkPolicy

```
## Firewall Kubernetes
```

Kubernetes Security Admission Controller Example

Seit: 1.2.22 Pod Security Admission

- 1.2.22 - ALpha - D.h. ist noch nicht aktiviert und muss als Feature Gate aktiviert (Kind)
- 1.2.23 - Beta -> d.h. aktiviert

Vorgefertigte Regelwerke

- privileges - keinerlei Einschränkungen
- baseline - einige Einschränkungen
- restricted - sehr streng

Praktisches Beispiel für Version ab 1.2.23 - Problemstellung

```
mkdir -p manifests
cd manifests
mkdir psa
cd psa
nano 01-ns.yml
```

```
## Schritt 1: Namespace anlegen
## vi 01-ns.yml

apiVersion: v1
kind: Namespace
metadata:
  name: test-ns1
  labels:
    pod-security.kubernetes.io/enforce: baseline
```

```
    pod-security.kubernetes.io/audit: restricted
    pod-security.kubernetes.io/warn: restricted
```

```
kubectl apply -f 01-ns.yml
```

```
## Schritt 2: Testen mit nginx - pod
## vi 02-nginx.yml

apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: test-ns1
spec:
  containers:
    - image: nginx
      name: nginx
      ports:
        - containerPort: 80
```

```
## a lot of warnings will come up
kubectl apply -f 02-nginx.yml
```

```
## Schritt 3:
## Anpassen der Sicherheitseinstellung (Phase1) im Container

## vi 02-nginx.yml

apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: test-ns1
spec:
  containers:
    - image: nginx
      name: nginx
      ports:
        - containerPort: 80
      securityContext:
        seccompProfile:
          type: RuntimeDefault
```

```
kubectl delete -f 02-nginx.yml
kubectl apply -f 02-nginx.yml
kubectl -n test-ns1 get pods
```

```
## Schritt 4:
## Weitere Anpassung runAsNotRoot
## vi 02-nginx.yml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: test-ns<tln>
spec:
  containers:
    - image: nginx
      name: nginx
      ports:
        - containerPort: 80
      securityContext:
        seccompProfile:
          type: RuntimeDefault
        runAsNonRoot: true
```

```
## pod kann erstellt werden, wird aber nicht gestartet
kubectl delete -f 02-nginx.yml
kubectl apply -f 02-nginx.yml
kubectl -n test-ns1 get pods
kubectl -n test-ns1 describe pods nginx
```

Praktisches Beispiel für Version ab 1.2.23 -Lösung - Container als NICHT-Root laufen lassen

- Wir müssen ein image, dass auch als NICHT-Root laufen kann
- .. oder selbst eines bauen (;o)) o bei nginx ist das bitnami/nginx

```
## vi 03-nginx-bitnami.yml
apiVersion: v1
kind: Pod
metadata:
  name: bitnami-nginx
  namespace: test-ns1
spec:
  containers:
    - image: bitnami/nginx
      name: bitnami-nginx
      ports:
        - containerPort: 80
      securityContext:
        seccompProfile:
          type: RuntimeDefault
        runAsNonRoot: true
```

```
## und er läuft als nicht root
kubectl apply -f 03_pod-bitnami.yml
kubectl -n test-ns1 get pods
```

Was muss ich bei der Netzwerk-Sicherheit beachten ?

Bereich 1: Kubernetes (Cluster)

```
1. Welche Ports sollten wirklich geöffnet sein ?

für Kubernetes

2. Wer muss den von wo den Kube-Api-Server zugreifen

- den Traffic einschränken
```

Bereich 2: Nodes

```
Alle nicht benötigten fremden Ports sollten geschlossen sein
Wenn offen, nur über vordefinierte Zugangswege (und auch nur bestimmte Nutzer)
```

Pods (Container / Image)

```
## Ingress (NetworkPolicy) - engmaschig stricken
## 1. Wer soll von wo auf welche Pod zugreifen können

## 2. Welche Pod auf welchen anderen Pod (Service)

ä Egress
## Welche Pods dürfen wohin nach draussen
```

Einschränking der Fähigkeien eines Pods

```
kein PrivilegeEscalation
nur notwendige Capabilities
unter einem nicht-root Benutzer laufen lassen
...

### Patching
```

pods -> neuestes images bei security vulnerablities

nodes -> auch neues patches (apt upgrade)

kubernetes cluster -> auf dem neuesten Stand

-> wie ist der Prozess ClusterUpdate, update der manifeste zu neuen API-Versionen

```

### RBAC
```

Nutzer (kubectl, systemnutzer -> pods)

1. Zugriff von den pods

2. Zugriff über helm / kubectl

Wer darf was ? Was muss der Nutzer können

```
### Compliance
```

PSP's / PSA PodSecurityPolicy was deprecated in Kubernetes v1.21, and removed from Kubernetes in v1.25

PSA - Pode Security Admission

```
## Kubernetes Probes (Liveness and Readiness)

### Übung Liveness-Probe



### Übung 1: Liveness (command)
```

What does it do ?

- At the beginning pod is ready (first 30 seconds)
- Check will be done after 5 seconds of pod being startet
- Check will be done periodically every 5 minutes and will check
  - for /tmp/healthy
  - if file is there will return: 0
  - if file is not there will return: 1
- After 30 seconds container will be killed
- After 35 seconds container will be restarted

```
```

cd

mkdir -p manifests/probes

cd manifests/probes

vi 01-pod-liveness-command.yml

apiVersion: v1 kind: Pod metadata: labels: test: liveness name: liveness-exec spec: containers:

- name: liveness image: busybox args:
  - /bin/sh
  - -c
  - touch /tmp/healthy; sleep 30; rm -f /tmp/healthy; sleep 600 livenessProbe: exec: command:
    - cat
    - /tmp/healthy initialDelaySeconds: 5 periodSeconds: 5

```
```

apply and test

kubectl apply -f 01-pod-liveness-command.yml kubectl describe -l test=liveness pods sleep 30 kubectl describe -l test=liveness pods sleep 5 kubectl describe -l test=liveness pods

```
```

cleanup

kubectl delete -f 01-pod-liveness-command.yml

```
### Übung 2: Liveness Probe (HTTP)
```

Step 0: Understanding Prerequisite:

This is how this image works:

after 10 seconds it returns code 500

http.HandleFunc("/healthz", func(w http.ResponseWriter, r *http.Request) { duration := time.Now().Sub(started) if duration.Seconds() > 10 { w.WriteHeader(500) w.Write([]byte(fmt.Sprintf("error: %v", duration.Seconds()))) } else { w.WriteHeader(200) w.Write([]byte("ok")) } })

```
```

Step 1: Pod - manifest

vi 02-pod-liveness-http.yml

status-code >=200 and < 400 o.k.

else failure

apiVersion: v1 kind: Pod metadata: labels: test: liveness name: liveness-http spec: containers:

- name: liveness image: k8s.gcr.io/liveness args:
  - /server livenessProbe: httpGet: path: /healthz port: 8080 httpHeaders:
    - name: Custom-Header value: Awesome initialDelaySeconds: 3 periodSeconds: 3

```
```

Step 2: apply and test

kubectl apply -f 02-pod-liveness-http.yml

after 10 seconds port should have been started

sleep 10 kubectl describe pod liveness-http

```


### Reference:

   * https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/

### Funktionsweise Readiness-Probe vs. Liveness-Probe


### Why / Howto /

  * Readiness checks, if container is ready and if it's not READY
    * SENDS NO TRAFFIC to the container


### Difference to LiveNess

  * They are configured exactly the same, but use another keyword
    * readinessProbe instead of livenessProbe

### Example
```

readinessProbe: exec: command: - cat - /tmp/healthy initialDelaySeconds: 5 periodSeconds: 5

```

### Reference

  * https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/#define-readiness-probes

## Kubernetes - Documentation

### LDAP-Anbindung

  * https://github.com/apprenda-kismatic/kubernetes-ldap

### Well-Known Annotations

  * https://kubernetes.io/docs/reference/labels-annotations-taints/

### Documentation zu microk8s plugins/addons

  * https://microk8s.io/docs/addons

### Shared Volumes - Welche gibt es ?

  * https://kubernetes.io/docs/concepts/storage/volumes/
```

```
## Kubernetes - Documentation - Learn Kubernetes

### Helpful to learn - Kubernetes

  * https://kubernetes.io/docs/tasks/

### Environment to learn

  * https://killercoda.com/killer-shell-cks

### Environment to learn II

  * https://killercoda.com/

### Youtube Channel

  * https://www.youtube.com/watch?v=01qcYSck1c4
```