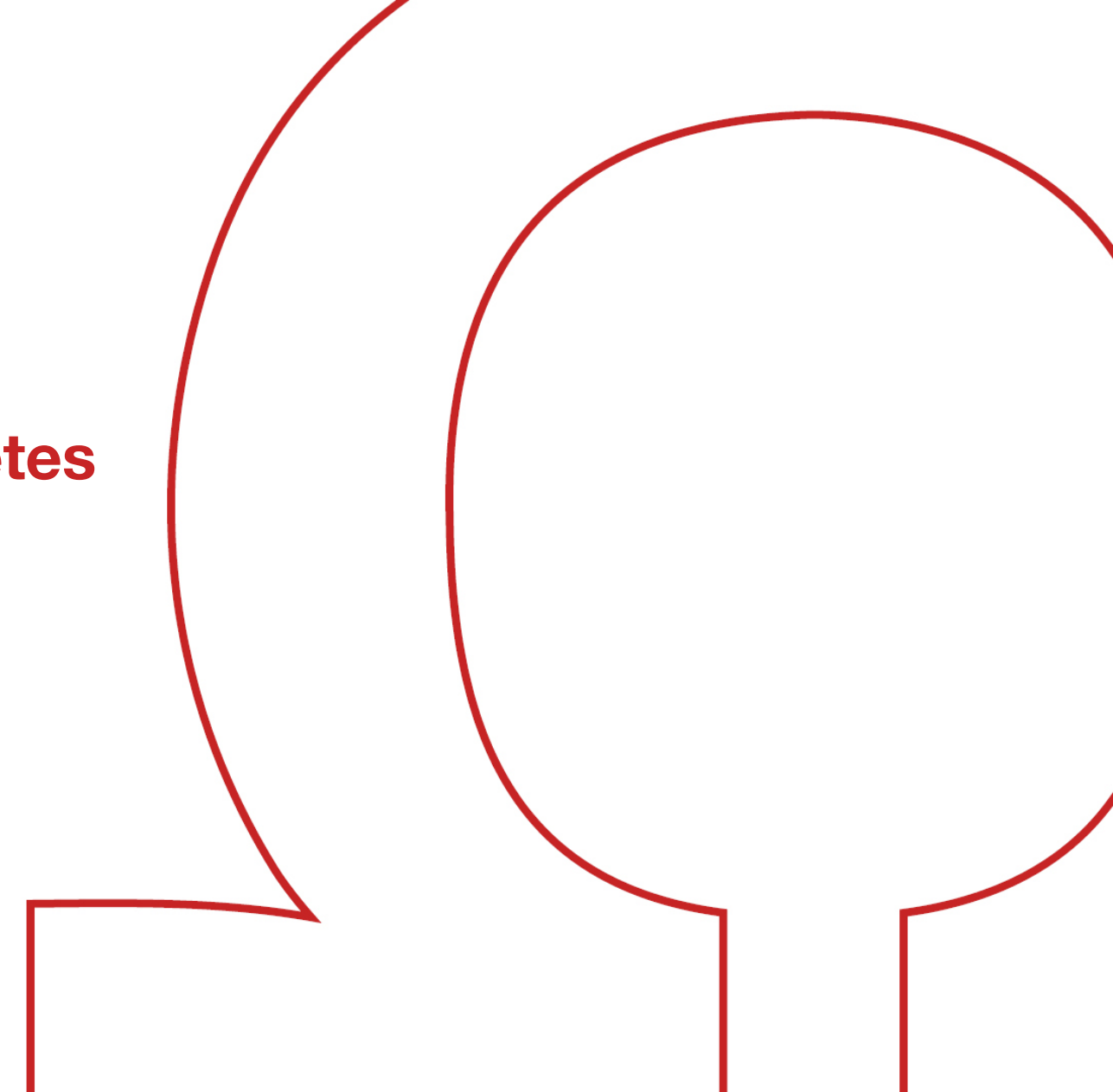
ohm Technische Hochschule Nürnberg

Introduction to Kubernetes

For Developers

Dr.-Ing. Georg Schwarz



About the lecturer

georg-schwarz.com

Dr.-Ing. Georg Schwarz

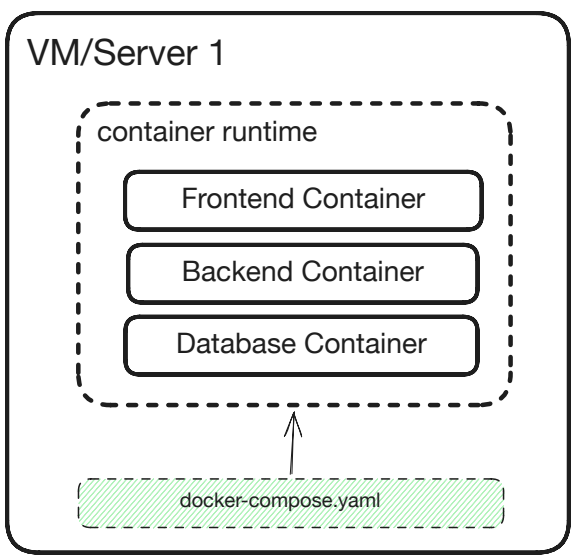
- Founder of diffcraft.io
- Certified kubernetes administrator (CKA)
- Full-stack software engineer
- PhD on microservices at FAU

Let’s connect on LinkedIn

Why Kubernetes?

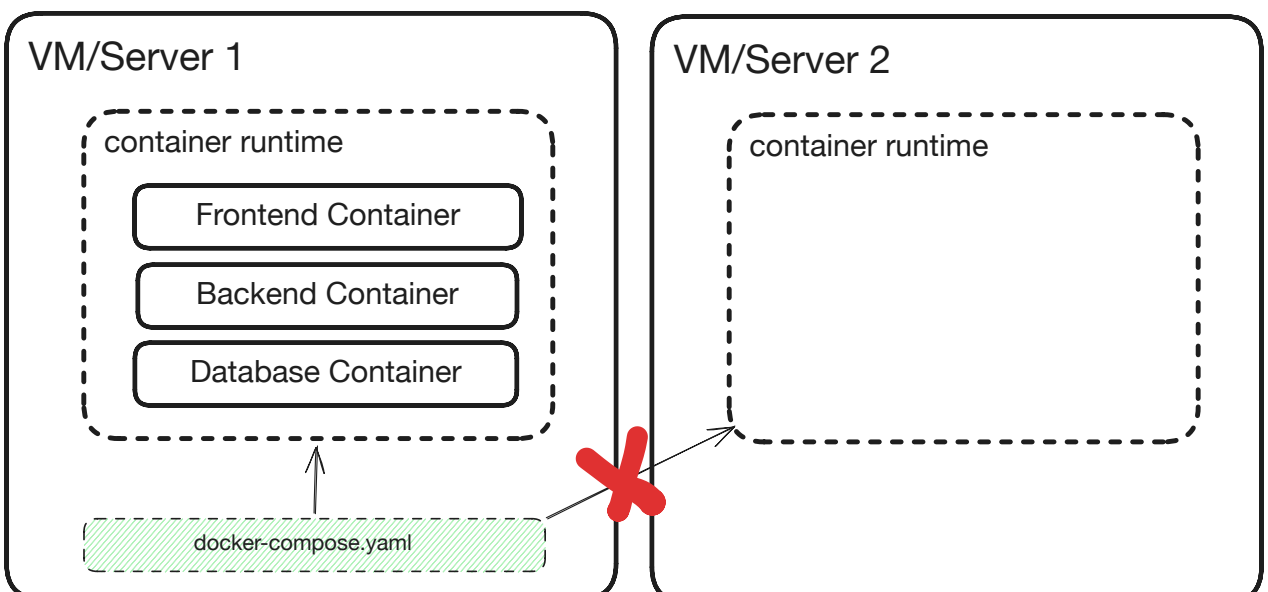
Where Docker Compose works well

- Local development.
- Small demos and integration tests.
- One command for a small stack: `docker compose up`



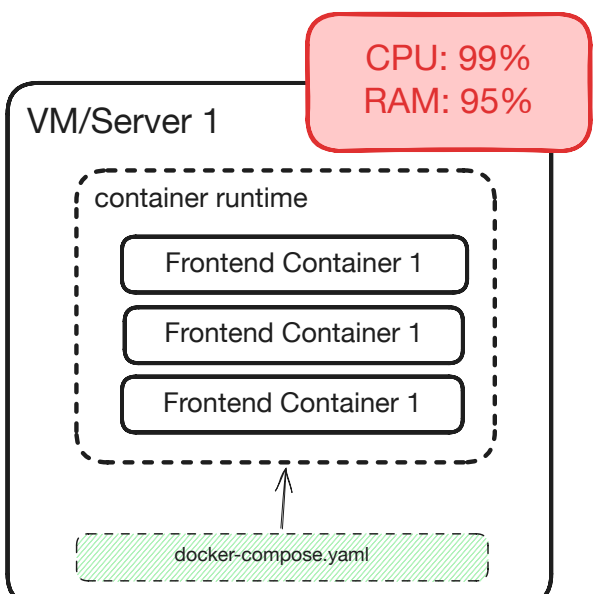
Docker Compose has a host boundary

- Compose schedules containers on one host.
- If that host fails, all containers on it fail.



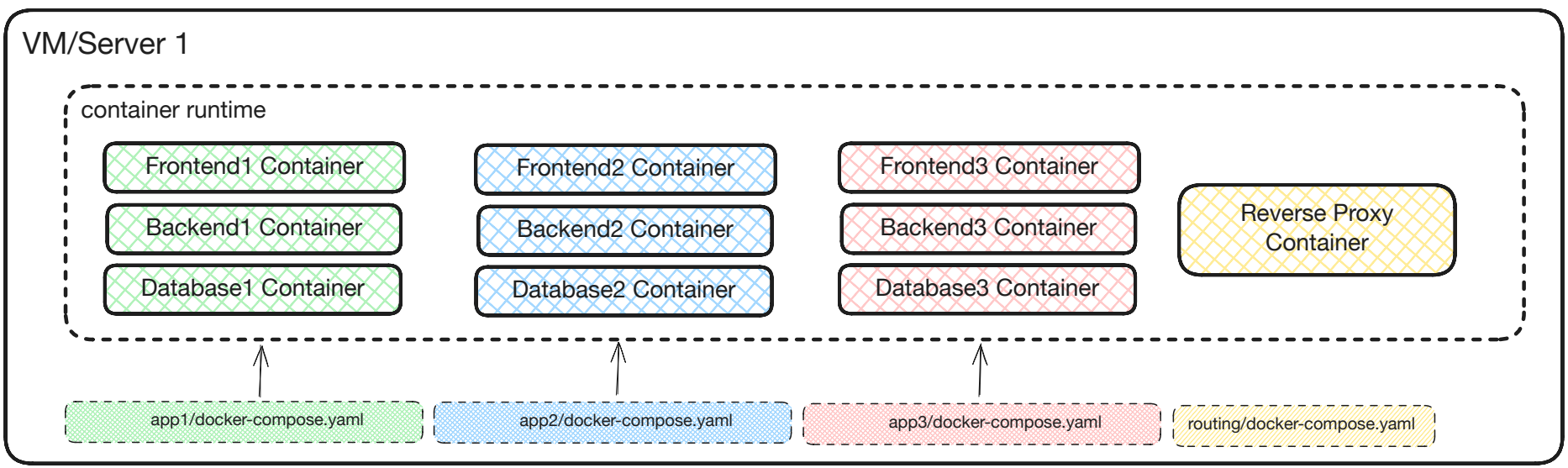
Scaling is limited by one machine

- You can start more replicas, but compete for the same CPU, memory, disk, and network.



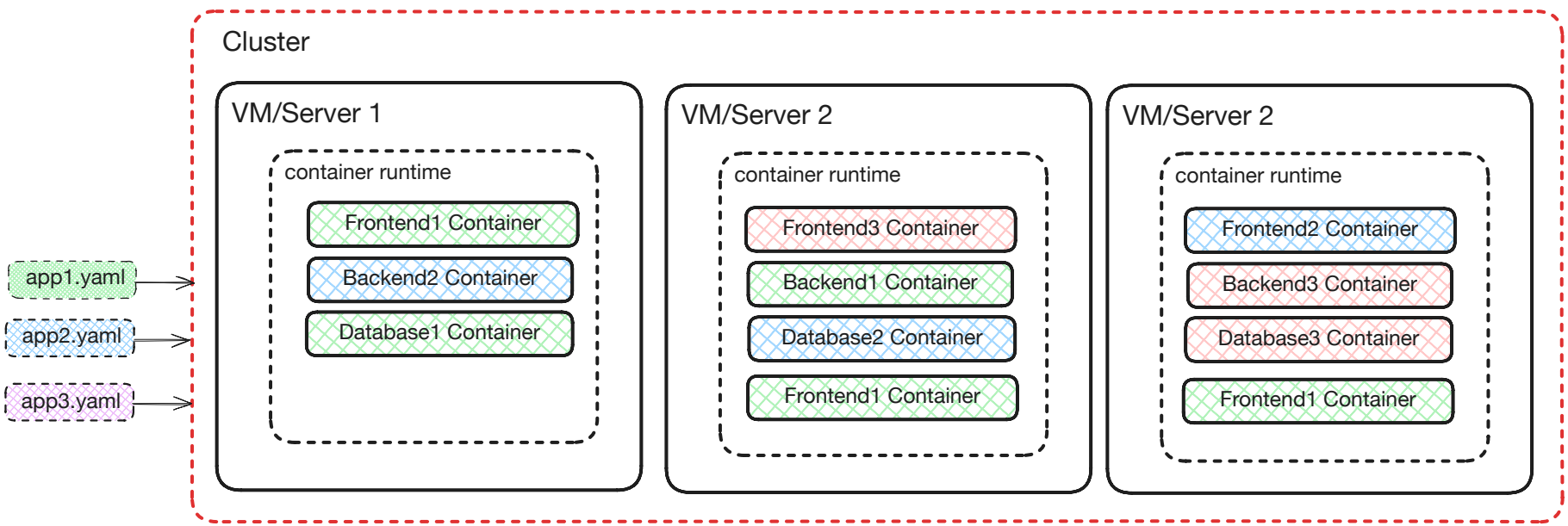
Many apps on one VM become messy

- Naming, ports, networks, and volumes must be coordinated manually.
- Isolation between teams/environments is weak.
- Operations become a collection of conventions and shell scripts.



Clusters: The missing abstraction

Kubernetes turns many machines into one programmable application platform.



What is Kubernetes?

Kubernetes Definition

> *"Kubernetes, also known as K8s, is an open source system for **automating deployment, scaling, and management of containerized applications**."*
>
> Source: https://kubernetes.io/

- Groups containers into logical units for easy management and discovery.
- Builds upon 15 years of experience of running production workloads at Google.
- Combined with best-of-breed ideas and practices from the community.

Kubernetes Claims

Planet scale: Scale without increasing your operations team.

Never outgrow: Deliver your applications consistently and easily no matter how complex your need is.

Run K8s anywhere: Move workloads to where it matters to you (on-premises, hybrid, or public cloud) - it's open source!

Source: https://kubernetes.io/

A Useful Mental Model

Kubernetes is the **operating system of the cloud**.

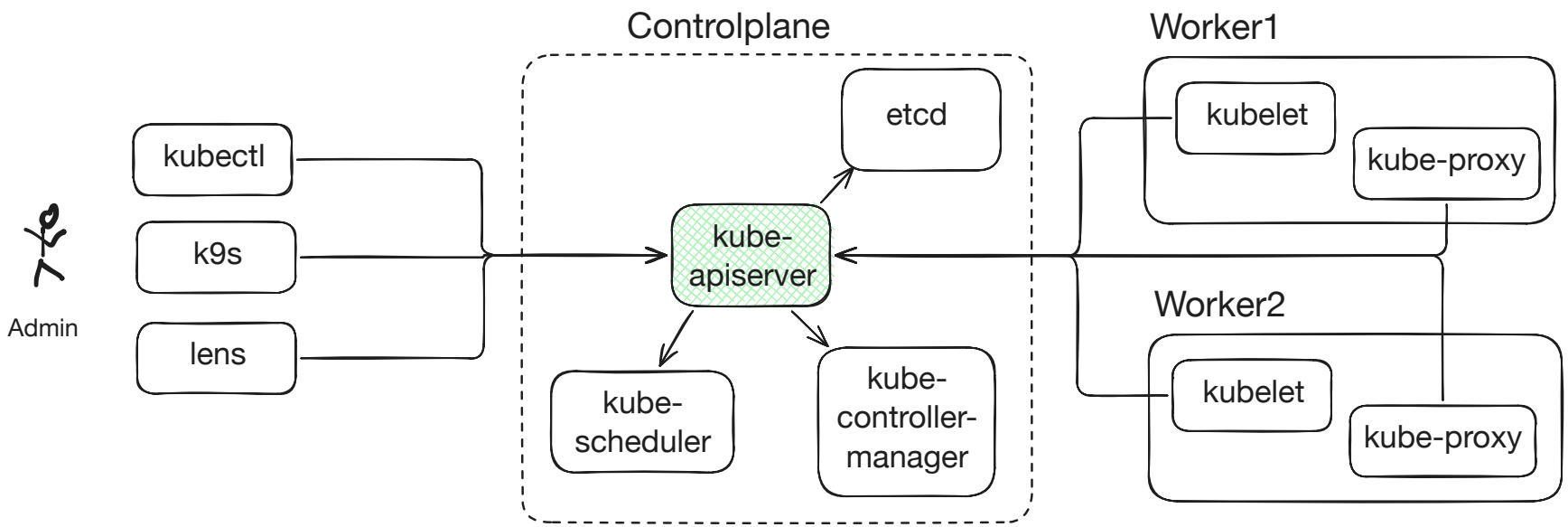
For developers, it is an **API for deployment**: you declare what should run, and Kubernetes reduces manual operational intervention.

It **schedules workloads** onto nodes with enough capacity and keeps applications running when nodes fail.

It is **portable and extensible**: you can run it in many environments and make it your own with custom resource types.

Kubernetes API

One API as entrypoint: the apiserver



- All cluster interaction goes through the API server.
- `kubectl`, dashboards, and control-plane components are API clients.

The Kubernetes HTTP API

- Resources are exposed through HTTP endpoints by the apiserver.
- Clients create, read, update, watch, and delete resources.

```http
GET <apiserver>/apis/apps/v1/namespaces/shop/deployments/catalog
Accept: application/json
```

- `apps/v1` is the **API group** and **version**.
- `shop` specifies the **namespace**.
- `deployments` is the **resource type**.
- `catalog` identifies the **concrete resource**.

Kubectl

- Standard CLI client for Kubernetes (uses the HTTP API).
- Config from `~/.kube/config` or from the location given in `KUBECONFIG`.

Desired state vs. actual state

- You declare what should exist.
- Kubernetes observes what exists.
- Controllers continuously move actual state toward desired state.

Pseudo code

```
while true:
    desired = read_from_api()
    observed = read_from_api()

    if observed != desired:
        write_correction_to_api()
```

Resources

- Kubernetes builds on resources (you can even define your own), see `kubectl api-resources`.
- In this lecture, we focus on a few of the standard resources.

Node

A machine in the cluster.

Namespace

A logical partition.

Pod

A running workload unit.

Deployment

Replicated Pods and updates.

ConfigMap

Non-sensitive configuration.

Service

Stable network endpoint.

Manifest YAML to describe state

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: catalog
  namespace: shop
spec:
  replicas: 3
  selector: ...
  template: ...
```

- `kubectl apply -f <manifest file>` to update the desired state.
- `kubectl delete -f <manifest file>` to delete some state.

Kubernetes YAML

`apiVersion`

Which API group and version defines this resource?

`kind`

Which type of resource is this?

`metadata`

Identity, labels, namespace, annotations.

`spec`

The desired state for this resource.

`metadata`

- `name` identifies the resource inside a namespace.
- `namespace` groups resources.
- `labels` connect resources to each other.

```yaml
...
metadata:
  name: catalog
  namespace: shop
  labels:
    app: catalog
    tier: backend
...
```

spec

- Describes the desired state.
- Exact fields depend on the resource `kind`.

```yaml
apiVersion: apps/v1
kind: Deployment
...
spec:
  replicas: 3
  template:
    spec:
      containers:
        - name: catalog
          image: ghcr.io/acme/catalog:1.4.2
```

Declarative vs. imperative

Imperative

Tell Kubernetes what to do now: create, scale, delete.

Declarative

Tell Kubernetes what should exist: apply resource manifests.

- Imperative commands are useful for exploration and quick experiments.
- Declarative manifests are repeatable, reviewable, and fit CI/CD.
- Production work should leave a readable desired state behind.

Cluster resources

Node

- A node is a worker machine in Kubernetes, can be VM or physical machine.
- **Worker node**: runs workloads, is managed by control plane
- **Control-plane**: orchestration layer

```
root@controlplane:~$ kubectl get nodes
NAME           STATUS   ROLES           AGE   VERSION
controlplane   Ready    control-plane   19d   v1.35.1
node01         Ready    <none>          19d   v1.35.1
```

Namespace

- Logical partition inside a cluster.
- Isolate teams, environments, and applications.
- Many resource names only need to be unique inside one namespace.

```
root@controlplane:~$ kubectl get namespaces
NAME                 STATUS   AGE
cilium-secrets       Active   19d
default              Active   19d
erp                  Active   20s
kube-node-lease      Active   19d
kube-public          Active   19d
kube-system          Active   19d
sales                Active   27s
```

Creating a namespace

- Declarative: `kubectl apply -f namespace.yaml`

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: shop
```

- Imperative: `kubectl create namespace shop`

Accessing namespaced resources

- Access resources of a namespace with `--namespace <namespace>` or `-n <namespace>`. Use flag `-A` to get from all namespaces.

```
root@controlplane:~$ kubectl get pods -n kube-system
NAME                                 READY   STATUS    RESTARTS      AGE
cilium-envoy-hrbhs                   1/1     Running   1 (22m ago)   19d
cilium-envoy-hwqjz                   1/1     Running   2 (22m ago)   19d
cilium-operator-5d8ddcb8d8-8h2jp     1/1     Running   3 (22m ago)   19d
cilium-vtmg2                         1/1     Running   2 (22m ago)   19d
cilium-vzvsz                         1/1     Running   1 (22m ago)   19d
coredns-5f68d5bd7f-2nf9j             1/1     Running   1 (22m ago)   19d
coredns-5f68d5bd7f-7h5gs             1/1     Running   1 (22m ago)   19d
etcd-controlplane                    1/1     Running   2 (22m ago)   19d
kube-apiserver-controlplane          1/1     Running   2 (22m ago)   19d
kube-controller-manager-controlplane 1/1     Running   2 (22m ago)   19d
kube-scheduler-controlplane          1/1     Running   2 (22m ago)   19d
```

Workload resources

Pod: the smallest deployable unit

- A Pod contains one or more containers.
- Containers in one Pod are scheduled together.
- They share network namespace and can talk over `localhost`.

Example imperative:

```
kubectl run web --image=nginx:1.27 --port=80 -n shop
```

Pod example

Example declarative:

YAML

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: web
  namespace: shop
  labels:
    app: web
spec:
  containers:
    - name: nginx
      image: nginx:1.27
      ports:
        - containerPort: 80
```

Practice time!

Recap: Resources so far

Node

A machine in the cluster.

Namespace

A logical partition.

Pod

A running workload unit.

Deployment

Replicated Pods and updates.

ConfigMap

Non-sensitive configuration.

Service

Stable network endpoint.

Workload resources

Pods have no controller

- A Pod is bound to one node.
- If that node fails, the Pod disappears with it:

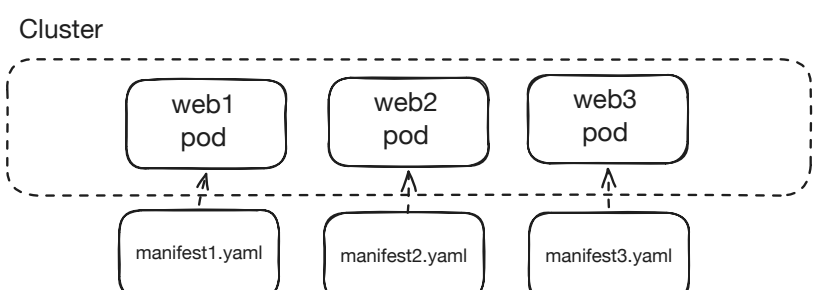
```
# Drain = clears workloads, emulates node failure
kubectl drain node01 --ignore-daemonsets --force
```

- A naked Pod does not express “please keep one running”.

Scaling Pods is bothersome

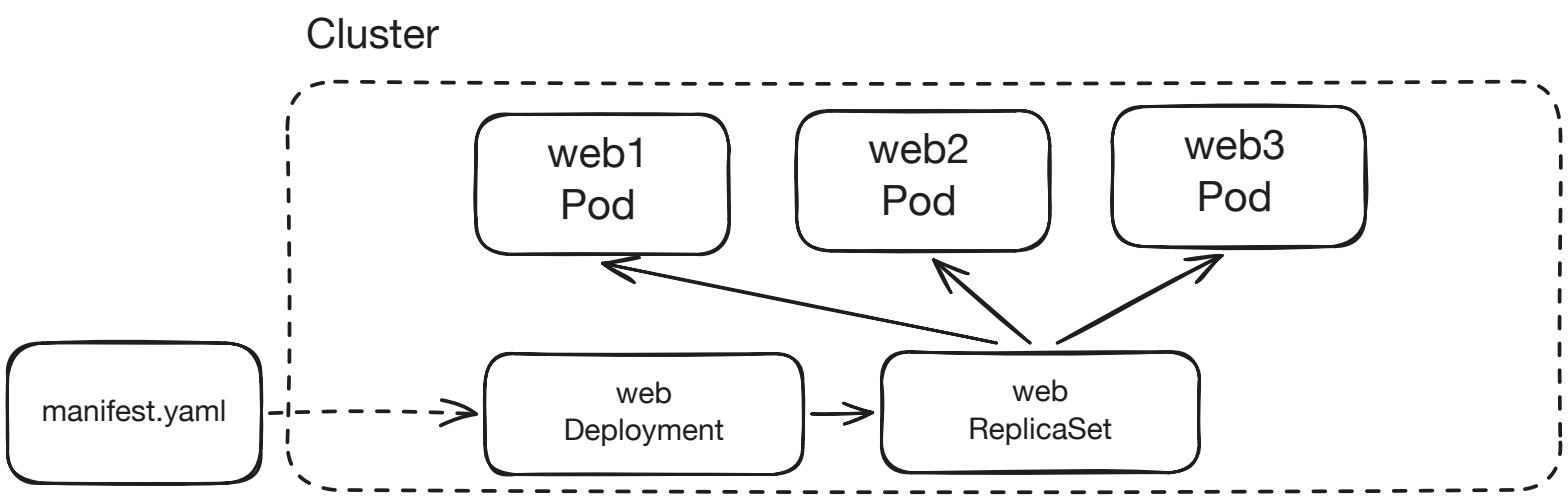
- Manual naming and lifecycle management does not scale well.
- Updating three Pods means touching three objects.

```
kubectl run web-1 --image=nginx:1.27 --port=80 -n shop
kubectl run web-2 --image=nginx:1.27 --port=80 -n shop
kubectl run web-3 --image=nginx:1.27 --port=80 -n shop
```



Deployment and ReplicaSet

- Deployment defines a Pod template and update strategy.
- ReplicaSet keeps the requested number of Pods running.
- If a Pod disappears, the ReplicaSet creates a replacement.



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
  namespace: shop
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: nginx
          image: nginx:1.30.2
          ports:
            - containerPort: 80
```

Deployment labels and selectors

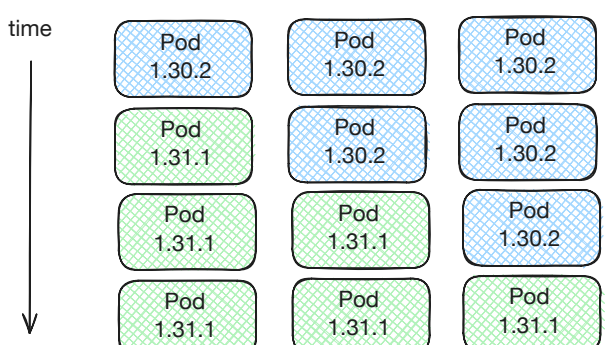
- `spec.selector` says which Pods belong to this Deployment.
- `spec.template.metadata.labels` creates labels on new Pods.
- These labels must match.

```yaml
...
spec:
  selector:
    matchLabels:
      app: catalog
  template:
    metadata:
      labels:
        app: catalog
    ...
```

Rolling update

- Kubernetes replaces Pods gradually instead of stopping all replicas at once.

```
kubectl set image deployment/web nginx=nginx:1.31.1
kubectl rollout status deployment/web
kubectl rollout undo deployment/web
```



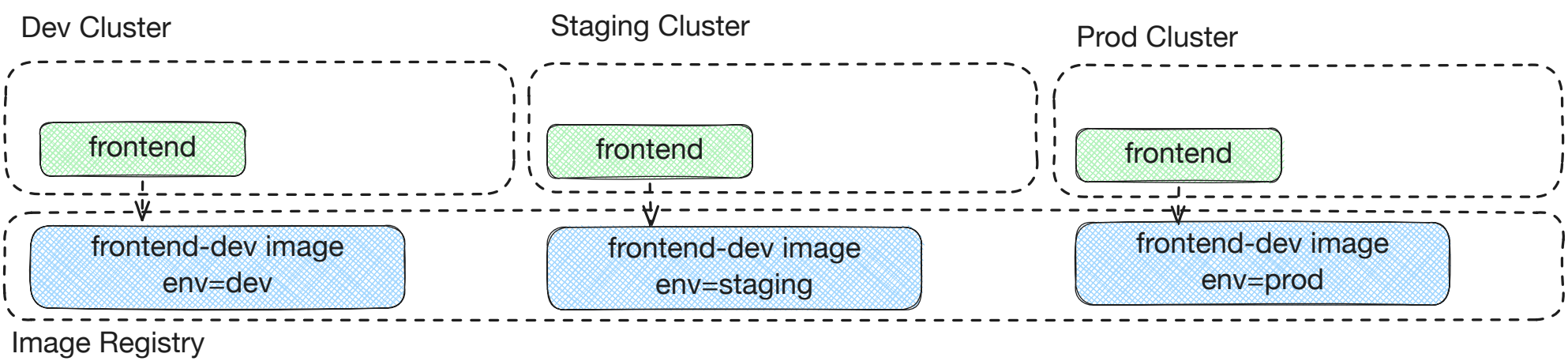
Useful Deployment commands

```
kubectl create deployment <name> --image <image>
kubectl get deployments
kubectl get replicasets
kubectl get pods --show-labels
kubectl get pods -l <label>
kubectl describe deployment <deployment>
kubectl scale deployment <deployment> --replicas=3
```

Configuration resources

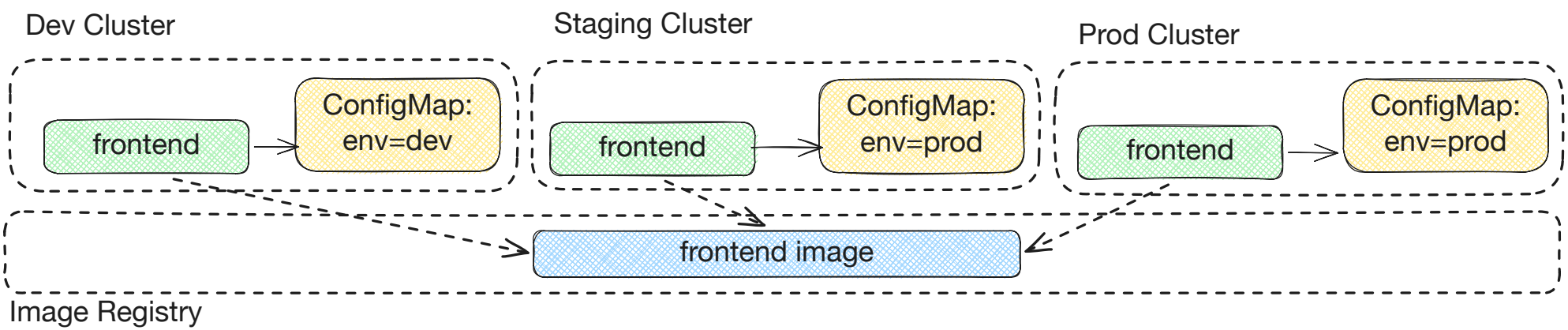
Configuration should not be baked into images

- Images should be reusable across environments.
- Runtime configuration belongs outside the image.
- Otherwise: image per environment, reconfiguration needs new image.



ConfigMap for non-sensitive configuration

- Stores plain configuration data.
- Good for feature flags, URLs, small config files.
- Not intended for passwords or tokens.



ConfigMap example

- Declarative YAML:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config
data:
  ENV: dev
```

- Imperative command:

```
kubectl create configmap my-config --from-literal=ENV=dev
```

How to use config in a Deployment

- In Pod definition, where containers are specified:

```yaml
...
  containers:
  - name: my-container
    ...
    envFrom:
      - configMapRef:
          name: my-config
...
```

Network resources

Pods are replaceable, IPs are not stable

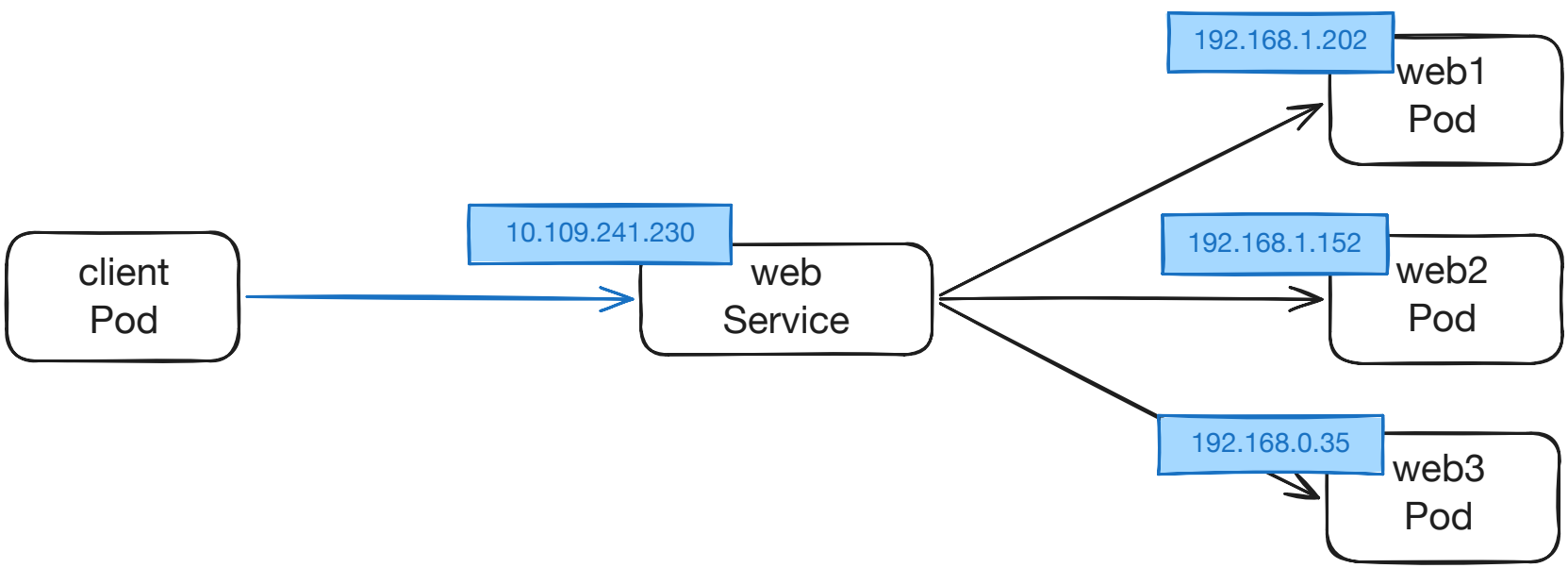
- Pods can be created, deleted, and moved.
- A Pod IP is not a good application endpoint.
- Clients need a stable name and address.

```
root@controlplane:~$ kubectl create deployment frontend --image nginx
root@controlplane:~$ kubectl get pods -o custom-columns=NAME:.metadata.name,IP:.status.podIP
NAME                        IP
frontend-9f6b85c8c-98q7t    192.168.1.136

root@controlplane:~$ kubectl delete pod frontend-9f6b85c8c-98q7t
root@controlplane:~$ kubectl get pods -o custom-columns=NAME:.metadata.name,IP:.status.podIP
NAME                        IP
frontend-9f6b85c8c-bd8l5    192.168.1.152
```

Service: stable endpoint for Pods

- A Service provides a stable virtual IP.
- It forwards traffic to matching Pods.



Service example

- Declarative YAML:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  selector:
    app: frontend
  ports:
    - name: http
      port: 80
      targetPort: 80
```

Service example

- Imperative command:

```
kubectl expose deployment frontend --port 80 --name frontend-service
```

Selector as the “glue”

- Deployment creates labeled Pods.
- Service selects Pods with matching labels.
- The Service does not care which node runs the Pod.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  selector:
    app: frontend
  ...
```

Inside-cluster DNS

- `CoreDNS` allows referencing to services via names:

  `<service>.<namespace>.svc.cluster.local`

- The full name is not necessary, depending where caller and callee are:

  `<service>` within the same namespace

  `<service>.<namespace>` from another namespace

Different types of services

- In this lecture:

ClusterIP

Expose a Service within the cluster only.

NodePort

Expose a Service on every node port.

- Outlook:

LoadBalancer

Ask cloud provider for an external load balancer.

Ingress

HTTP routing into the cluster.

NodePort Service

- Exposes on a port of every cluster node: `<node-ip>:<node-port>`
- Useful for local clusters and demos.

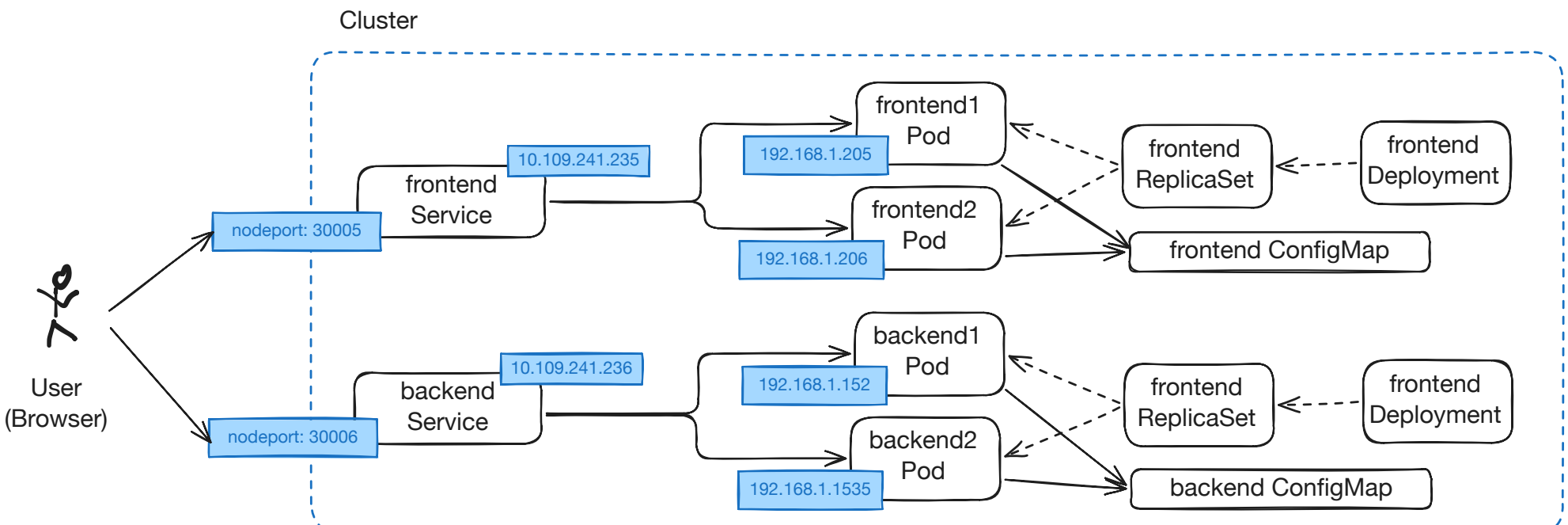
YAML

```yaml
apiVersion: v1
kind: Service
metadata:
  name: frontend-nodeport
spec:
  type: NodePort
  selector:
    app: frontend
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30082
```

Practice time!

Recap

Real applications are a composition of many resources



Recap

- Kubernetes is a **cluster abstraction** with a resource API.
- Developers mostly work with **declarative YAML manifests**.
- Core resources for a stateless app: Namespace, Deployment, Pod, ConfigMap, Service.
- Labels and selectors connect resources.
- **Controllers reconcile desired state into running infrastructure.**

Outlook

What else exists in Kubernetes?

Secrets

Sensitive config.

DaemonSets

One Pod per node.

Jobs / CronJobs

Tasks and schedules.

Volumes / Claims

Persistent storage.

StatefulSets

Stable Pod identity.

Ingress / Gateway

HTTP entry.

NetworkPolicies

Pod traffic rules.

Identity / RBAC

Accounts, roles, bindings.

Autoscaling

Metric-based replicas.

DisruptionBudgets

Maintenance availability.

Quotas / Limits

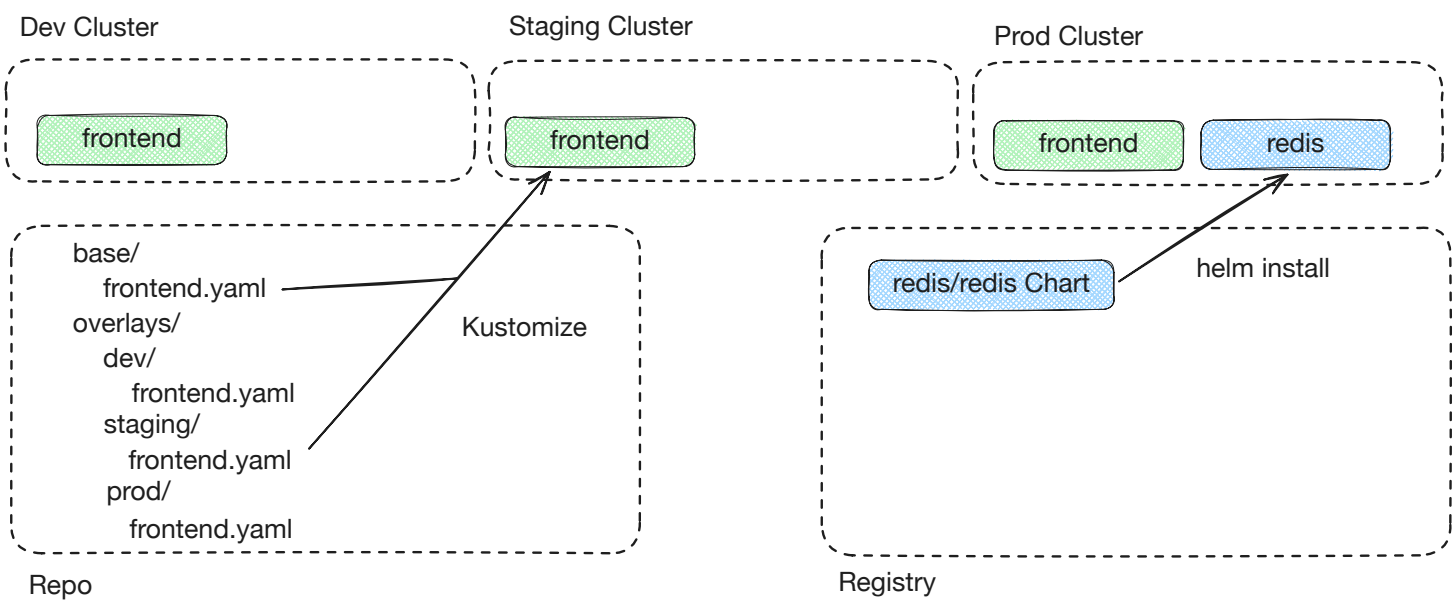
Namespace constraints.

CRDs

Custom resource types.

Tooling beyond raw YAML

- Larger systems need packaging and environment overlays.
- Common tools: Kustomize, Helm, Argo CD, Flux.



Platform engineering

- Kubernetes becomes the foundation for **internal developer platforms**.
- Platform teams package operational knowledge as self-service APIs:
  - Databases as managed services
  - Monitoring and alerting by default
  - Security and policy guardrails
  - Golden paths for deploying applications

Thanks for listening!