📖 Why Kubernetes Exists

> **Kubernetes** is an open-source platform for hosting containers and providing a cloud-neutral API to manage storage, networking, security, and other resources for applications.

- Provides **continuous reconciliation** of the desired state of deployments.
- Enables **reproducibility** of infrastructure, eliminating drift across identical components.
- Integrates **DevOps** practices directly into the application lifecycle.

🧩 Key Terms Review

> A quick reference for essential Kubernetes terminology.

- **CNI / CSI** – Container Networking Interface / Container Storage Interface, enabling pluggable networking and storage.
- **Container** – Docker or OCI image that runs an application.
- **Control plane** – "Brains" of the cluster (API server, scheduler, controller manager).
- **DaemonSet** – Runs a copy of a Pod on **every** node.
- **Deployment** – Manages a set of identical Pods.
- **kubectl** – CLI for interacting with the Kubernetes API server.
- **kubelet** – Node-side agent that enforces the desired state.
- **Node** – A machine (VM or physical) that runs a kubelet.
- **OCI** – Open Container Initiative image format.
- **Pod** – Smallest deployable unit; encapsulates one or more containers.

🏗️ Infrastructure Drift Problem

> Managing infrastructure as code prevents "drift" where servers diverge from their intended configuration.

Common manual toil avoided by Kubernetes:

Manual Task	Why It Causes Drift	Kubernetes Solution
Updating Java version on many servers	Missed hosts → inconsistent runtimes	Declarative Pod/Deployment specs
Scaling hardware manually	Human error, delayed provisioning	Autoscaling controllers
Hand-crafting load-balancer rules	Inconsistent routing	Service objects + kube-proxy
Forgetting to document changes	Knowledge loss	Version-controlled YAML manifests

🐋 Containers & Images

> **Containers** add isolation, allowing multiple versions of the same library to coexist.

```
# Dockerfile for a MySQL image
FROM alpine:3.15.4
RUN apk add --no-cache mysql
ENTRYPOINT ["/usr/bin/mysqld"]
```

- Images are **OCI tarballs** with layered filesystem contents.
- Running a container unpacks the image and starts a process on the host.
- Example of running two containers manually:

```
docker run -t -i ui -p 80:80
docker run -t -i microservice-a -p 8080:8080
```

```
docker run -t -i microservice-b -p 8081:8081
docker run -t -i cockroach:cockroach -p 6379:6379
```

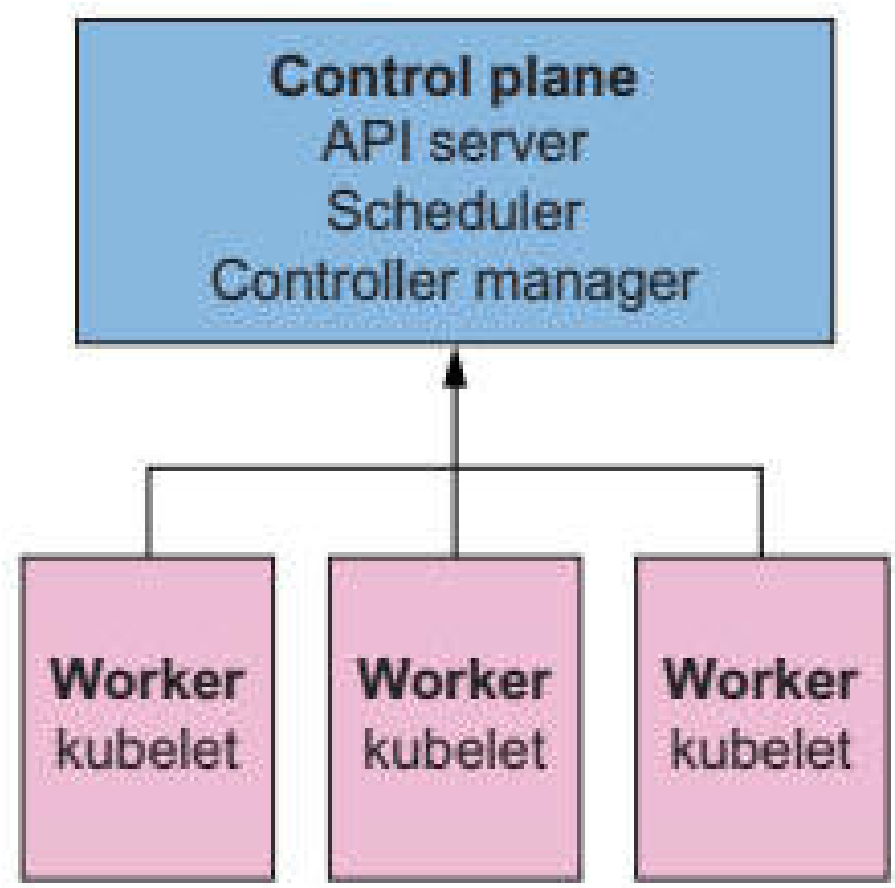
🛠️ Core Foundation of Kubernetes

- All objects are defined as **plain YAML** (or JSON) files.
- The same syntax configures Pods, Services, RBAC, Storage, etc.
- Example: building a simple MySQL pod.

```
apiVersion: v1
kind: Pod
metadata:
  name: core-mysql
spec:
  containers:
    - name: mysql
      image: myregistry.com/mysql-server:v1.0
```

```
kubectl create -f core-mysql.yaml
```

🖥️ Kubernetes Architecture & Components



The diagram visualizes the control plane (API server, scheduler, controller manager) communicating with multiple worker nodes, each running a kubelet.

Key components:

- **Control plane** – API server, scheduler, controller manager, etcd.
- **Worker nodes** – Run kubelet, container runtime, kube-proxy, and a CNI plugin.
- **etcd** – Consistent key-value store for cluster state.

🧭 The Kubernetes API

> The API server exposes a **RESTful** interface for CRUD operations on all cluster objects.

- Objects have **apiVersion**, **kind**, and **metadata**.
- Example of listing API resources:

```
kubectl api-resources | head
```

Output snippet:

```
NAME            SHORTNAMES   NAMESPACED   KIND
bindings        ...          false        Binding
componentstatuses cs         false        ComponentStatus
configmaps      cm           true         ConfigMap
...
```

🛒 Example Use Cases

1️⃣ Online Retailer

- Needs rapid seasonal scaling.
- Benefits from Kubernetes auto-scaling, high availability, and cloud-agnostic deployment.

2️⃣ Online Giving Platform

- Grew from a single VM to many microservices (Python, Java, NGINX, etc.).
- Migration to Kubernetes reduced VM count from ~30 to 5, eliminated Puppet, and improved disaster recovery.

❓ When Not to Use Kubernetes

Scenario	Reason
High-Performance Computing (HPC)	Container overhead can add latency; nanosecond precision may be required.
Legacy monoliths with strict hardware dependencies	Hard to containerize or unsupported by vendors.
Rigid legacy migrations	Minimal benefit if the app cannot be broken into microservices.

🟢 The Pod Concept

> A **Pod** is the smallest Kubernetes object that encapsulates one or more containers sharing Linux namespaces.

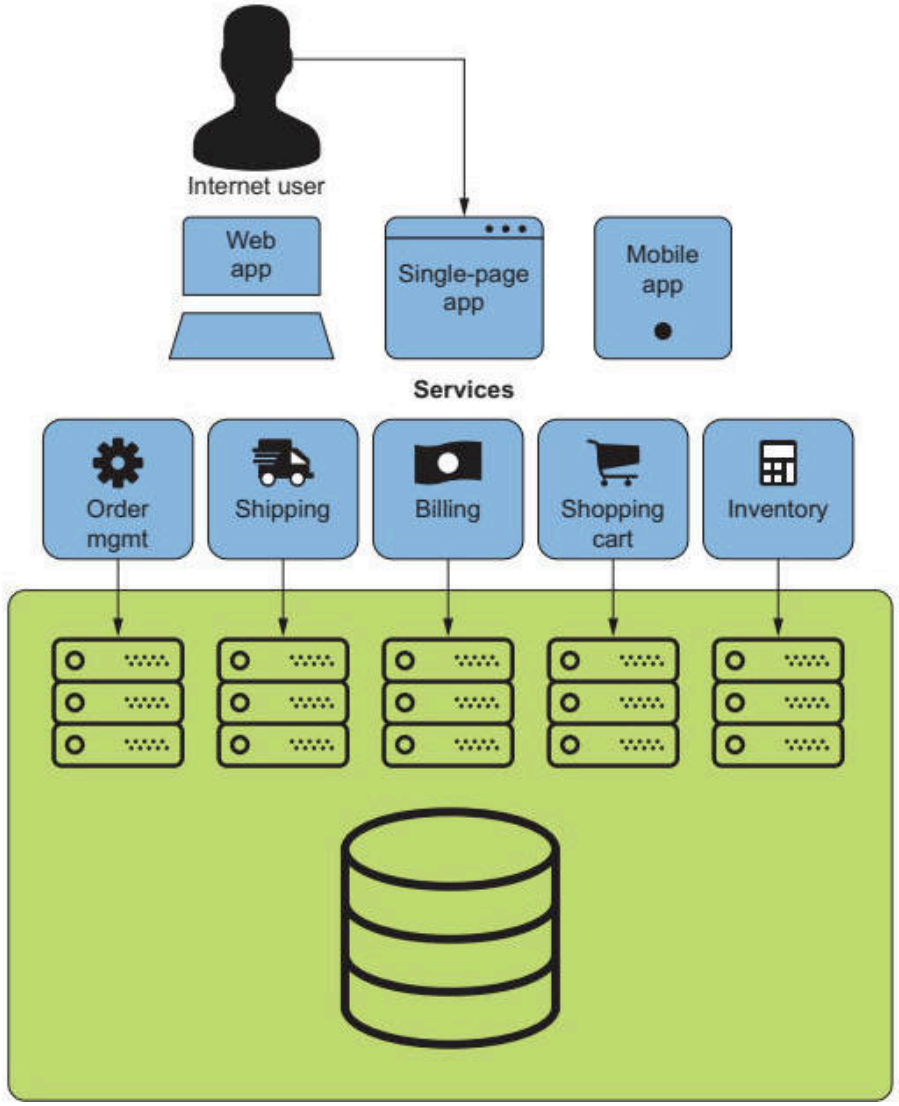
- Provides a **network namespace**, **PID namespace**, **IPC namespace**, **cgroup**, **mount**, and **user** namespaces.
- Pods are created via higher-level controllers (Deployments, StatefulSets, DaemonSets).

Pod YAML Example

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

```
kubectl apply -f busybox-pod.yaml
```

🌐 Example Web Application (Zeus Zap)



The diagram breaks the system into UI, services, and infrastructure layers, illustrating how microservices communicate.

Components:

- **UI** – NGINX serving a single-page app.

- **Microservices** – Two Python/Django services.
- **Database** – CockroachDB (PostgreSQL-compatible) on port 6379.
- **Messaging, caching, search** – Additional services (Redis, Elasticsearch, etc.).

Operational Challenges Highlighted

- **Port conflicts** when running multiple UI containers.
- **Dynamic IP handling** for database relocation.
- **High availability** for the database.
- **Burst traffic** from e-sport events.

🏢 Infrastructure for Web Application

To run the above stack without orchestration, you'd need:

- Multiple VMs or physical servers.
- Load balancers, DNS, persistent storage, security policies, CI/CD pipelines, monitoring, and logging.

Kubernetes abstracts all these via declarative objects, reducing manual toil.

📦 Pod Definition & Creation

```
apiVersion: v1
kind: Pod
metadata:
  name: zeus-ui
spec:
  containers:
    - name: nginx
      image: nginx:1.21
      ports:
        - containerPort: 80
```

```
kubectl apply -f zeus-ui.yaml
```

- The **kubelet** on the chosen node watches the API for this Pod, pulls the image via the CRI, and starts it.
- A **pause container** is launched first to set up the network namespace.

👷 Linux Namespaces in Pods

Namespace	Purpose
PID	Isolates process IDs.
Network	Gives each Pod its own IP address.
IPC	Isolates inter-process communication (shared memory, semaphores).
cgroup	Controls CPU, memory, and I/O limits.
Mount	Provides a separate filesystem view.
User	Maps user IDs inside the container.

These namespaces enable the Pod to behave like a lightweight VM while sharing the host kernel.

🧩 Node API Object

> A **Node** is a Kubernetes API object representing a worker machine.

```
apiVersion: v1
kind: Node
metadata:
  name: kind-control-plane
  labels:
    kubernetes.io/hostname: kind-control-plane
spec:
  podCIDR: 10.244.0.0/24
```

```
apiVersion: v1
kind: Node
metadata:
  annotations:
    kubeadm.alpha.kubernetes.io/cri-socket:
      /run/containerd/containerd.sock
```

The CRI socket used. With kind (and most clusters), this is the containerd socket.

The snippet shows a Node's metadata, spec, and status sections, including its CIDR, capacity, and condition fields.

Key fields:

- **metadata.annotations** – e.g., node.alpha.kubernetes.io/ttl.
- **spec.podCIDR** – IP range for Pods on this node.
- **status.conditions** – MemoryPressure, DiskPressure, PIDPressure, Ready, etc.

📈 Control Plane & Scaling

- The **scheduler** assigns Pods to Nodes based on resource availability, affinity/anti-affinity, and taints.
- **Controller manager** runs loops that ensure desired state (e.g., replica count, StatefulSet ordering).
- **Autoscaling** options:
  1. **Horizontal Pod Autoscaler (HPA)** – Adds/removes Pods.
  2. **Vertical Pod Autoscaler (VPA)** – Adjusts CPU/memory requests.
  3. **Cluster Autoscaler** – Adds/removes Nodes.

Scaling Example

```
# Increase UI deployment to 300 replicas
kubectl scale --replicas=300 deployment zeus-front-end-ui
```

Sequence of events (illustrated in the book):

1. kubectl scale updates the Deployment's spec.
2. The **ReplicationController** notices the change.
3. The **scheduler** picks Nodes for new Pods.
4. The **kubelet** on each Node creates the Pods via the CRI.

🔧 Autoscaling & Cost Management

- **Pod density** – Packing many Pods onto a node reduces infrastructure cost.
- Steps to achieve optimal density:
  1. **Profile** each application's CPU/memory usage.
  2. **Select** node sizes that can host multiple Pods comfortably.
  3. **Group** compatible workloads using **taints**, **tolerations**, and **affinity** rules.
- **Noisy neighbor** mitigation: use **PodPriority**, **resource quotas**, and **QoS classes**.

> **Cost-saving tip:** Shut down development clusters during weekends (set node count to zero) and spin them up on demand.

📚 Summary Highlights

- **Pod** = atomic deployment unit built from Linux namespaces.

- **Control plane** orchestrates Pods, storage, networking, and scaling.
- **kubectl** is the single tool for all CRUD operations against the API server.
- **Declarative YAML** drives the entire cluster, from Nodes to Services.
- **Autoscaling** and **pod density** are core mechanisms for balancing performance with cost.

---## 🛠️ Fundamental Linux Primitives in Kubernetes

- **kube-proxy** – creates *iptables* rules that route Service traffic to Pods.
- **Container Network Interface (CNI)** – uses the same network proxy for implementing **NetworkPolicies**.
- **Container Storage Interface (CSI)** – socket used by the *kubelet* to talk to storage back-ends (e.g., EBS, NFS).
- **Container runtime commands** – unshare, mount, nsenter, kill, ps, ls etc. are the building blocks for isolation, networking and process management.

> **Definition** – *Linux primitives* are low-level OS tools that manipulate files, devices, sockets or namespaces; Kubernetes composes them to implement Pods, Services, and other resources.

Key Commands Table

Command	Primary Use	Security Implication
swapoff	Disable swap (required for deterministic CPU/Memory accounting)	Prevents memory paging that could hide OOM events
iptables	Set up network forwarding / firewall rules (used by kube-proxy)	Misconfiguration can expose services
mount	Bind a device or directory into a namespace	Improper binds may leak host data
systemd	Starts the *kubelet* process	Central to node lifecycle
socat	Bidirectional stream forwarding (used by kubectl port-forward)	Allows remote access to pods
nsenter	Enter another process's namespaces (PID, network, mount)	Enables deep debugging
unshare	Create new namespaces for a child process (PID, net, mount)	Provides strong isolation
ps	List processes (used to detect zombies, rogue pods)	Visibility into container health

---

📦 Building and Running Pods

Minimal BusyBox Pod YAML

```
apiVersion: v1
kind: Pod
metadata:
  name: core-k8s
  labels:
    role: just-an-example
    app: my-example-app
spec:
  containers:
  - name: busybox
    image: docker.io/busybox:latest
    command: ['sleep','10000']
    ports:
    - name: webapp-port
      containerPort: 80
      protocol: TCP
```

- **Labels** enable selection by Services or kubectl queries.
- The **command** runs a long-running sleep so the Pod stays alive for inspection.

Inspecting the Pod

```
# List all Pods (all namespaces)
kubectl get pods --all-namespaces

# Show only the Pod phase
kubectl get pods -o=jsonpath='{.items[0].status.phase}'
# → Running

# Show Pod IP
kubectl get pods -o=jsonpath='{.items[0].status.podIP}'
# → 10.244.0.11

# Show host IP
kubectl get pods -o=jsonpath='{.items[0].status.hostIP}'
# → 172.17.0.2
```

🔧 Linux Commands & Utilities for Kubernetes

Utility	Role in Pod Lifecycle
swapoff	Guarantees that memory limits are respected.
iptables	Implements Service routing (kube-proxy).
mount	Exposes storage volumes inside the container.
systemd	Starts the kubelet, which drives the control loop.
socat	Powers kubectl port-forward.
nsenter	Enters a Pod's namespaces for debugging.
unshare	Creates isolated PID, network, and mount namespaces.
ps	Checks for zombie processes or runaway containers.

🧩 "Everything Is a File" (or File Descriptor)

> **Quote** – *"Everything is a file"* is a core Linux principle; devices, sockets, and even process information appear as files under /dev, /proc, or /sys.

- **Directories** are files containing lists of other file names.
- **Device files** (/dev/eth0) let tools like ls verify device presence.
- **Sockets & pipes** appear as special files enabling inter-process communication.

The **/proc** filesystem provides real-time process metadata, e.g., /proc/<PID>/cgroup.

📂 Using chroot to Isolate a Process

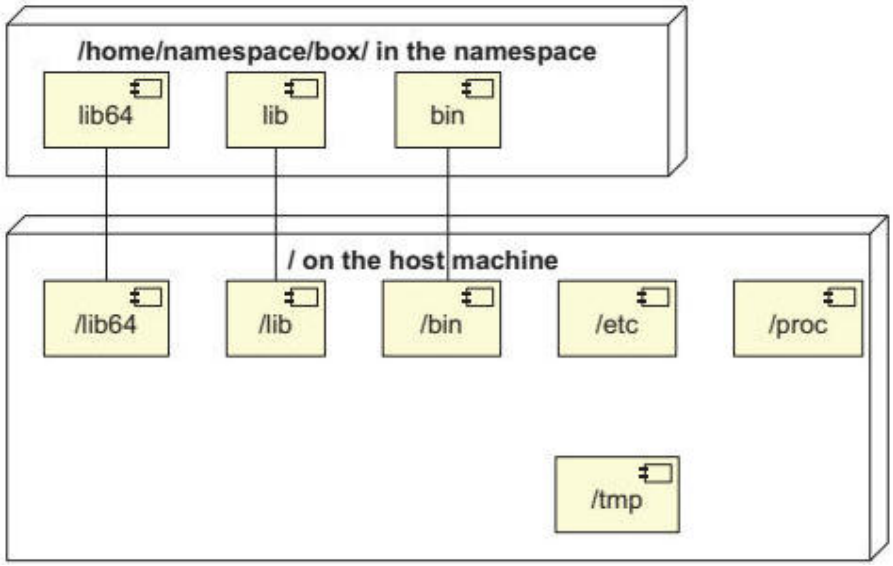
The following script creates a minimal sandbox (a "poor-man's VM") and drops into a Bash shell:

```
#!/bin/bash
mkdir -p /home/namespace/box/{bin,lib,lib64,proc}
cp -v /usr/bin/kill /home/namespace/box/bin/
cp -v /usr/bin/ps   /home/namespace/box/bin/
```

```
cp -v /bin/bash    /home/namespace/box/bin/
cp -v /bin/ls      /home/namespace/box/bin/
cp -r /lib/*      /home/namespace/box/lib/
cp -r /lib64/*    /home/namespace/box/lib64/
mount -t proc proc /home/namespace/box/proc
chroot /home/namespace/box /bin/bash
```

- **Result:** The Bash process sees only the files explicitly copied into /home/namespace/box.
- **Use case:** Safe testing of destructive commands (rm -rf /) without harming the host.

Figure 3.3 – chroot vs Host Filesystem



The diagram shows how the chrooted environment mounts only a subset of the host's directories, illustrating the isolation achieved by chroot.

📂 Using mount for Data Access

```
# Bind-mount /tmp into the sandbox as /data
mount --bind /tmp /home/namespace/box/data
```

- The sandbox can now read/write /data/*, which maps to the host's /tmp.
- **Security note:** Bind-mounts expose host data; Kubernetes often disables hostPath in production for this reason.

🔐 Securing Processes with unshare

Running the previous chroot script inside an unshare namespace prevents the sandbox from seeing or killing host processes (e.g., the kubelet).

```
unshare -p -n -f --mount-proc=/home/namespace/box/proc \
  chroot /home/namespace/box /bin/bash
```

- **-p** – new PID namespace (the process thinks it is PID 1).
- **-n** – new network namespace (isolated lo interface).
- **-f** – fork after unsharing.

Figure 3.4 – unshare Process Isolation

The output demonstrates a tiny process table visible inside the new namespace, confirming isolation from host processes.

🌐 Creating a Network Namespace

```
unshare -p -n -f --mount-proc=/home/namespace/box/proc \
  chroot /home/namespace/box /bin/bash
# Inside the shell
ip a   # Shows only lo and any manually added interfaces
```

- No eth0 is present because a CNI plugin has not been invoked.
- Real Pods receive an eth0 with an IP from the cluster's overlay network (e.g., 10.244.0.x).

📊 Inspecting Pods and Services

Using iptables to View Service Rules

```
# Show all iptables rules that reference cluster IPs
iptables-save | grep -E 'KUBE-SVC|KUBE-SEP'
```

- **KUBE-SVC-...** rules forward traffic from a Service IP to one of its endpoints.
- **KUBE-SEP-...** rules perform DNAT to the Pod's IP and port.

kube-dns Example

```
kubectl exec -ti core-k8s -- mount | grep resolv.conf
# → /dev/sda1 on /etc/resolv.conf type ext4 (rw,relatime)
```

- The DNS configuration is mounted from the host, enabling Pods to resolve service names.

📦 Kind: Local Kubernetes Cluster Setup

1. Install **Docker**, **kubectl**, and **kind**.
2. Create a cluster:

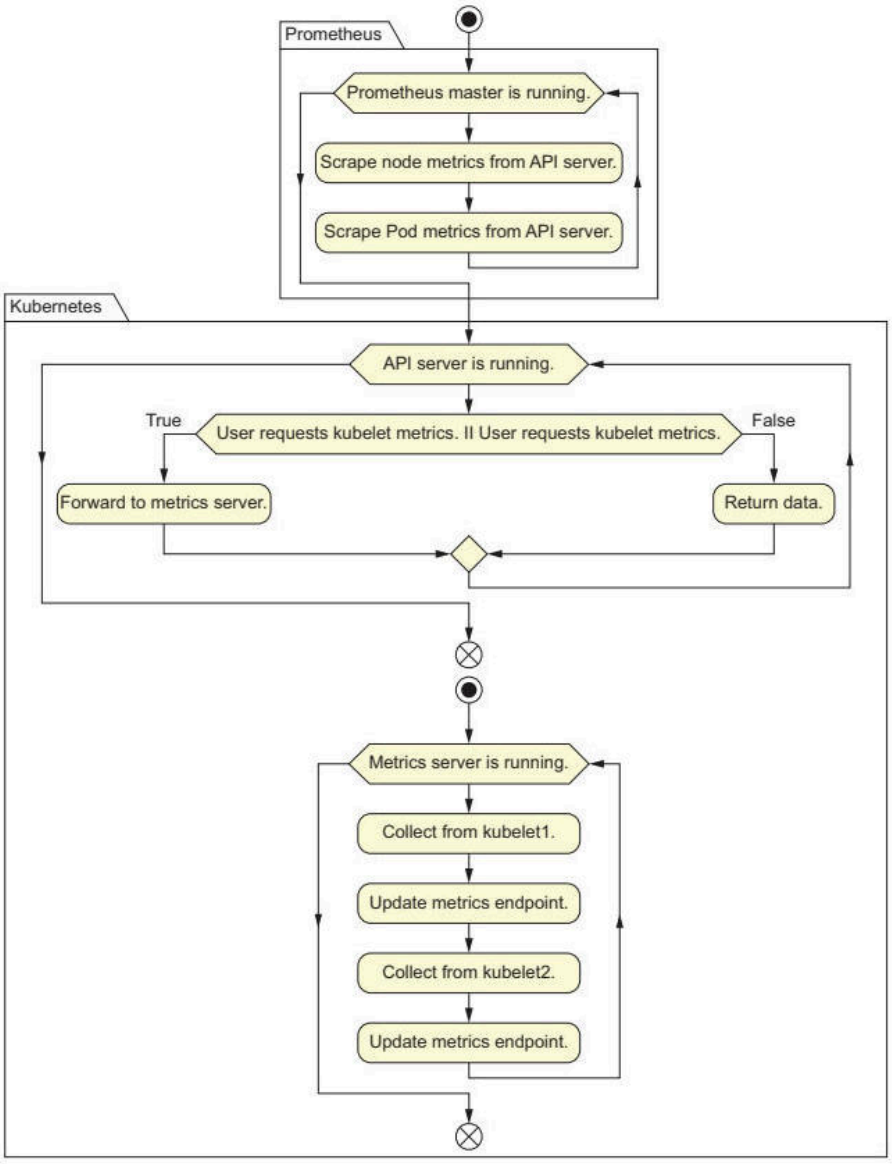
```
kind delete cluster --name=kind   # Clean up any previous cluster
kind create cluster
```

3. Verify:

```
kubectl get pods --all-namespaces
docker ps   # Shows the kind node container
```

```
docker exec -it <container-id> /bin/sh   # SSH-like access to the node
```

Figure 4.1 – Kind Cluster Workflow



Kind builds a node inside a Docker container, enabling rapid local experimentation.

📈 cgroups: Resource Management

- **cgroups** (control groups) partition CPU, memory, I/O, etc. among processes.
- Kubernetes creates a cgroup per Pod under /sys/fs/cgroup/.../kubepods/....

Example: Limiting Memory to 10 bytes

```
mkdir /sys/fs/cgroup/memory/chroot0
echo "10" > /sys/fs/cgroup/memory/chroot0/memory.limit_in_bytes
```

```
echo "0"  > /sys/fs/cgroup/memory/chroot0/memory.swappiness
echo $$  > /sys/fs/cgroup/memory/chroot0/tasks   # Add current PID
```

Running ls now fails with *“Cannot allocate memory”*.

Resources Stanza (Greedy Pod)

```
apiVersion: v1
kind: Pod
metadata:
  name: greedy-pod
spec:
  containers:
  - name: busybox
    image: busybox:latest
    command: ['sleep','10000']
    resources:
      limits:
        memory: "200Mi"
      requests:
        memory: "100Mi"
```

```
$ cat << EOF > greedy-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: core-k8s-greedy
spec:
  containers:
    - name: any-old-name-will-do
      image: docker.io/busybox:latest
      command: ['sleep','10000']
      resources:
       limits:
          memory: "200Mi"
        requests:
          memory: "100Mi"
EOF
```

Tells Kubernetes to create a cgroup to limit (or not) available CPU

The manifest tells the kubelet to create a cgroup that caps the container’s memory.

Allocatable Resources Equation

> **Formula** – $\text{Allocatable} = \text{Node capacity} - \text{kube-reserved} - \text{system-reserved}$

- **Node capacity** – total CPU, memory, storage on the node.
- **kube-reserved** – resources needed by the kubelet, container runtime, etc.
- **system-reserved** – OS daemon overhead.

QoS Classes

QoS Class	Definition	Typical Use
Guaranteed	*Both* requests **and** limits are set for **CPU** *and* **memory**.	Critical services that must not be evicted.

Burstable	requests are set, but limits may be higher or omitted.	Workloads that need a baseline but can burst.
BestEffort	No requests or limits defined.	Low-priority jobs; first to be killed under pressure.

Kubernetes automatically assigns the class based on the Pod's resources stanza.

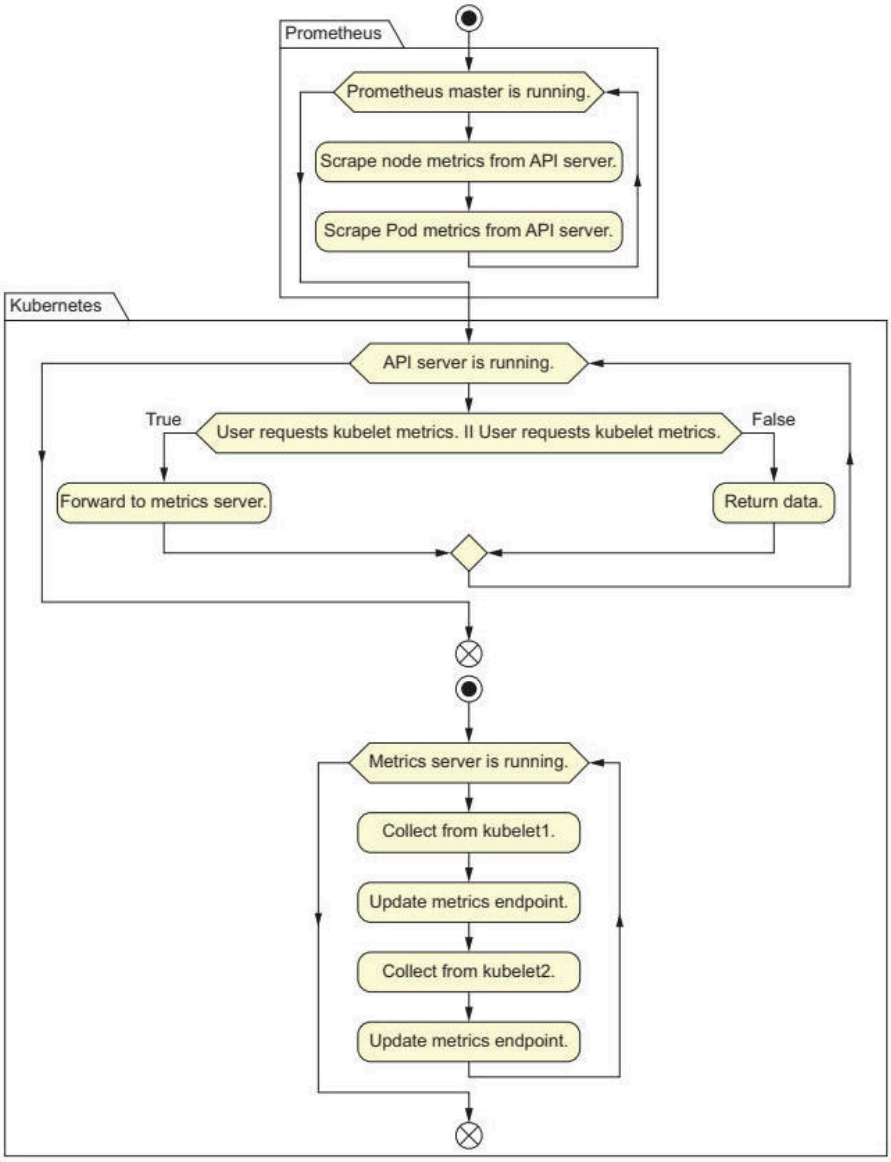
📈 Monitoring with Prometheus & cAdvisor

- **cAdvisor** (built into the kubelet) exposes per-cgroup metrics (CPU, memory, network).
- **Prometheus** scrapes these metrics from the API server's /metrics/cadvisor endpoint.

Prometheus Configuration (scrape every 3 s)

```
global:
  scrape_interval: 3s
  evaluation_interval: 3s
scrape_configs:
  - job_name: prometheus
    metrics_path: /api/v1/nodes/kind-control-plane/proxy/metrics/cadvisor
    static_configs:
      - targets: ['localhost:8001']
```

Figure 4.2 – Prometheus Monitoring Architecture



The flowchart shows Prometheus pulling metrics from the API server, which aggregates data from kubelets (cAdvisor).

- **Metrics Types**
  - **Gauge** – current value (e.g., CPU usage).
  - **Counter** – monotonically increasing (e.g., total requests).
  - **Histogram** – distribution of latency or size.
- Example query to see etcd latency:

```
kubectl proxy &
curl localhost:8001/metrics | grep etcd_request_duration_seconds_bucket
```

📦 Summary of Core Concepts

- **Linux primitives** (iptables, mount, unshare, cgroups) are the scaffolding behind every Kubernetes feature.
- **Pods** are essentially isolated processes created with chroot, mount, and namespace isolation.

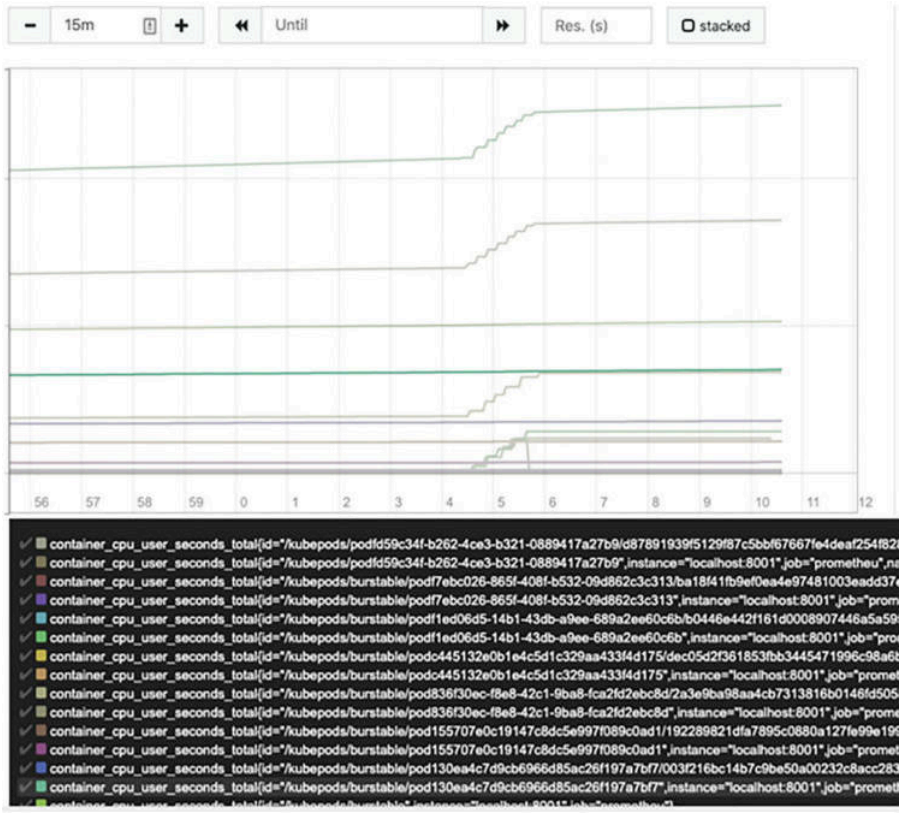
- **cgroups** enforce resource limits; QoS classes derive from the resources stanza.
- **Kind** provides a lightweight Docker-based test cluster; useful for hands-on experimentation.
- **Prometheus + cAdvisor** give deep visibility into kernel-level resource usage, enabling proactive troubleshooting. ## 📊 Prometheus Metrics & Outage Characterization

Key metrics to monitor in a busy cluster

Metric	Description
container_memory_usage_bytes	Current memory consumption of a container.
container_fs_writes_total	Total number of filesystem write operations.
container_memory_cache	Amount of memory used for caching.

> **Definition** – *Gauge*: a metric that represents a single numerical value that can go up or down (e.g., cluster "up" flag).
> **Definition** – *Histogram*: records the distribution of a set of values (e.g., request latency).
> **Definition** – *Counter*: a cumulative metric that only increases (e.g., total requests served).

Figure 4.3 – Plotting metrics in a busy cluster



The line graph visualizes spikes in container_cpu_user_seconds_total after running a stress-test YAML. Hovering over the metrics links the values back to the specific containers and processes generating them.

Example: Generating Load

```
# WARNING: this creates heavy CPU and network load
kubectl apply -f https://raw.githubusercontent.com/giantswarm/kube-stresscheck/master/examples/node.yaml
```

Running the above on a laptop will cause noticeable CPU spikes and fan noise, illustrating how Prometheus captures rapid changes in container activity.

🌐 Kubernetes Service Types & SDN Basics

Service Type	IP Scope	Typical Use
ClusterIP	Internal only	Service-to-service communication inside the cluster.
NodePort	Exposes on every node's IP	Simple external access; maps a random high port (e.g., 50491) to the service port.
LoadBalancer	External cloud IP (if supported)	Public-facing services; cloud provider provisions a load-balancer IP.

> **Definition** – *kube-proxy*: the component that programs kernel-level routing (iptables/IPVS) to forward Service IPs to the appropriate Pod IPs.

Service creation example (ClusterIP)

```
kubectl create service clusterip my-service-1 --tcp="100:100"
```

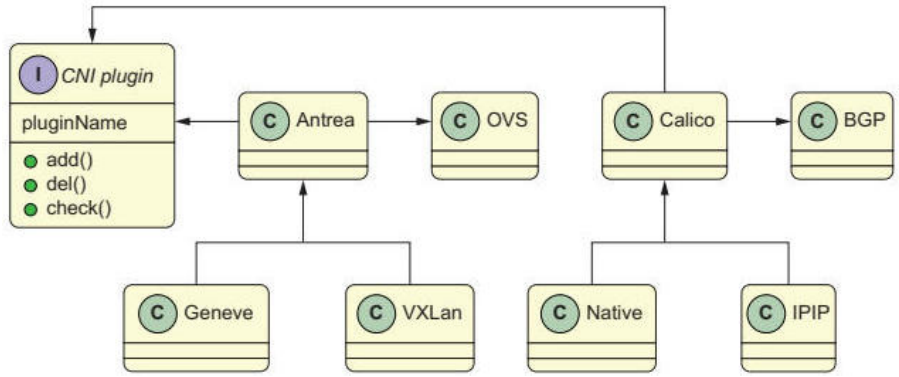
Repeated three times yields three distinct Service IPs in the 10.96.0.0/12 range, e.g.:

```
svc-1  ClusterIP 10.96.7.53   80/TCP
svc-2  ClusterIP 10.96.152.223 80/TCP
svc-3  ClusterIP 10.96.43.92   80/TCP
```

NodePort conversion – modify the Service YAML to set type: NodePort; Kubernetes allocates a random port (e.g., 30357) that forwards to the original Service port.

🧭 kube-proxy Routing Flow

Figure 5.1 – Traffic flow from a LoadBalancer into a Kubernetes cluster



The diagram shows how external traffic hits the LoadBalancer IP, is translated to a Service IP, then routed by kube-proxy (iptables or IPVS) to the target Pod.

Key points

1. **External IP** (LoadBalancer) → **Service IP** (ClusterIP)

2. **kube-proxy** installs NAT rules (DNAT/SNAT) in the kernel.
3. Packets reach the Pod's IP on the node where the Pod lives.

kube-proxy operates in three modes: iptables, ipvs, and userspace. Most clusters default to iptables.

🛠️ CNI Specification & Provider Landscape

> **Definition** – *CNI (Container Network Interface)*: a generic specification that defines how a container runtime requests network resources (add, delete, check).

CNI provider categories

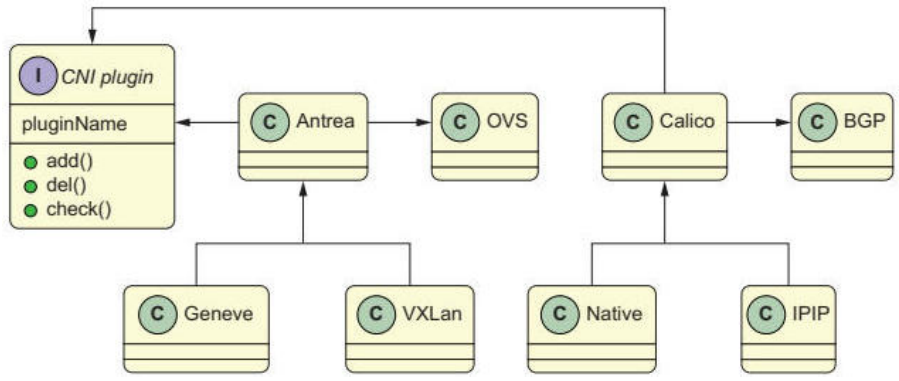
Provider	Primary Technology	Notable Features
Calico	BGP (Layer 3)	Supports IPIP, VXLAN, XDP; can replace kube-proxy.
Antrea	Open vSwitch (Layer 2)	Bridges pods via OVS; includes Antrea Proxy.
Flannel	Simple bridge	Legacy, minimal feature set.
Cilium	eBPF/XDP	High-performance, kernel-bypass networking.
KindNet	Minimal CNI for kind	Single-subnet, for local testing only.
Cloud-specific (GKE, EKS, Azure)	Proprietary	Tight integration with cloud load-balancers.

The CNI **binary** implements three core commands:

1. **ADD** – attach a container to a network.
2. **DELETE** – detach a container.
3. **CHECK** – verify proper setup.

🔌 Calico vs. Antrea Architecture

Figure 5.2 – CNI networking in Calico and Antrea



Both plugins expose add(), del(), check() to the container runtime, then use different data-plane technologies (BGP vs. OVS) to program routes.

Aspect	Calico	Antrea
Routing	Layer 3 BGP routes per node	Layer 2 OVS bridge per node
Tunnel types	IPIP, VXLAN, Geneve	Geneve (via OVS)

NetworkPolicy	iptables rules	OVS flow rules (table 90)
Proxy replacement	Can replace kube-proxy (via BGP)	Antrea Proxy (OVS-based)
Deployment	DaemonSet + controller pod	DaemonSet + controller pod

Both use a DaemonSet to run a per-node agent and a controller pod to aggregate policies.

⚙️ Installing Calico in a kind Cluster

1. Disable the default kind-net CNI (allows a real CNI).
2. Create a custom kind config (sets a large pod subnet).

```
# kind-Calico-conf.yaml
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
networking:
  disableDefaultCNI: true
  podSubnet: 192.168.0.0/16
nodes:
  - role: control-plane
  - role: worker
```

```
kind create cluster --name=calico --config=kind-Calico-conf.yaml
```

3. Install Calico manifests

```
wget https://docs.projectcalico.org/manifests/calico.yaml
kubectl apply -f calico.yaml
```

*After installation, kubectl get pods -n kube-system should show calico-node and calico-kube-controllers pods in **Ready** state.*

Verifying Calico routes

```
# On a Calico node
ip route
```

Typical output includes routes via tunl0 for remote node subnets and cali* interfaces for local pods.

🧩 Installing Antrea in a kind Cluster

```
# kind-Antrea-conf.yaml
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
networking:
  disableDefaultCNI: true
  podSubnet: 192.168.0.0/16
nodes:
  - role: control-plane
  - role: worker
```

```
kind create cluster --name=antrea --config=kind-Antrea-conf.yaml
kubectl apply -f https://github.com/vmware-tanzu/antrea/releases/download/v0.8.0/antrea.yml -n kube-system
```

Antrea-specific interfaces

- antrea-gw0 – OVS gateway handling all pod traffic.
- ovs-system – OVS bridge managing the virtual switch.
- genev_sys_* – Geneve tunnel endpoints.

```
# Show Antrea OVS flows (policy table 90)
kubectl -n kube-system exec -it $(kubectl get pod -n kube-system -l app=antrea-agent -o name) -- ovs-ofctl du
```

📡 CNI Routing & Tunnels

CNI	Tunnel Device	Routing Style
Calico	tunl0 (IPIP)	Per-Pod route entries (many).
Antrea	antrea-gw0 (Geneve)	One gateway route per node (.1 address).
Flannel	flannel.1 (VXLAN)	Simple overlay, one route per node.

In Calico, each remote pod appears as a separate route entry; Antrea collapses traffic to a single gateway address, simplifying the kernel routing table.

🔐 NetworkPolicy Implementation

> **Definition** – *NetworkPolicy*: a Kubernetes API object that defines allow-list rules for pod traffic (ingress, egress, or both).

Simple deny-all policy

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: web-deny-all
spec:
  podSelector:
    matchLabels:
      app: web
  ingress: []   # no allowed inbound traffic
```

Allow-only port 80 from app: web2

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: web-allow-http
spec:
  podSelector:
    matchLabels:
      app: web
  ingress:
  - ports:
    - port: 80
    from:
    - podSelector:
        matchLabels:
          app: web2
```

How policies appear in the data plane

- **Calico** – adds iptables chains (cali-tw-…) with MARK and DROP rules.
- **Antrea** – creates OVS flows in table 90, using conjunction rules to combine multiple criteria.

Inspecting Calico iptables changes

```
iptables-save | grep -A2 "cali-tw"
```

Inspecting Antrea OVS flows

```
kubectl -n kube-system exec -it $(kubectl get pod -n kube-system -l app=antrea-agent -o name) -- ovs-ofctl du
```

🌉 Ingress Controllers & Contour Example

Ingress purpose – expose HTTP/HTTPS services on a single external IP, routing based on hostnames/paths.

Deploy Contour in kind

```
git clone https://github.com/projectcontour/contour.git
kubectl apply -f contour/examples/contour
```

Add a host entry for local testing:

```
echo "127.0.0.1 my-service.local" | sudo tee -a /etc/hosts
```

Ingress resource for my-service

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
spec:
  rules:
  - host: my-service.local
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: my-service
            port:
              number: 8080
```

Figure 4.6 – Ingress routing flow (illustrative; not in transcript)

> **Note** – The upcoming **Gateway API** will eventually replace the Ingress API, offering richer routing primitives.

💾 Pod Storage Fundamentals

> **Definition** – *PersistentVolume (PV)*: a cluster-wide storage resource provisioned by an administrator.
> **Definition** – *PersistentVolumeClaim (PVC)*: a request for storage by a pod.
> **Definition** – *StorageClass*: a profile that determines how a PV is dynamically provisioned (e.g., standard, local-path).

Example PVC & Pod (dual-volume)

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: dynamic1
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 100Ki
---
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: busybox
    image: busybox
    volumeMounts:
    - name: shared
      mountPath: /shared
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
    volumeMounts:
    - name: dynamic1
      mountPath: /var/www
    - name: shared
      mountPath: /shared
  volumes:
  - name: dynamic1
    persistentVolumeClaim:
      claimName: dynamic1
  - name: shared
    emptyDir: {}
```

Result: busybox and nginx share /shared (ephemeral emptyDir), while nginx writes persistent data to /var/www backed by the dynamically provisioned PVC.

Verifying PVC provisioning

```
kubectl get sc
# Example output:
# NAME    PROVISIONER          RECLAIMPOLICY   AGE
# standard (default) rancher.io/local-path   Delete   9d
```

📌 Summary of Core Concepts

- **Prometheus** provides gauges, histograms, and counters to observe cluster health and pinpoint outages.
- **kube-proxy** translates Service IPs to Pod IPs using iptables/IPVS; it is distinct from the CNI data plane.
- **CNI plugins** (Calico, Antrea, etc.) attach pods to a network, each employing different routing (BGP vs. OVS).

- **NetworkPolicies** enforce pod-level firewalls; Calico uses iptables, Antrea uses OVS flows.
- **Ingress controllers** (e.g., Contour) map external HTTP traffic to internal Services; the future **Gateway API** will extend this model.
- **Persistent storage** in Kubernetes is abstracted via PV, PVC, and StorageClass, allowing pods to retain data across restarts.

These notes are modular; each ## section can be combined with other study guides as needed.## 📦 Dynamic Storage Provisioning

> **Dynamic provisioning** automatically creates a PersistentVolume (PV) when a PersistentVolumeClaim (PVC) is submitted, without manual PV creation.

- The controller watches for PVCs, calls the provisioner (e.g., **local-path-provisioner** bundled with kind), and creates a PV that is immediately bound to the PVC.
- After binding, the scheduler confirms the PVC is deployable, moving the Pod from **Pending** → **ContainerCreating** → **Running**.

```
$ kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
pvc-74879bc4-e2da-4436-9f2b-5568bae4351a   100k       RWO            Delete           Bound    default/dynami
```

*The standard **StorageClass** (see image below) tells Kubernetes how to provision the volume.*

```
$ kubectl get sc -o yaml
apiVersion: v1
items:
- apiVersion: storage.k8s.io/v1
  kind: StorageClass
  metadata:
    annotations:
      kubectl.kubernetes.io/last-applied-configuration: |
        {"apiVersion":"storage.k8s.io/v1",
           "kind":"StorageClass","metadata":{
         "annotations":{
                "storageclass.kubernetes.io/is-default-class": "true"}
              ,"name":"standard"
            },
            "provisioner":"rancher.io/local-path",
            "reclaimPolicy":"Delete",
          "volumeBindingMode":"WaitForFirstConsumer"}
      storageclass.kubernetes.io/is-default-class: "true"
    name: standard
  provisioner: rancher.io/local-path
kind: List
```

The is-default-class makes this the go-to volume for Pods wanting storage without needing to explicitly request a storage class.

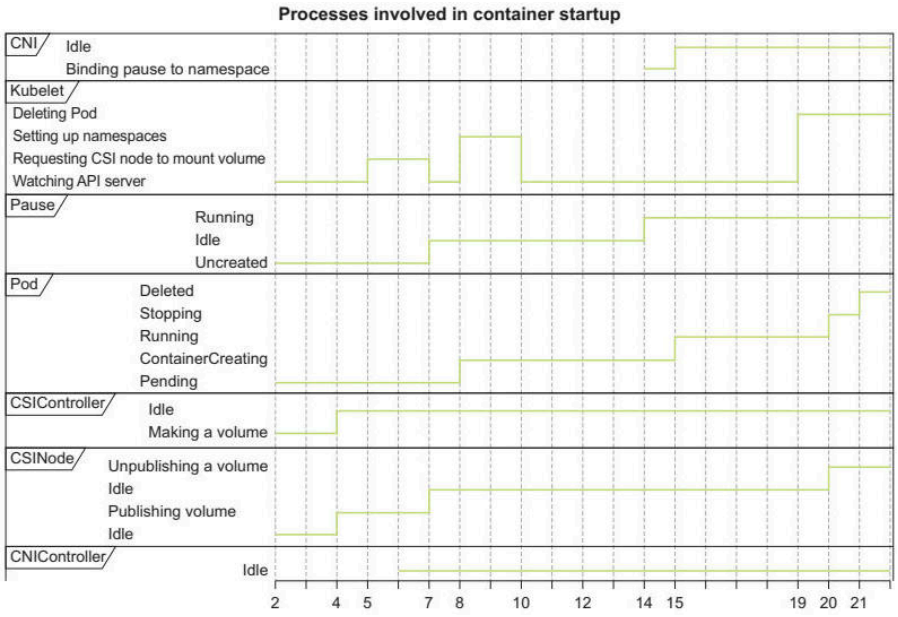
You can have many different storage classes in a cluster.

*The YAML defines a standard StorageClass with the rancher.io/local-path provisioner, a Delete reclaim policy, and WaitForFirstConsumer binding mode. Declaring a class as **default** lets PVCs without an explicit class automatically receive it.*

🔧 Container Storage Interface (CSI)

> The **CSI** provides a vendor-agnostic plug-in model for storage, replacing the older *in-tree* drivers baked into the kubelet.

- Vendors ship CSI drivers as DaemonSets (node side) and controllers (control-plane side).
- CSI decouples storage logic from the Kubernetes core, allowing independent updates of storage software.



The diagram visualizes the interaction between the kubelet, CSI controller, node plugin, and the container runtime during pod startup.

🗂️ CSI Architecture & Services

Service Category	Purpose	Typical RPC Methods
Identity	Self-identifies the driver (metadata, version) so the control plane knows it is available.	GetPluginInfo, GetPluginCapabilities
Controller	Handles volume lifecycle: create, delete, attach, detach.	CreateVolume, DeleteVolume, ControllerPublishVolume
Node	Executes node-local actions such as mounting and unmounting volumes.	NodeStageVolume, NodePublishVolume, NodeUnstageVolume

These services communicate over **gRPC** sockets (e.g., /plugin/csi.sock) that the kubelet contacts.

```
// Simplified CSI service definitions (highlighted for clarity)
service Identity {
    rpc GetPluginInfo (GetPluginInfoRequest) returns (GetPluginInfoResponse) {}
}
service Controller {
    rpc CreateVolume (CreateVolumeRequest) returns (CreateVolumeResponse) {}
}
service Node {
    rpc NodePublishVolume (NodePublishVolumeRequest) returns (NodePublishVolumeResponse) {}
}
```

```
service Identity {
  rpc GetPluginInfo(GetPluginInfoRequest)
  rpc GetPluginCapabilities(GetPluginCapabilitiesRequest)
  rpc Probe (ProbeRequest)
}

service Controller {
  rpc CreateVolume (CreateVolumeRequest)
  rpc DeleteVolume (DeleteVolumeRequest)
  rpc ControllerPublishVolume (ControllerPublishVolumeRequest)
}

service Node {
  rpc NodeStageVolume (NodeStageVolumeRequest)
  rpc NodeUnstageVolume (NodeUnstageVolumeRequest)
  rpc NodePublishVolume (NodePublishVolumeRequest)
  rpc NodeUnpublishVolume (NodeUnpublishVolumeRequest)
  rpc NodeGetInfo (NodeGetInfoRequest)
  ...
}
```

The Identity service tells Kubernetes what type of volumes can be created by the controllers running in a cluster.

The Create and Delete methods are called before a node can mount a volume into a Pod, implementing dynamic storage.

The Node service is the part of CSI that runs on a kubelet, mounting the volume created previously into a specific Pod on demand.

The snippet shows the three CSI services and their key RPCs, illustrating how a storage vendor implements the specification.

🛠️ CSI Driver Workflow

1. **Register driver** – The driver advertises its name and capabilities via the **Identity** service.
2. **CreateVolume** – The controller (external provisioner) receives a PVC event, calls CreateVolume on the driver, and receives a volume ID.
3. **PublishVolume** – The scheduler binds the PVC to a node; the node plugin receives a NodePublishVolume request and performs the bind mount.
4. **Mount** – The kubelet mounts the volume into the pod's filesystem namespace.

> **Bind mount** is the Linux operation that mirrors a directory into another location, enabling the pod to access the provisioned storage.

📊 Storage Types Comparison

Storage Type	Lifespan	Durability	Typical Use-Case
Local	Persistent on the node	High (disk stays)	Heavy-weight, data-intensive apps that need node-local fast storage
emptyDir	Exists while the pod runs on the node	None (lost on pod termination)	Scratch space, fast temporary storage, shared logs
PVC (dynamic)	Persistent across nodes (depending on class)	Configurable (Retain/Delete)	General purpose storage, stateful workloads, databases

EmptyDir is cheap and fast (can be tmpfs), but unsuitable for data that must survive pod restarts.

📁 PersistentVolume & PVC Lifecycle

1. **Pod request** – A Pod manifest includes a volumeClaimTemplates (StatefulSet) or a direct PVC reference.
2. **Scheduler** – Finds a node meeting the storage topology.
3. **PVC creation** – The control plane watches PVC objects; the appropriate controller creates a PV via the CSI driver.
4. **Binding** – The PV is bound to the PVC; the scheduler now considers the pod runnable.
5. **Kubelet mounts** – Using CSI node services, the volume is mounted into the pod's container.
6. **Pod runs** – Containers read/write to the mounted volume.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mydata
spec:
  storageClassName: fast
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

🧩 StorageClasses & Binding Modes

- **StorageClass** abstracts the provisioner details (e.g., rancher.io/local-path, pd-ssd).
- **BindingMode** determines when the volume is provisioned:
  - Immediate – volume created as soon as PVC appears.
  - WaitForFirstConsumer – provisioning deferred until the pod is scheduled, improving data locality.

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: rancher.io/local-path
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer
parameters:
  type: pd-ssd
```

Admins can define multiple classes (e.g., bigdata, postgres, ai) to match workload performance and retention requirements.

🗄️ HostPath & CNI Integration

- **hostPath** mounts a directory from the node into a pod. Useful for system-level tasks (e.g., CNI plugins installing binaries).
- Many CNI and CSI drivers rely on hostPath for privileged operations such as writing binaries to /opt/cni/bin or exposing host sockets.

```yaml
volumes:
- name: cni-bin
  hostPath:
    path: /opt/cni/bin
    type: DirectoryOrCreate
```

> **Caution:** Using hostPath is often considered an anti-pattern for application workloads because it ties pods to specific nodes.

🪟 CSI on Non-Linux OS

- CSI is OS-agnostic; on Windows, the **csi-proxy** project mediates between the kubelet and Windows-specific storage APIs (PowerShell, etc.).
- As Windows support matures, CSI drivers can run as DaemonSets on Windows nodes, similar to Linux deployments.

📚 Summary

- **Dynamic provisioning** + **CSI** = fully automated, vendor-agnostic storage that scales with workloads.

- **StorageClasses** let administrators expose multiple storage profiles (performance, retention, locality) without changing application YAML.
- **emptyDir** provides fast, transient storage; **local** offers node-bound durability; **PVC** (via CSI) delivers persistent, portable volumes.
- The **CSI** model separates **identity**, **controller**, and **node** services, simplifying driver development and enabling hot-swap of storage back-ends.
- **hostPath** remains a powerful tool for system-level integrations (CNI/CSI bootstrapping) but should be avoided for regular app data.
- CSI works across Linux and Windows, with platform-specific helpers (e.g., csi-proxy) handling OS differences.

These notes can be combined with other sections to form a complete Kubernetes storage study guide.## 🌐 DNS & Pod Networking

> **DNS resolution** in a cluster links hostnames to IPs, which are then routed by the network proxy or CNI.

- **Service IP** → network proxy forwards to a pod endpoint.
- **Pod IP** → CNI makes the IP directly routable.
- **External IP** → outbound traffic is NAT-ed via iptables so responses return to the originating pod.

/etc/resolv.conf inside a pod (example):

```
search default.svc.cluster.local svc.cluster.local cluster.local
nameserver 10.96.0.10
options ndots:5
```

- The **search** list appends domains until a query succeeds.
- **Nameserver 10.96.0.10** is the cluster-internal **kube-dns/CoreDNS** service.

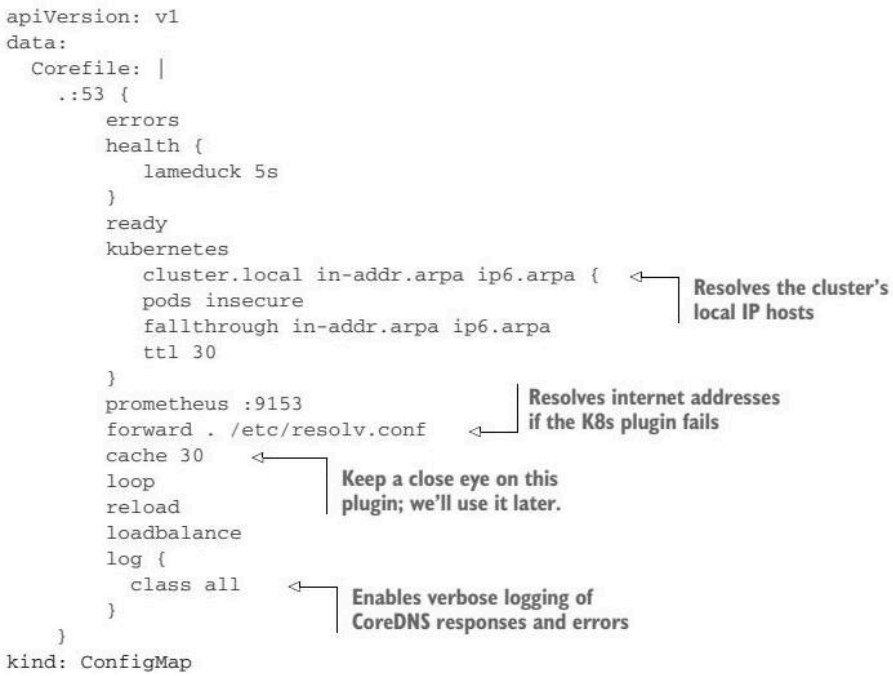
Example DNS query (from an NGINX pod):

```
# wget web-ss-0.nginx.default.svc.cluster.local
Connecting to web-ss-0.nginx.default.svc.cluster.local (10.244.0.13:80)
```

🧩 CoreDNS – Cluster-First Resolver

> CoreDNS processes DNS queries through a chain of **plugins**, each on its own line in the ConfigMap.

```
kubectl get cm coredns -n kube-system -o yaml
```



The image shows a CoreDNS ConfigMap with plugins for health checks, Kubernetes integration, and logging.

- **Kubernetes plugin** resolves cluster-local names.
- If it fails, the **forward** plugin uses the host's resolv.conf to reach the internet.

Pod DNS policy

dnsPolicy	Effect
ClusterFirst (default)	Pods use CoreDNS as primary resolver.
Default	Pods inherit the node's /etc/resolv.conf.

⚙️ Tuning CoreDNS Cache

The **cache** plugin stores results (default 30 s). Long cache times can delay DNS updates after scaling events.

Fix:

```
kubectl edit cm coredns -n kube-system   # set cache TTL to 5s
```

Parameter	Default	Recommended (small clusters)
cache TTL	30 s	5 s (faster refresh)

🏗️ Control Plane Overview (Kind)

> **Kind** runs a full control plane inside Docker containers.

```
kind create cluster
kubectl -n kube-system get po -o custom-columns=":metadata.name"
```

```
$ docker exec -it \
$(docker ps | grep kind | awk '{print $1}') \
/bin/bash
root@kind-control-plane:/$ ps aux | grep \
"/usr/bin/kubelet"
root     722 11.7  3.5 1272784 71896 ?     Ssl  23:34
➥ 1:10 /usr/bin/kubelet
➥ --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf
➥ --kubeconfig=/etc/kubernetes/kubelet.conf
➥ --config=/var/lib/kubelet/config.yaml --container-runtime=remote
➥ --container-runtime-endpoint=/run/containerd/containerd.sock
➥ --fail-swap-on=false --node-ip=172.17.0.2
➥ --fail-swap-on=false
```

Gets an interactive terminal running inside the kind container

ps (process status) for the kubelet

The running kubelet process

The screenshot lists core control-plane pods (coredns, etcd, kube-apiserver, etc.).

Key components:

- **etcd** – strongly consistent key/value store.
- **kube-apiserver** – RESTful front-end.
- **kube-controller-manager** – runs controllers (node, replication, endpoints).
- **kube-scheduler** – assigns pods to nodes.
- **kube-proxy** – service networking.

📦 API Server & Resources

> The API server exposes built-in and custom resources.

```
kubectl api-resources -o name | head -n 20
```

```
$ kubectl api-resources -o name | head -n 20
bindings
componentstatuses
configmaps
endpoints
events
limitranges
namespaces
nodes
persistentvolumeclaims
persistentvolumes
pods
podtemplates
replicationcontrollers
resourcequotas
secrets
serviceaccounts
services
mutatingwebhookconfigurations.admissionregistration.k8s.io
validatingwebhookconfigurations.admissionregistration.k8s.io
customresourcedefinitions.apiextensions.k8s.io
```

Shows the available APIs, just looking at the first 20 using head

List of core and custom API groups (e.g., bindings, configmaps, admissionregistration.k8s.io).

API versioning

Level	Stability	Typical use
Alpha	Experimental, not for production	Early feature testing
Beta	Production-ready, guaranteed support	Widely used (e.g., DaemonSet)

GA	Fully stable	Default in clusters

📚 Custom Resource Definitions (CRDs)

- Enable developers to add **own API objects** without modifying the core API server.
- Applied with kubectl apply -f <CRD-yaml>.
- Used by operators, Istio, Envoy, etc., to extend cluster functionality.

📈 Scheduler Plugin Framework

> The scheduler reads a **Plugins** struct; each phase runs a specific set of plugins.

```
type Plugins struct {
    QueueSort   *PluginSet
    PreFilter   *PluginSet
    Filter      *PluginSet
    PostFilter  *PluginSet
    PreScore    *PluginSet
    Score       *PluginSet
    Reserve     *PluginSet
    Permit      *PluginSet
    PreBind     *PluginSet
    Bind        *PluginSet
    PostBind    *PluginSet
}
```

```
PreScore: &schedulerapi.PluginSet{
  Enabled: []schedulerapi.Plugin{
    {Name: interpodaffinity.Name},
    {Name: podtopologyspread.Name},
    {Name: tainttoleration.Name},
  },
},
```

All the defined plugins have already run during the filtering process.

Snippet shows PreScore plugin list; comment notes that preceding plugins have already run.

Key phases (simplified)

1. **QueueSort** – orders pending pods.
2. **PreFilter** – prepares node data (e.g., resource fit).
3. **Filter** – discards unschedulable nodes (taints, affinity, volume limits).
4. **Score** – ranks remaining nodes (balanced resources, image locality).
5. **Bind** – writes the binding to the API server.

Scoring example (CPU & memory weighting):

$$ \text{Score} = \frac{\text{cpu_fraction} + \text{memory_fraction}}{\text{weightSum}} $$

🛠️ Controller Manager & Storage

- **Node controller** watches node health and updates the Node API object.
- **Replication controller** (now **ReplicaSet**) maintains desired pod counts.
- **Endpoint controller** populates Endpoints objects for services.

Headless Service (no ClusterIP) returns pod IPs directly; useful for StatefulSets.

🔐 Service Accounts & Tokens

- Every namespace gets a **default ServiceAccount**.
- Pods inherit it unless serviceAccountName is specified.
- Token is auto-mounted; disable with automountServiceAccountToken: false.

☁️ Cloud Controller Manager (CCM)

> CCM abstracts cloud-specific actions (nodes, routes, load balancers).

```
type Interface interface {
    Initialize(clientBuilder ControllerClientBuilder, stop <-chan struct{})
    LoadBalancer() (LoadBalancer, bool)
    Instances() (Instances, bool)
    // … other cloud-specific methods …
}
```

- Implements three main controllers (node, route, service).
- Runs as a single binary but may spawn additional control loops.

📚 etcd – The Consistent Data Store

> etcd provides **strong consistency** via the **Raft** consensus algorithm.

Raft basics (simplified)

1. One **leader** and multiple **followers** (odd number of nodes).
2. Client writes → leader → replicates to majority.
3. Once a majority acknowledges, the write is committed.
4. If the leader fails, followers elect a new leader.

Write-ahead log (WAL)

- Guarantees durability: a write returns only after **fsync** to disk.
- Performance tip: **fsync ≤ 250 ms** on SSDs; > 1 s indicates trouble.

Performance histogram example

le (seconds)	Count
0.001	1239
0.002	2365
…	…
1.024	2588
+Inf	2588

> Most writes complete well under **0.008 s** in a healthy cluster.

Size limits

Setting	Default
Max request size	**1.5 MiB** (--max-request-bytes)
DB size recommendation	**≤ 8 GiB** (typical < 2 GiB)
Memory for large clusters	**≈ 64 GiB** (to hold watches)

Encryption at rest

- API server flag --encryption-provider-config points to a YAML defining providers (AES-GCM, AES-CBC, SecretBox).
- First provider encrypts; all providers are tried for decryption.

🔧 etcd Operations

```
# Health check (embedded perf test)
etcdctl --endpoints=https://127.0.0.1:2379 \
        --cacert=/etc/kubernetes/pki/etcd/ca.crt \
        --cert=/etc/kubernetes/pki/etcd/server.crt \
        --key=/etc/kubernetes/pki/etcd/server.key \
        check perf
```

- **Throughput** ≈ 150 writes/s, **slowest** ≈ 0.2 s on a small kind cluster.

Running etcd in a pod for debugging

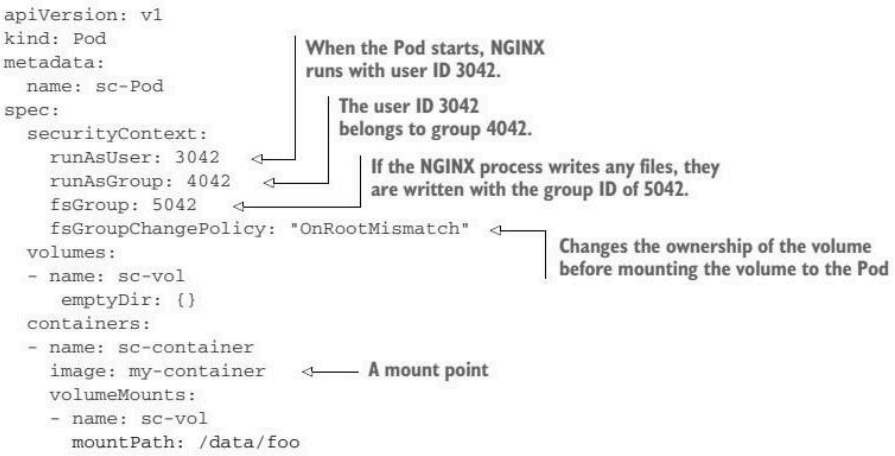
```
apiVersion: v1
kind: Pod
metadata:
  name: etcdclient
  namespace: kube-system
spec:
  hostNetwork: true
  containers:
  - name: etcdclient
    image: ubuntu
    command: ["sleep","6000"]
    env:
    - name: ETCDCTL_API
      value: "3"
    - name: ETCDCTL_ENDPOINTS
      value: "https://127.0.0.1:2379"
    volumeMounts:
    - name: etcd-certs
      mountPath: /etc/kubernetes/pki/etcd
  volumes:
  - name: etcd-certs
    hostPath:
      path: /etc/kubernetes/pki/etcd
```

🔐 Container & Pod Security

Best practices

1. **Never run containers as root** – define a non-root user (runAsUser).
2. **Use minimal base images** (e.g., Google Distroless).
3. **Scan images** for CVEs (e.g., Trivy, Clair).
4. **Apply linters** (hadolint, shellcheck) in CI pipelines.

SecurityContext example

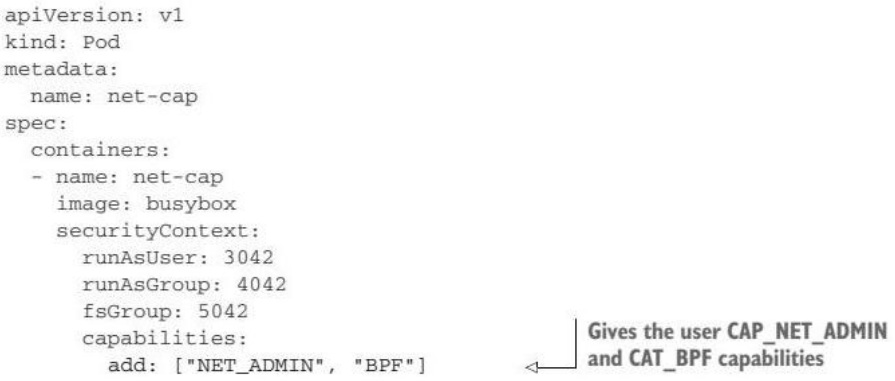



```
apiVersion: v1
kind: Pod
metadata:
  name: sc-Pod
spec:
  securityContext:
    runAsUser: 3042
    runAsGroup: 4042
    fsGroup: 5042
    fsGroupChangePolicy: "OnRootMismatch"
  volumes:
  - name: sc-vol
     emptyDir: {}
  containers:
  - name: sc-container
    image: my-container
    volumeMounts:
    - name: sc-vol
      mountPath: /data/foo
```

The snippet sets runAsUser: 3042, runAsGroup: 4042, fsGroup: 5042, and mounts an emptyDir volume.

```
spec:
  securityContext:
    runAsUser: 3042
    runAsGroup: 4042
    fsGroup: 5042
```

Adding Linux capabilities




```
apiVersion: v1
kind: Pod
metadata:
  name: net-cap
spec:
  containers:
  - name: net-cap
    image: busybox
    securityContext:
      runAsUser: 3042
      runAsGroup: 4042
      fsGroup: 5042
      capabilities:
        add: ["NET_ADMIN", "BPF"]
```

Pod adds NET_ADMIN and BPF capabilities without granting full root.

```
securityContext:
  capabilities:
    add: ["NET_ADMIN", "BPF"]
```

Pod Security Policies (PSP) *(deprecated → Pod Security Admission)*

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restricted
spec:
  privileged: false
  runAsUser:
    rule: MustRunAsNonRoot
```

```
  seLinux:
    rule: RunAsAny
  volumes:
    - 'configMap'
    - 'secret'
```

- Enforced at admission time; blocks non-compliant pods.

ServiceAccount token automount

```
apiVersion: v1
kind: Pod
metadata:
  name: no-sa
spec:
  automountServiceAccountToken: false
```

- Disables the default token for pods that don't need API access.

📊 Summary Tables

CoreDNS Cache Impact

Action	Expected DNS latency	Observation with 30 s cache
Scale-down → Scale-up StatefulSet	< 1 s	May experience several-second stalls until cache expires.
Reduce TTL to 5 s	< 1 s	Faster DNS refresh, lower stale-record risk.

etcd Performance Checklist

Check	Command	Success criteria
Disk latency	` curl localhost:2381/metrics	grep fsync `	
Leader health	etcdctl endpoint status --write-out=table	One leader, others follower
Encryption	Verify --encryption-provider-config present	Secrets stored encrypted at rest

---

📚 Key Takeaways

- **DNS** is tightly coupled with pod networking; resolv.conf points to CoreDNS (10.96.0.10).
- **CoreDNS** uses plugins; tuning cache TTL improves DNS freshness.
- The **control plane** (etcd, apiserver, scheduler, controller-manager) can be inspected with kind.
- **API resources** are versioned (alpha → beta → GA); CRDs extend the API without core changes.
- **Scheduler** operates via a plugin pipeline (QueueSort → Filter → Score → Bind).
- **etcd** provides strict consistency via Raft; monitor fsync latency and leader elections.
- **Security** starts at the container: avoid root, use minimal images, scan for CVEs, apply security contexts, limit capabilities, and disable unnecessary service-account tokens.

These notes can be combined with other sections to build a complete Kubernetes study guide.## 🔐 TLS Cipher Suite Management

- **Protocol**: Transport Layer Security (**TLS**)
- **Key exchange**: Elliptic Curve Diffie-Hellman Ephemeral (**ECDHE**)
- **Authentication**: Elliptic Curve Digital Signature Algorithm (**ECDSA**)
- **Encryption**: AES-256 in CBC mode

> **Note:** Monitor your TLS posture to ensure it aligns with organizational standards.

- Update cipher suites on any Kubernetes service via the --tls-cipher-suites flag, e.g.:

```
--tls-cipher-suites=TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA,TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
```

- Apply the same change to **API server**, **scheduler**, **controller-manager**, and **kubelet** if a suite is vulnerable.
- Avoid over-restricting suites; some clients (e.g., AWS ELB) may not support newer elliptic-curve ciphers. Consider **TCP health checks** instead of HTTPS when using custom cipher suites.

🖥️ Immutable OS vs. Patching Nodes

- **Immutable OS**: All binaries are read-only; nodes are replaced rather than patched.
- Benefits:
  - Eliminates on-disk modifications → reduces attack surface.
  - Simplifies node replacement & version upgrades.
- Kubernetes components on a **control-plane node**:

Component	Present
kubelet binary	✓
kube-proxy binary	✓
Container runtime (containerd)	✓
etcd image	✓

- **Worker node**: Same as above **except** etcd (not supported on Windows).
- For Windows workers, build custom immutable images because the OS is not redistributable.
- See **Tanzu Community Edition** for a ready-made immutable-image workflow.

🛡️ Isolated Container Runtimes

- **Docker Engine** uses namespaces but still shares the host kernel → vulnerable if the kernel is compromised.
- Projects providing a **virtual kernel sandbox**:

Project	Status	Used by
gVisor	Mature	Google Cloud
Firecracker	Mature	AWS
Kata	Mature	Various
IBM Nabla	Emerging	–

- These runtimes run containers on a **micro-VM** or **virtualized kernel**, offering stronger isolation.

⚡ Resource Attacks & Limits

Why limits matter

- Pods share finite CPU, memory, and disk on a node.
- Unrestricted containers can cause a **Denial-of-Service (DoS)** on the node.

Resource limit YAML example

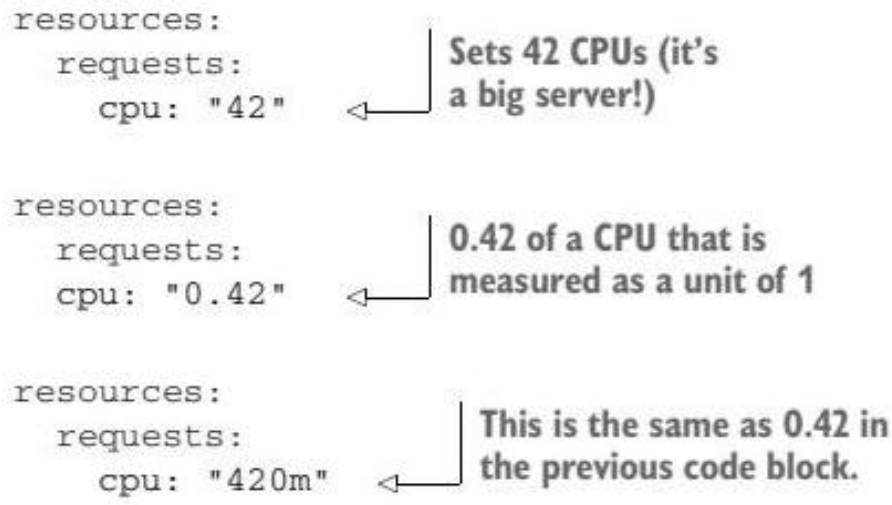
```
apiVersion: v1
kind: Pod
metadata:
  name: core-kube-limited
spec:
  containers:
  - name: app
    image: my-image
    resources:
      requests:
        memory: "42Mi"
        cpu: "42m"
        ephemeral-storage: "21Gi"
      limits:
        memory: "128Mi"
        cpu: "84m"
        ephemeral-storage: "42Gi"
```

- **Requests** → minimum resources the scheduler reserves.
- **Limits** → maximum resources; exceeding triggers a restart, then termination if repeated.

🔪 CPU Units

Kubernetes treats **1 CPU** as one hyper-thread (bare metal) or one vCPU (cloud).

Notation	Meaning
1	1 full CPU core
0.25	¼ core
250m	250 millicores (same as 0.25)



The image illustrates three CPU request styles—42 CPUs (big server), 0.42 CPUs, and 420m—showing how the same capacity can be expressed differently.

📚 Memory Units

Memory can be specified using decimal or binary suffixes:

Suffix	Decimal	Binary
K / Ki	10^3 bytes	2^{10} bytes
M / Mi	10^6 bytes	2^{20} bytes
G / Gi	10^9 bytes	2^{30} bytes
…	…	…

Examples (all ~129 MB):

```
resources:
  requests:
    memory: "128974848"      # plain bytes
    memory: "129e6"          # scientific notation
    memory: "129M"           # megabits ≈ 1.613 × 10⁷ B
    memory: "123Mi"          # mebibytes
```

💾 Ephemeral Storage

- Applies to **emptyDir** (except tmpfs), node-level log directories, and writable container layers.
- Exceeding limits → **kubelet evicts** the Pod.

Extended resources (e.g., GPUs) have separate limits defined in the Kubernetes docs.

🌐 Host Networks vs. Pod Networks

- **Pod network (default)** → isolates Pods from host ports; recommended for security.
- **Host network** (hostNetwork: true) → Pod shares the node's network namespace → larger **blast radius**.

```
apiVersion: v1
kind: Pod
metadata:
  name: host-Pod
spec:
  hostNetwork: true
```

> **Tip:** Use host networking only for system-level Pods (e.g., CNI agents), never expose them directly to the Internet.

📦 Full-Featured Pod Example

```
apiVersion: v1
kind: Pod
metadata:
  name: example-Pod
spec:
  automountServiceAccountToken: false
  securityContext:
    runAsUser: 3042
    runAsGroup: 4042
    fsGroup: 5042
    capabilities:
      add: ["NET_ADMIN"]
  hostNetwork: true
  volumes:
  - name: sc-vol
```

```
    emptyDir: {}
  containers:
  - name: sc-container
    image: my-container
    resources:
      requests:
        memory: "42Mi"
        cpu: "42m"
        ephemeral-storage: "1Gi"
      limits:
        memory: "128Mi"
        cpu: "84m"
        ephemeral-storage: "2Gi"
    volumeMounts:
    - name: sc-vol
      mountPath: /data/foo
  serviceAccountName: network-sa
```

- Disables default service-account token.
- Sets explicit UID/GID and adds NET_ADMIN.
- Runs on the host network with resource limits.

🔐 API Server Security & RBAC

Admission Webhooks

- **ImagePolicyWebhook**: validates container images before admission.
- Pods failing admission stay **Pending**.

Role-Based Access Control (RBAC)

- Enabled via --authorization-mode=RBAC.
- Core RBAC objects:

Object	Scope
Role	Namespace-limited
ClusterRole	Cluster-wide
RoleBinding	Grants a Role to users/groups/service-accounts
ClusterRoleBinding	Grants a ClusterRole globally

Verbs (allowed actions)

- API resource verbs: get, list, create, update, patch, watch, proxy, redirect, delete, deletecollection
- Non-resource HTTP verbs: get, post, put, delete

Example Role & Binding (list & patch Pods)

```
# Role
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: Pod-labeler
  namespace: rbac-example
rules:
- apiGroups: [""]
  resources: ["Pods"]
  verbs: ["list", "patch"]
---
# ServiceAccount
apiVersion: v1
```

```
kind: ServiceAccount
metadata:
  name: Pod-labeler
  namespace: rbac-example
---
# RoleBinding
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: Pod-labeler
  namespace: rbac-example
subjects:
- kind: ServiceAccount
  name: Pod-labeler
roleRef:
  kind: Role
  name: Pod-labeler
  apiGroup: rbac.authorization.k8s.io
```

Subresources (e.g., Pod logs)

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: Pod-and-Pod-logs-reader
  namespace: rbac-example
rules:
- apiGroups: [""]
  resources: ["Pods", "Pods/log"]
  verbs: ["get", "list"]
```

Subjects can include Users, Groups, ServiceAccounts.

🐞 Debugging RBAC

- **Audit logs** capture *who, what, where, how* for every API request.
- Enable via --audit-policy-file on the API server.
- Helpful tools:

Tool	Purpose
rbac-lookup (ReactiveOps)	List roles for a subject
kubectl-rbac (OctarineSec)	Show permissions for a subject
audit2rbac (Jordan Liggit)	Generate RBAC from audit logs

🔑 Authn, Authz, and Secrets

- Do **not** use default admin certificates.
- Prefer **IAM service accounts** for cloud-native identity.
- Disable basic username/password auth; rely on TLS client certificates.

IAM Service Accounts (cloud)

- Every node gets an IAM role; Pods inherit it unless overridden.
- **Least-privilege**: create dedicated roles for each workload.
- Tools for per-Pod IAM isolation: kube2iam, kiam.

Private API Servers

- Place the API server behind a **private network** or bastion host.
- Whitelist load-balancer IPs; avoid exposing the API server to the Internet.

🌐 Network Security

NetworkPolicies (requires a CNI that supports them, e.g., Calico)

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-ingress
  namespace: web
spec:
  podSelector: {}
  policyTypes: ["Ingress"]
```

- Denies all inbound traffic to pods in web.

Allow traffic from a specific namespace

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: access-web
  namespace: web
spec:
  podSelector:
    matchLabels:
      app: nginx
  policyTypes: ["Ingress"]
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          name: test-bed
```

- **Label namespaces** (kubectl label namespaces test-bed name=test-bed) to make the selector work.

Default deny-all (both ingress & egress)

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all
  namespace: test-bed
spec:
  podSelector: {}
  policyTypes: ["Ingress", "Egress"]
```

- After applying, pods lose DNS access; add an egress rule for DNS if needed:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: dns-egress
  namespace: test-bed
spec:
  podSelector: {}
  policyTypes: ["Egress"]
```

```
  egress:
  - to:
    - namespaceSelector:
        matchLabels:
          name: kube-system
    ports:
    - protocol: UDP
      port: 53
```

🛠️ Service Mesh (brief)

- Provides **mutual TLS** and advanced traffic policies.
- Common meshes: **Istio**, **Linkerd**, **Consul**.
- Not required for day-one clusters; consider only if you need its capabilities.

📜 Open Policy Agent (OPA)

- Declarative policy engine usable as an **admission controller** or sidecar.
- Two main components:

Component	Role
OPA Gatekeeper	CRD-based policies, audit capabilities
OPA admission controller	Direct webhook enforcement

Installing OPA Gatekeeper (kind example)

```
kind delete cluster
kind create cluster
kubectl apply -f https://raw.githubusercontent.com/open-policy-agent/gatekeeper/v3.7.0/deploy/gatekeeper.yaml
```

```
kubectl -n gatekeeper-system get po
# Expected output: gatekeeper-audit-..., gatekeeper-controller-manager-...
```

Sample ConstraintTemplate (enforce specific container registry)

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: enforcespecificcontainerregistry
spec:
  crd:
    spec:
      names:
        kind: EnforceSpecificContainerRegistry
  targets:
  - target: admission.k8s.gatekeeper.sh
    rego: |
      package enforcespecificcontainerregistry
      violation[{"msg": msg}] {
        container := input.review.object.spec.containers[_]
        satisfied := [good |
          repo = input.parameters.repos[_];
          good = startswith(container.image, repo)
        ]
        not any(satisfied)
        msg := sprintf("container ‘%v' has an invalid image repo ‘%v', allowed repos are %v",
```

```
                    [container.name, container.image, input.parameters.repos])
      }
```

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: EnforceSpecificContainerRegistry
metadata:
  name: enforcespecificcontainerregistrytestns
spec:
  match:
    kinds: [{apiGroups: [""], kinds: ["Pod"]}]
    namespaces: ["test-ns"]
  parameters:
    repos: ["myrepo.com"]
```

- Attempting to create a pod with an image outside myrepo.com will be denied.

🏢 Multi-Tenancy

Trust Level	Typical Use-Case
High trust	Different departments of the same organization sharing a cluster.
Medium/Low trust	External customers in separate namespaces.
Zero trust	Data subject to legal constraints; require strict isolation (often separate clusters).

- Kubernetes namespace is the primary isolation boundary, but it does not prevent cross-namespace API access.
- For strong isolation, consider separate clusters, RBAC, NetworkPolicies, and possibly a service mesh.

📋 Kubernetes Tips (quick checklist)

- ✅ Private API server endpoint (no internet exposure).
- ✅ Enable RBAC.
- ✅ Apply NetworkPolicies.
- ✅ Disable username/password auth.
- ✅ Use dedicated service accounts; disable default token mounting.
- ✅ Avoid hostNetwork unless required.
- ✅ Define resource quotas and per-Pod limits.

📦 Application Management – Guestbook Example

Installing the classic Guestbook (single YAML)

```
kubectl create -f https://github.com/kubernetes/examples/blob/master/guestbook/all-in-one/guestbook-all-in-on
```

- Caution: Curling YAML directly from the internet can be risky; prefer trusted repos.

Challenges with monolithic YAML

- Hard to customize, version, or scale.
- Repeating large blocks leads to maintenance overhead.

Modularizing with Carvel tools (ytt, kapp, kapp-controller)

Install Carvel toolkit

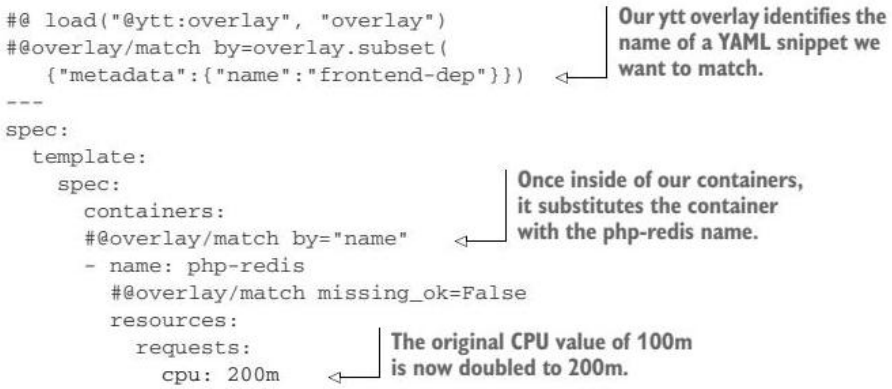
```
# macOS
brew tap vmware-tanzu/carvel && brew install ytt kbld kapp imgpkg kwt vendir
# Linux
curl -L https://carvel.dev/install.sh | bash
```

Split resources into separate files (example directory layout)

```
v1/
  fe-dep.yaml
  fe-svc.yaml
  redis-master-dep.yaml
  redis-master-svc.yaml
  redis-slave-dep.yaml
  redis-slave-svc.yaml
```

Patch CPU requests with ytt overlays

> **Image 2** shows a ytt overlay that matches the frontend-dep pod and updates its CPU request to 200m.



The overlay matches metadata.name: frontend-dep and substitutes the container's CPU request.

```
# v2/ytt-cpu-overlay.yml
#@ load("@ytt:overlay", "overlay")
#@overlay/match by=overlay.subset({"metadata":{"name":"frontend-dep"}})
---
spec:
  template:
    spec:
      containers:
      #@overlay/match by="name"
      - name: php-redis
        resources:
          requests:
            cpu: 200m   # new request
```

- Apply overlays and deploy with **kapp**:

```
kapp deploy -a guestbook \
  -f <(ytt -f ./v1/ -f ./v2/ytt-cpu-overlay.yml -f ./v2/ytt-cpu-overlay-db.yml)
```

> **Image 3** demonstrates the kapp deploy command used to install the Guestbook app.

```
$ kapp deploy -a kc -f https://github.com/vmware-tanzu/
      carvel-kapp-controller/releases/latest/
        download/release.yml
$ kapp deploy -a default-ns-rbac -f
    https://raw.githubusercontent.com/vmware-tanzu/
      carvel-kapp-controller/
       develop/examples/rbac/default-ns.yml
```

Installs kapp-controller using the kapp tool

Installs a simple RBAC definition

The image shows the two-step kapp deployment workflow.

- **Inspect** the deployed app:

```
kapp list
kapp inspect --app=guestbook
```

- **Delete** atomically:

```
kapp delete --app=guestbook
```

Managing the Guestbook with kapp-controller (Operator-style)

```
apiVersion: kappctrl.k14s.io/v1alpha1
kind: App
metadata:
  name: guestbook
  namespace: default
spec:
  serviceAccountName: default-ns-sa
  fetch:
  - git:
      url: https://github.com/jayunit100/k8sprototypes
      ref: origin/master
      subPath: carvel-guestbook/v2/output/
  template:
  - ytt: {}
  deploy:
  - kapp: {}
```

> **Image 4** shows the YAML snippet for a kapp App CR.

```
apiVersion: kappctrl.k14s.io/v1alpha1
kind: App
metadata:
  name: guestbook
  namespace: default
spec:
  serviceAccountName: default-ns-sa
  fetch:
  - git:
      url: https://github.com/jayunit100/k8sprototypes
      ref: origin/master

      # We have a directory, named 'app', in the root of our repo.
      # Files describing the app (i.e. pod, service) are in that directory.
      subPath: carvel-guestbook/v1/
  template:
  - ytt: {}
  deploy:
  - kapp: {}
```

Uses the service account created previously for this app installation

Specifies where our app is defined

Because the code for our app actually lives in carvel-guestbook/v1/, we need to specify this subpath.

The CR defines where the Git repo lives, which templates to render, and that kapp should perform the deployment.

- Install **kapp-controller**:

```
kapp deploy -a kc -f https://github.com/vmware-tanzu/carvel-kapp-controller/releases/latest/download/release.
kapp deploy -a default-ns-rbac -f https://raw.githubusercontent.com/vmware-tanzu/carvel-kapp-controller/devel
```

- Create the Guestbook App CR (kubectl create -f guestbook-app.yaml).
- The controller watches the CR, renders YTT templates, and uses Kapp to reconcile the resources—providing **declarative, state-ful** application management.

📊 Summary Tables

Resource Specification Units

Resource	Decimal Form	Millicore / Binary Form
CPU	0.5	500m
Memory	256Mi	268435456 bytes
Ephemeral Storage	10Gi	10737418240 bytes

Common Security Controls

Control	Scope	Typical Implementation
TLS Cipher Suites	API Server, Scheduler, Controller-Manager, Kubelet	--tls-cipher-suites flag
Immutable OS	Node OS	Replace VM/image instead of patching
Isolated Runtimes	Container runtime	gVisor, Firecracker, Kata
Resource Limits	Pods	resources.requests / resources.limits
RBAC	API Access	Roles, ClusterRoles, Bindings
NetworkPolicies	Pod ↔ Pod traffic	Calico, Antrea, Cilium
OPA/Gatekeeper	Admission control	ConstraintTemplates + Constraints
IAM Service Accounts	Cloud API access	Per-Pod roles via kube2iam / kiam

---

📚 Further Reading

- **Calico CRD list** (example output)

```
kubectl get crd | grep calico
#@ Output includes bgpconfigurations.crd.projectcalico.org, networkpolicies.crd.projectcalico.org, et
```

> **Image 5** displays a fragment of a Calico configuration spec, illustrating the depth of CRD-based configuration.

```
...
spec:
  controllers:
    namespace:
      reconcilerPeriod: 5m0s
    node:
      reconcilerPeriod: 5m0s
      syncLabels: Enabled
    policy:
      reconcilerPeriod: 5m0s
    serviceAccount:
      reconcilerPeriod: 5m0s
    workloadEndpoint:
      reconcilerPeriod: 5m0s
  etcdV3CompactionPeriod: 10m0s
  healthChecks: Enabled
  logSeverityScreen: Info
  prometheusMetricsPort: 9094
status:
  environmentVars:
    DATASTORE_TYPE: kubernetes
    ENABLED_CONTROLLERS: node
  runningConfig:
    controllers:
      node:
        hostEndpoint:
          autoCreate: Disabled
        syncLabels: Disabled
    etcdV3CompactionPeriod: 10m0s
    healthChecks: Enabled
    logSeverityScreen: Info
```

We can disable healthChecks if we don't feel it's necessary.

Sets the port on which our Calico kube controller serves up its Prometheus metrics. Here's where we can change the port if needed.

The snippet shows fields such as reconciler period, log severity, and Prometheus metrics port.

- Explore **Tanzu Community Edition** for a production-grade, immutable-image Kubernetes distribution.
- Test **NetworkPolicy conformance** with Sonobuoy:

```
sonobuoy run --e2e-focus=NetworkPolicy
sonobuoy status
```
