Docker & Kubernetes

You put the workshop shop API into containers and run it in Kubernetes: from image and Compose to rollout, Gateway API, and hardening.

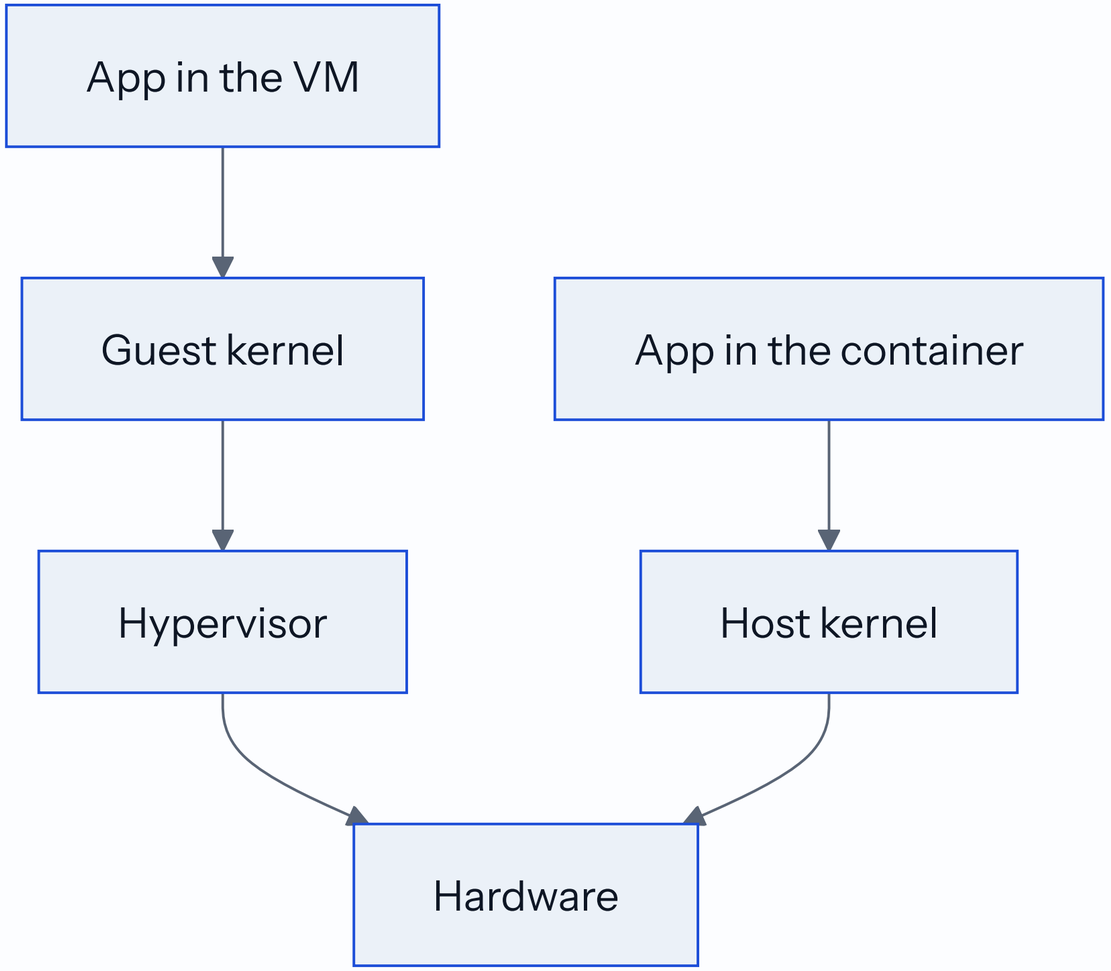
Mike Bild

Exercises: github.com/MikeBild/introduction-docker

Containers and images

A container is a process, not a VM · A good image is built in multiple stages · Data and networks live outside the container

What is inside a container?



Isolation comes from the kernel

- Namespaces separate processes, network, and file system
- cgroups limit memory and CPU
- The image only provides the file system

Four commands show it

```
docker run --rm node:26-slim node -e "console.log(process.pid)"
# 1
docker run --rm node:26-slim uname -r
# same kernel as on the host
docker run --rm --memory 256m --cpus 0.5 node:26-slim \
  cat /sys/fs/cgroup/memory.max /sys/fs/cgroup/cpu.max
# 268435456  /  50000 100000
docker run --rm node:26-slim id
# uid=0(root)
```

When a VM is the better choice

- Third-party code you do not trust
- A different operating system or a custom kernel
- Strict tenant isolation on shared hardware

An image consists of layers

- Each instruction in the Dockerfile creates a layer
- Docker rebuilds only from the first changed layer
- Put what rarely changes at the top

Two stages, one small image

```
FROM node:26-slim AS deps
WORKDIR /app
COPY package.json package-lock.json ./
RUN --mount=type=cache,target=/root/.npm npm ci --omit=dev

FROM node:26-slim
ENV NODE_ENV=production
WORKDIR /app
COPY --from=deps /app/node_modules ./node_modules
COPY package.json ./
COPY src ./src
USER node
CMD ["node", "src/main.ts"]
```

The build context is not a trash can

```
node_modules
test
.git
.env*
*.md
compose*.yaml
```

Choose the base image deliberately

- node:26-slim for everyday use, Debian with glibc
- node:26-alpine is smaller, but uses musl
- Distroless for production without a shell

What makes an image slow or unsafe

- COPY . . before npm ci makes the cache worthless
- npm start as CMD swallows SIGTERM
- Root in the container is root on the kernel

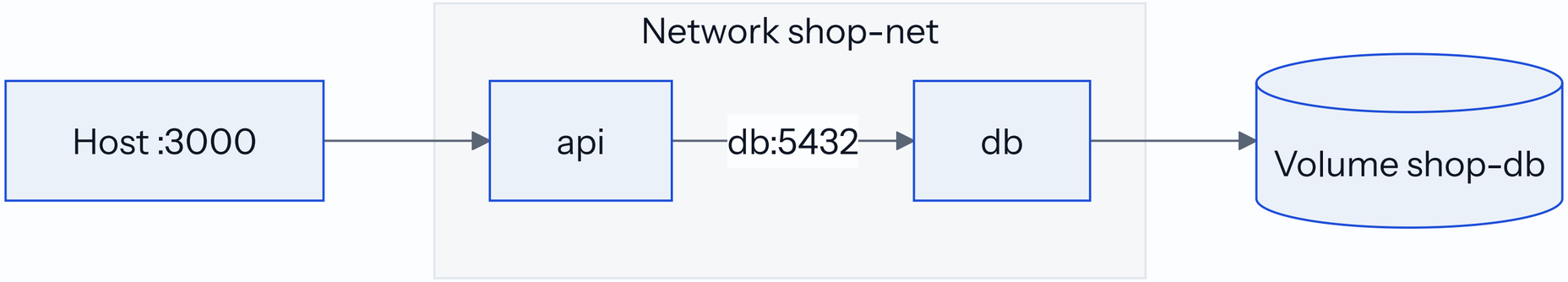
The container is disposable

- The container file system dies with it
- Docker manages a named volume separately
- A bind mount points to a host folder

Postgres with a named volume

```
docker volume create shop-db
docker run -d --name db --network shop-net \
  -v shop-db:/var/lib/postgresql \
  -v "$PWD/db/init.sql:/docker-entrypoint-initdb.d/init.sql:ro" \
  -e POSTGRES_PASSWORD=werkstatt -e POSTGRES_DB=shop \
  postgres:18
```

Containers find each other by name



Small mistakes lose data or open the database

- Volume at the wrong path: data still lands in the container
- Without a dedicated network, there is no name resolution
- -p 5432:5432 exposes the database to the whole network

Compose, health checks, logs, registry

Compose describes the whole stack · Started does not mean ready · Logs belong on stdout, not in files · An image is not ready until it is in the registry

From command to target state

- docker run describes steps
- compose.yaml describes the result
- One command starts everything: docker compose up

The workshop shop as a stack

```yaml
services:
  api:
    build: .
    image: shop-api:1.0.0
    environment:
      DATABASE_URL: postgres://postgres:werkstatt@db:5432/shop
    ports: ["3000:3000"]
    depends_on: [db]
  db:
    image: postgres:18
    environment: { POSTGRES_PASSWORD: werkstatt, POSTGRES_DB: shop }
    volumes:
      - db-data:/var/lib/postgresql
      - ./db/init.sql:/docker-entrypoint-initdb.d/init.sql:ro
volumes:
  db-data:
```

Everyday work with Compose

```
docker compose up -d --build
docker compose ps
docker compose logs -f api
docker compose exec db psql -U postgres -d shop
docker compose down
```

What Compose does not guarantee

- short-form depends_on only orders startup
- The API can be ready before Postgres
- version: in the header is obsolete and ignored

Running, but not working

- running only means: The process exists
- healthy means: The service answers requests
- Docker knows the difference only with a health check

Health checks for Postgres and API

```
db:
  healthcheck:
    test: ["CMD", "pg_isready", "-U", "postgres", "-d", "shop"]
    interval: 5s
    retries: 10
api:
  healthcheck:
    test:
      - CMD
      - node
      - -e
      - fetch('http://localhost:3000/readyz').then((r) => process.exit(r.ok ? 0 : 1))
    interval: 10s
    start_period: 10s
```

Wait for healthy

```
docker compose up -d --wait
docker compose ps    # api (healthy), db (healthy)
```

What a health check does not do

- It protects startup, not runtime
- Docker does not restart unhealthy containers
- Expensive checks put load on the service itself

The app logs, the platform forwards

```
export const log = (level: Level, msg: string, fields: Record<string, unknown> = {}) =>
  console.log(JSON.stringify({ time: new Date().toISOString(), level, msg, ...fields }))

log('info', 'order placed', { orderId, totalCents })
```

All services in one place

```
docker compose logs -f
docker compose logs -t --since 10m api
docker compose logs --no-log-prefix api | jq -c 'select(.msg == "order placed")'
```

Rotation instead of a full disk

```yaml
x-logging: &logging
  driver: local
  options:
    max-size: "10m"
    max-file: "3"
services:
  api:
    logging: *logging
  db:
    logging: *logging
```

When central logs are needed

- Multiple hosts or a cluster
- Search across days and weeks
- Alerts on error rates

The name is the address

- Full name: host:port/name:tag
- Without a host, Docker uses docker.io
- The digest sha256:... names the content

Your own registry in one line

```
docker run -d --name registry --restart=always \
  -p 127.0.0.1:5001:5000 \
  -v registry-data:/var/lib/registry \
  registry:3
```

Tag, push, pull

```
docker tag shop-api:1.0.0 localhost:5001/shop-api:1.0.0
docker push localhost:5001/shop-api:1.0.0
curl http://localhost:5001/v2/shop-api/tags/list
docker image rm localhost:5001/shop-api:1.0.0
docker pull localhost:5001/shop-api:1.0.0
```

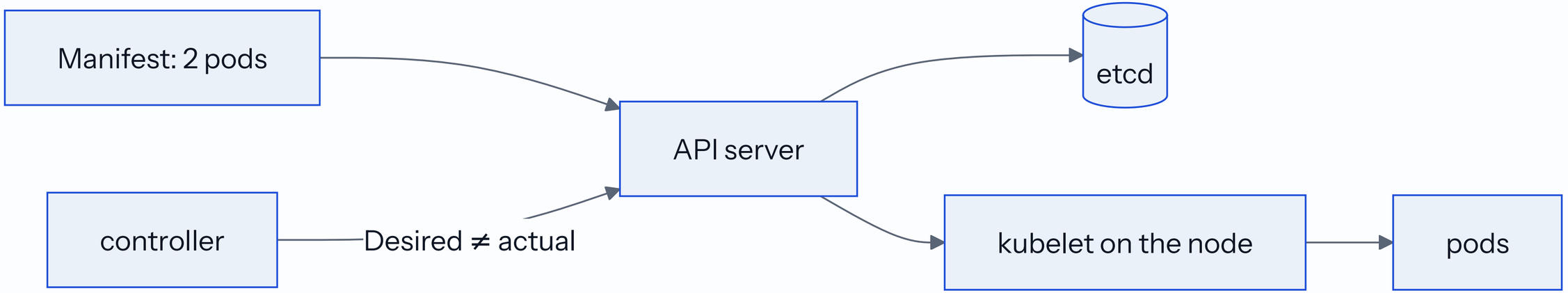
Operating a registry

- Outside localhost, only with TLS and login
- Old layers disappear only through garbage collection
- Hosted registries take care of operations for you

Kubernetes basics

Kubernetes constantly reconciles desired and actual state · You start Deployments, not Pods · A Service gives Pods a stable name · Configuration does not belong in the image

Desired state instead of command sequence



Control plane and nodes

- Control plane: API server, scheduler, controllers, etcd
- Nodes: kubelet and container runtime
- kubectl talks only to the API server

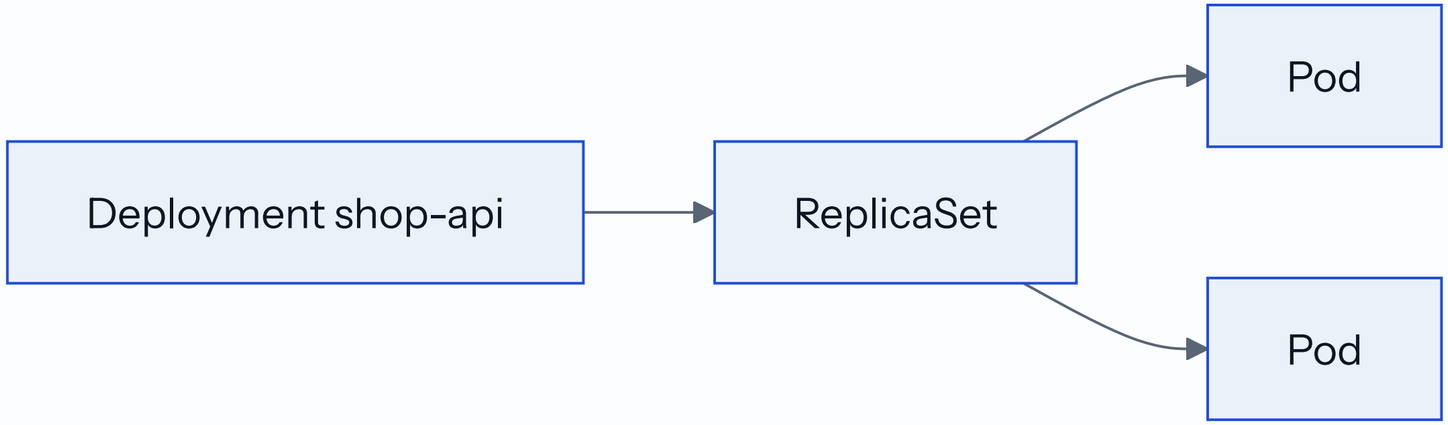
A cluster on your laptop

```
kind create cluster --name werkstatt --config kind-config.yaml
kind load docker-image shop-api:1.0.0 --name werkstatt
kubectl get nodes
```

Explore objects instead of guessing

```
kubectl get pods -A
kubectl api-resources
kubectl explain deployment.spec.strategy
kubectl get deployment coredns -n kube-system -o yaml
```

Pod, ReplicaSet, Deployment



The API Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: shop-api
  namespace: werkstatt
spec:
  replicas: 2
  selector:
    matchLabels: { app.kubernetes.io/name: shop-api }
  template:
    metadata:
      labels: { app.kubernetes.io/name: shop-api }
    spec:
      containers:
        - name: api
          image: shop-api:1.0.0
          imagePullPolicy: IfNotPresent
          ports: [{ name: http, containerPort: 3000 }]
```

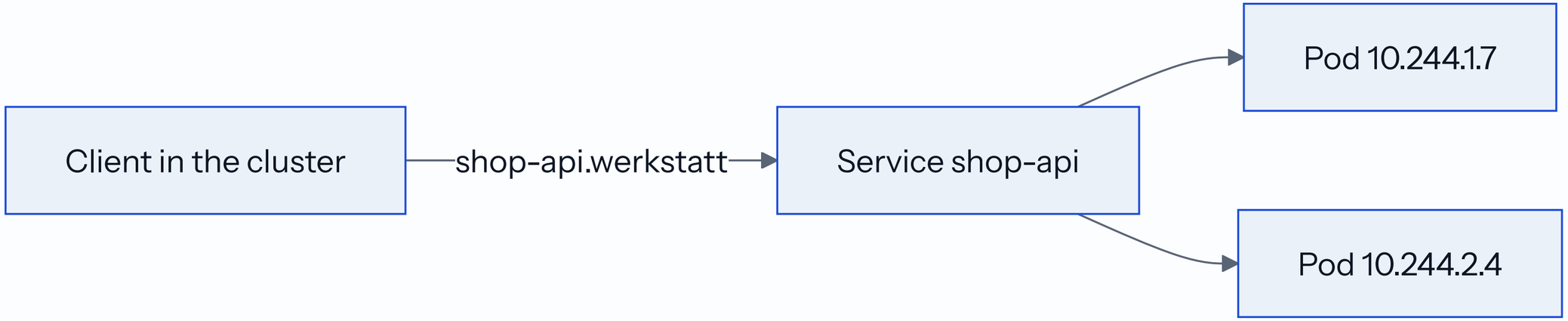
Apply and observe

```
kubectl apply -f .
kubectl get pods -n werkstatt -o wide
kubectl delete pod -n werkstatt -l app.kubernetes.io/name=shop-api --wait=false
kubectl get pods -n werkstatt -w
kubectl scale deployment shop-api -n werkstatt --replicas=3
```

Why does the Pod not start?

- ImagePullBackOff: Image is missing in the cluster
- CrashLoopBackOff: The process keeps exiting
- Pending: No Node has enough space

Why Pods need a Service



The API Service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: shop-api
  namespace: werkstatt
spec:
  selector: { app.kubernetes.io/name: shop-api }
  ports:
    - name: http
      port: 80
      targetPort: http
```

Test inside and outside

```
kubectl run curl -n werkstatt --rm -it --restart=Never \
  --image=curlimages/curl -- curl -s http://shop-api/products
kubectl get endpointslices -n werkstatt -l kubernetes.io/service-name=shop-api
kubectl port-forward -n werkstatt service/shop-api 8080:80
curl localhost:8080/products
```

Types of Services

- ClusterIP: only in the cluster, the default
- NodePort: one port on every Node
- LoadBalancer: an external IP from the cloud provider

One image, many environments

- The image is the same in test and production
- ConfigMap carries settings like port and database host
- Secret carries passwords and tokens

Create ConfigMap and Secret

```
kubectl create secret generic shop-db -n werkstatt \
  --from-literal=POSTGRES_PASSWORD=werkstatt
```

Use them in the Deployment

```
containers:
  - name: api
    envFrom:
      - configMapRef: { name: shop-api-config }
    env:
      - name: DB_PASSWORD
        valueFrom:
          secretKeyRef: { name: shop-db, key: POSTGRES_PASSWORD }
      - name: DATABASE_URL
        value: postgres://$(DB_USER):$(DB_PASSWORD)@$(DB_HOST):5432/$(DB_NAME)
```

What a Secret cannot do

- Base64 is encoding, not encryption
- RBAC controls who may read Secrets
- Changes do not reach running Pods

Running in the cluster

Probes tell Kubernetes when a pod is working · Plan requests, cap limits · An update without downtime is a rollout · Gateway API replaces Ingress

Three probes, three outcomes

- Startup: until it succeeds, the others wait
- Liveness: if it fails, the kubelet restarts it
- Readiness: if it fails, the pod leaves the service

Probes for the API

```
containers:
  - name: api
    startupProbe:
      httpGet: { path: /livez, port: http }
      periodSeconds: 2
      failureThreshold: 15
    livenessProbe:
      httpGet: { path: /livez, port: http }
      periodSeconds: 10
    readinessProbe:
      httpGet: { path: /readyz, port: http }
      periodSeconds: 5
      failureThreshold: 2
```

When the database is missing

```
kubectl scale statefulset postgres -n werkstatt --replicas=0
kubectl get pods -n werkstatt -l app.kubernetes.io/name=shop-api -w
# READY 0/1, RESTARTS 0
kubectl scale statefulset postgres -n werkstatt --replicas=1
```

Wrong probes kill healthy pods

- Dependencies in the liveness probe trigger restart cascades
- Timeouts that are too tight kill pods under load
- Without a startup probe, a slow start dies in a loop

Requests and limits

- Requests: what the scheduler reserves on the node
- Limits: the most the container can get
- CPU is throttled, memory terminates the process

API resources

```
containers:
  - name: api
    resources:
      requests:
        cpu: 100m
        memory: 512Mi
      limits:
        memory: 512Mi
```

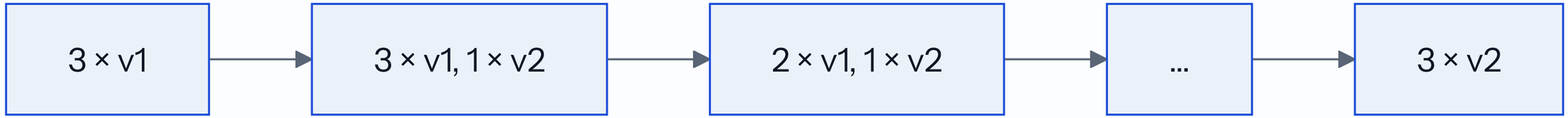
Node.js knows its limit

```
kubectl exec -n werkstatt deploy/shop-api -- /nodejs/bin/node -e \
  "console.log(require('v8').getHeapStatistics().heap_size_limit / 2 ** 20)"
# 268 with 512Mi limit, 262 with 256Mi limit
```

Misconfigured resources hurt under load

- Without requests, the scheduler packs nodes too full
- A heap above the limit leads to OOMKilled without a log
- Too little CPU slows probes and makes pods unready

Swap Pods step by step



The strategy in the Deployment

```yaml
spec:
  replicas: 3
  minReadySeconds: 5
  strategy:
    type: RollingUpdate
    rollingUpdate: { maxSurge: 1, maxUnavailable: 0 }
  template:
    spec:
      terminationGracePeriodSeconds: 30
      containers:
        - name: api
          lifecycle:
            preStop:
              sleep: { seconds: 5 }
```

Roll out and roll back

```
kubectl set image deployment/shop-api api=shop-api:1.1.0 -n werkstatt
kubectl rollout status deployment/shop-api -n werkstatt
kubectl rollout history deployment/shop-api -n werkstatt
kubectl rollout undo deployment/shop-api -n werkstatt
```

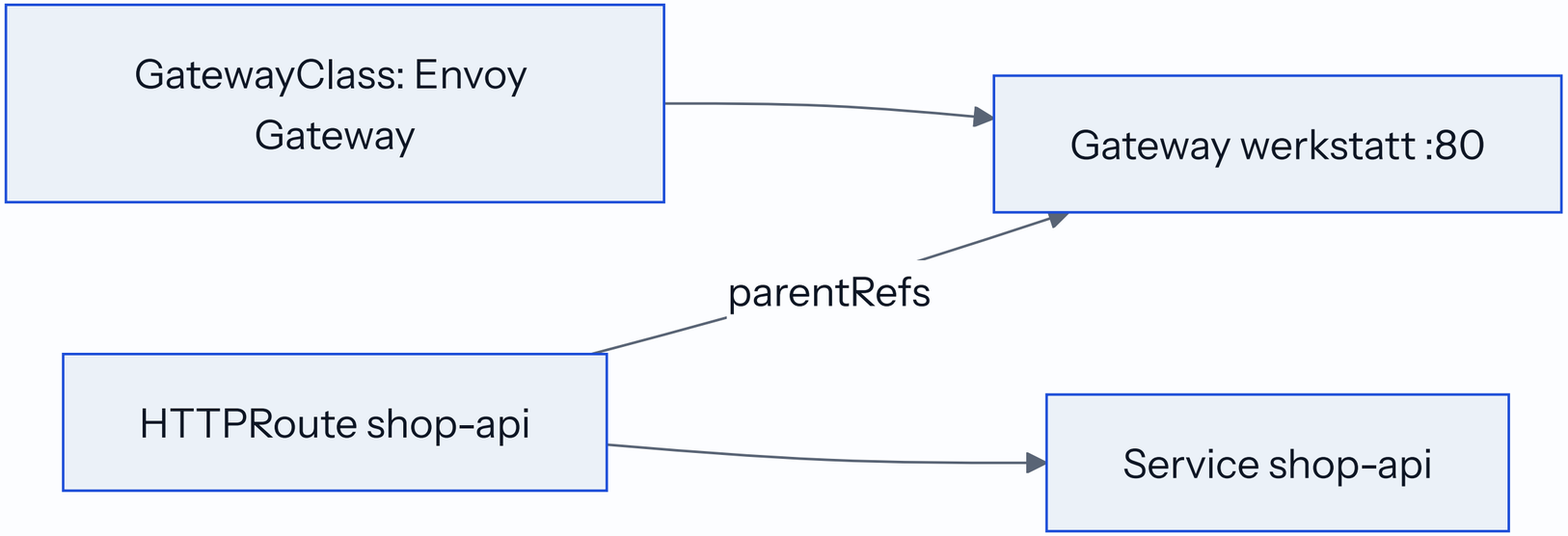
When a rollout hangs

- The new version never becomes ready: the rollout stops
- No preStop: requests land on terminated Pods
- Database migrations must work with both versions

From Ingress to Gateway API

- Ingress is frozen; new features are only in Gateway API
- ingress-nginx has not been maintained since March 2026
- Gateway API is a standard with many implementations

Three objects, three roles



One entry point for the namespace

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata: { name: werkstatt, namespace: werkstatt }
spec:
  gatewayClassName: envoy
  listeners:
    - name: http
      protocol: HTTP
      port: 80
      allowedRoutes: { namespaces: { from: Same } }
```

The API route

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata: { name: shop-api, namespace: werkstatt }
spec:
  parentRefs: [{ name: werkstatt }]
  rules:
    - matches: [{ path: { type: PathPrefix, value: /api } }]
      filters:
        - type: URLRewrite
          urlRewrite:
            path: { type: ReplacePrefixMatch, replacePrefixMatch: / }
      backendRefs: [{ name: shop-api, port: 80 }]
```

Try it locally

```
kubectl apply --server-side -f \
  https://github.com/envoyproxy/gateway/releases/download/v1.9.3/install.yaml
kubectl apply -f .
kubectl get gateway,httproute -n werkstatt
kubectl get service -n envoy-gateway-system \
  -l gateway.envoyproxy.io/owning-gateway-name=werkstatt
kubectl port-forward -n envoy-gateway-system service/<name> 8080:80
curl localhost:8080/api/products
```

Security

A small image has fewer vulnerabilities

Every package is attack surface

- Shell and package managers also help attackers
- Every library brings its own CVEs
- What is not in the image does not need patching

Scanning shows the difference

```
trivy image --severity HIGH,CRITICAL node:26-slim
#  HIGH: 49 (measurement from 10/29/2026)
trivy image --severity HIGH,CRITICAL shop-api:1.1.0
#  HIGH: 0  (Distroless)
trivy image --exit-code 1 --severity CRITICAL --ignore-unfixed shop-api:1.1.0
```

Distroless instead of slim

```
FROM node:26-slim AS deps
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm ci --omit=dev

FROM gcr.io/distroless/nodejs26-debian13:nonroot
ENV NODE_ENV=production
WORKDIR /app
COPY --from=deps /app/node_modules ./node_modules
COPY package.json ./
COPY src ./src
CMD ["src/main.ts"]
```

Remove container permissions

```
api:
  image: shop-api:1.1.0
  read_only: true
  cap_drop: [ALL]
  security_opt: ["no-new-privileges:true"]
  healthcheck:
    test:
      - CMD
      - /nodejs/bin/node
      - -e
      - fetch('http://localhost:3000/readyz').then((r) => process.exit(r.ok ? 0 : 1))
```

Questions?

Mike Bild · mike.bild@gmail.com · mikebild.dev

Workshops für Teams, die Architekturentscheidungen selbst treffen