Omar Latrach

TECHNICAL ARTICLE

PUBLISHED · SEPTEMBER 26, 2026

Worker Node Anatomy: kubelet, Container Runtime, and kube-proxy

Understand how kubelet, the container runtime, and kube-proxy cooperate on every worker node—and how to inspect each layer without confusing their responsibilities.

This is Learn post 16 of the 54-post Certified Kubernetes Administrator (CKA) preparation path. The previous lesson explained how the control plane records intent and assigns Pods. This lesson crosses the API boundary and follows that intent onto a worker node.

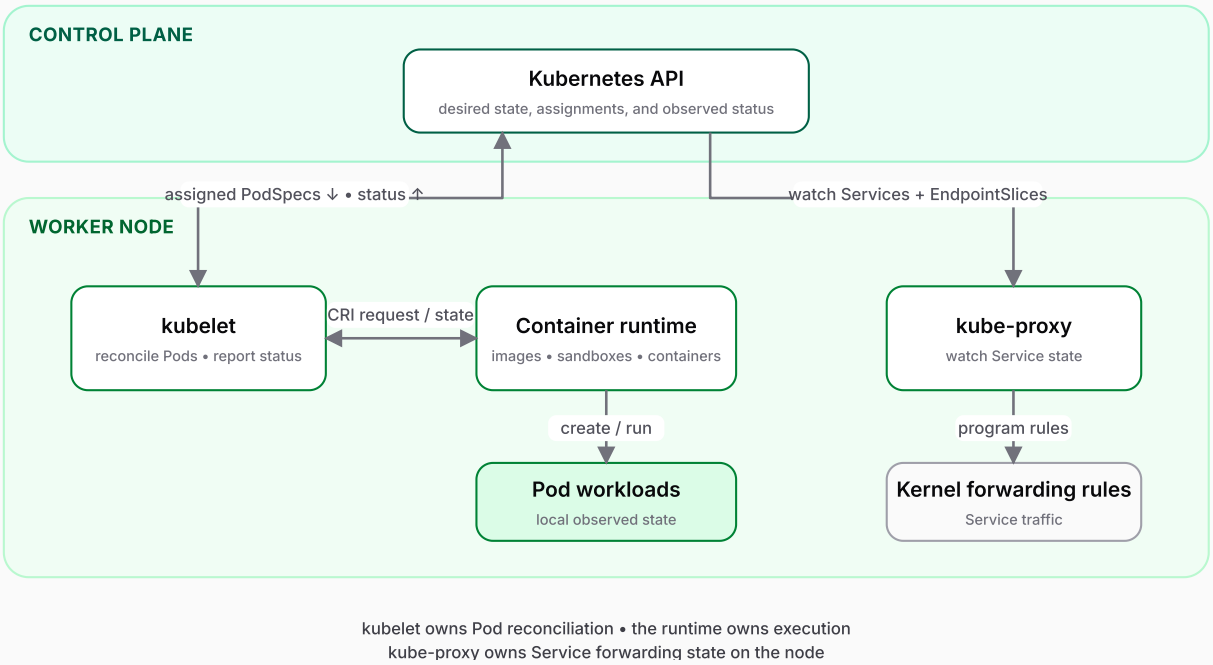
A worker node is not one Kubernetes process. It is a cooperating stack: kubelet reconciles assigned Pods, a Container Runtime Interface (CRI) implementation executes containers, and kube-proxy normally programs node-level rules for Service traffic. Knowing which layer owns which result is the foundation of efficient node administration.

What you'll learn

- Separate the responsibilities of kubelet, the container runtime, and kube-proxy.
- Trace an assigned Pod from its API object to a running container on one node.
- Read Node status, conditions, runtime information, and heartbeat Leases.
- Use kubectl, systemctl, journalctl, and crictl at the layer each tool can actually observe.
- Explain why healthy Pods do not prove that node-level Service routing is healthy.

The mental model: foreman, engine room, and traffic rules

> kubelet is the node's foreman: it receives Pod intent and checks the result. The container runtime is the engine room: it pulls images and creates Pod sandboxes and containers. kube-proxy is the traffic-rule controller: it turns Service and EndpointSlice state into local packet-forwarding behavior.



Three ownership loops on a worker node

kubelet reconciles Pods, the runtime executes containers, and kube-proxy maintains Service forwarding state.

The arrows explain the ownership boundaries. The API server does not start a container. The scheduler does not contact the runtime. kube-proxy does not decide where a Pod runs. Each component watches or receives a particular kind of desired state and reconciles its own local result.

Because the three components expose different evidence, administrators move between the cluster API, the host service manager, and the CRI instead of expecting one command to explain the entire node.

A Node object is the API view of a machine

The physical or virtual machine is the worker host. Its Node object is the control plane's API representation of that host: identity, labels, taints, addresses, capacity, allocatable resources, software versions, and conditions. The object is valuable evidence, but it is not the machine itself.

Start with an inventory view. The custom columns below keep the three facts most relevant to this lesson together: readiness, kubelet version, and runtime implementation.

```bash
kubectl get nodes -o custom-columns='NAME:.metadata.name,READY:.status.conditions[?(@.type=="Ready")].status,KUBELET:.status.nodeInfo.kubeletVersion,RUNTIME:.status.nodeInfo.containerRuntimeVersion'
```

```plaintext
NAME      READY   KUBELET   RUNTIME
worker1   True    v1.34.1   containerd://2.0.6
worker2   True    v1.34.1   containerd://2.0.6
```

The version values are examples; your cluster will differ. READY=True means the control plane currently regards the node as healthy enough to accept work. It does not prove that every container or every network path on that node is healthy.

kubelet: the node-level Pod reconciler

kubelet is an agent on each node. For ordinary API-managed Pods, it watches for Pod specifications whose spec.nodeName matches its node. It then repeatedly compares those specifications with runtime state and acts to make the containers run as declared.

This is the same reconciliation idea used by controllers, but the output is local execution rather than another API object. Because the scheduler has already selected the node, kubelet does not reschedule the Pod elsewhere. Because the runtime owns low-level execution, kubelet does not implement container creation itself.

From assignment to observed status

1. The scheduler records a node name on the Pod through the API.
2. The kubelet on that node observes the assigned PodSpec.

3. The kubelet asks the runtime through CRI to prepare the Pod sandbox, obtain images, and create and start containers.
4. The kubelet evaluates container and probe results and updates Pod status through the API server.
5. If actual state later drifts—for example, a restartable container exits—the kubelet continues reconciling according to the Pod specification.

The kubelet also mounts declared volumes, runs configured probes, reports resource information, and manages static Pods such as the control plane examples from post 15. Those actions still support its central promise: keep the assigned Pod's local reality aligned with its specification.

Node status and heartbeats

The kubelet reports node conditions such as Ready, MemoryPressure, DiskPressure, and PIDPressure. These are summaries of the node's state, not diagnoses by themselves. kubectl describe node also combines conditions with capacity, allocatable resources, and recent events.

```bash
NODE=worker1

kubectl describe node "$NODE"
kubectl get node "$NODE" \
  -o custom-columns='TYPE:.status.conditions[*].type,STATUS:.status.conditions[*].status'
```

```plaintext
TYPE     MemoryPressure,DiskPressure,PIDPressure,Ready
STATUS   False,False,False,True
```

Read each condition by position: the example reports no memory, disk, or process-ID pressure and reports the node Ready. When detail matters, use kubectl describe node or JSON/YAML rather than relying on this compressed display.

For liveness, the kubelet also renews a lightweight Lease object with the same name as the Node in the kube-node-lease namespace. The control plane uses that heartbeat alongside Node status to judge reachability.

```bash
kubectl get lease -n kube-node-lease "$NODE" \
  -o custom-columns='NODE:.metadata.name,HOLDER:.spec.holderIdentity,RENEWED:.spec.renewTime'
```

```plaintext
NODE     HOLDER    RENEWED
worker1  worker1   2026-09-26T11:48:32.123456Z
```

A recent renewTime shows that the control plane is receiving kubelet heartbeats. If the timestamp stops advancing, the important boundary is communication between that kubelet and the API server—not kube-proxy's Service rules.

Observe kubelet on the host

On kubeadm-style Linux nodes, kubelet is normally a systemd service rather than a Pod. When the API view cannot explain why a node stopped reporting, inspect that host service directly. The exact unit layout can differ on managed or non-systemd systems.

```bash
sudo systemctl is-active kubelet
sudo systemctl status kubelet --no-pager
sudo journalctl -u kubelet --since '15 minutes ago' --no-pager
```

systemctl answers whether the host supervisor considers kubelet active. journalctl provides the process's local explanation: API authentication errors, rejected configuration, runtime connection failures, image errors, probe results, or failed volume

setup. Filter by time first so old failures do not distract from current behavior.

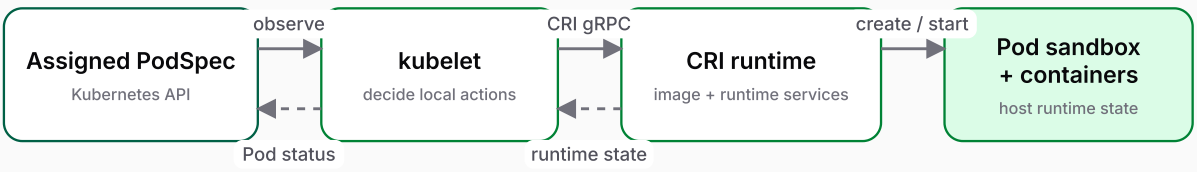
```bash
sudo systemctl cat kubelet
sudo grep -E '^(apiVersion|kind|containerRuntimeEndpoint|staticPodPath):' \
  /var/lib/kubelet/config.yaml
```

The unit shows how kubelet starts; the configuration file shows component settings when that conventional kubeadm path is used. Inspect before editing. Linux preparation and runtime configuration are the subjects of post 17, while systematic node-failure repair belongs to the later Practice phase.

The container runtime: local execution behind CRI

A container runtime manages the container lifecycle on the host. Kubernetes commonly uses CRI-compatible runtimes such as containerd or CRI-O. The kubelet acts as a CRI client and communicates with the runtime through a local gRPC endpoint, commonly exposed as a Unix socket on Linux.

CRI is the contract that keeps Kubernetes orchestration separate from a particular runtime implementation. Because the runtime implements that contract, kubelet can request image operations and Pod/container operations without embedding containerd- or CRI-O-specific logic.



Typical CRI operations: PullImage • RunPodSandbox • CreateContainer • StartContainer
The runtime never receives a PodSpec directly from the API server.

From an assigned PodSpec to running containers

The API communicates Pod intent to kubelet; only kubelet communicates that intent to the runtime through CRI.

The runtime does not choose a node, reconcile a Deployment, or interpret a Service. It executes local requests and returns local runtime state. That narrower responsibility is why a Pod can be assigned to a node yet still fail before its application process starts: scheduling succeeded, but node-side realization did not.

Identify the runtime from Kubernetes

```bash
kubectl get node "$NODE" \
  -o jsonpath='{.status.nodeInfo.containerRuntimeVersion}{"\n"}'
```

```plaintext
containerd://2.0.6
```

This value comes from Node status and tells you which service and documentation are relevant on that host. If it reports containerd, inspect containerd; if it reports CRI-O, inspect CRI-O. Do not assume Docker commands describe modern Kubernetes runtime state.

Use crictl for the CRI layer

crictl is a command-line client for CRI-compatible runtimes. Run it on the node, with the endpoint configured for that node's runtime. It is especially useful when kubectl shows an assigned Pod but you need to see whether a sandbox or container exists locally.

```bash
sudo crictl info
sudo crictl pods
sudo crictl ps -a
sudo crictl images
```

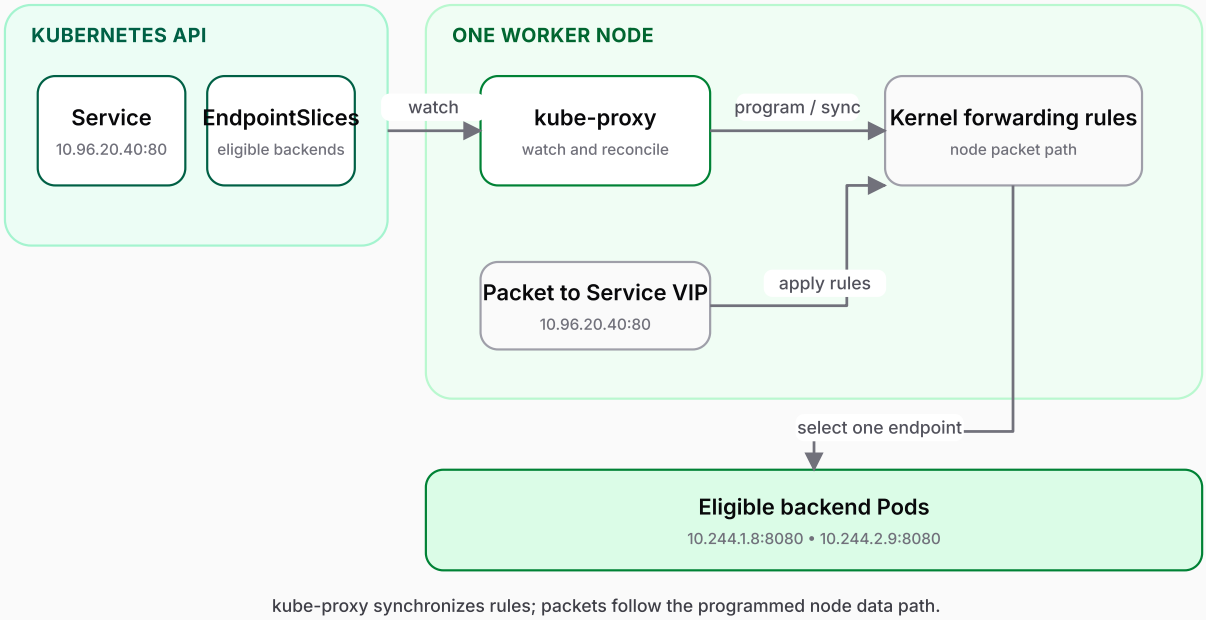
info checks the CRI connection and reports runtime configuration. pods lists Pod sandboxes; ps -a includes stopped as well as running containers; images shows the runtime's local image store. These are runtime facts, so they can remain available even when the API server or kubectl path is unavailable.

A warning that crictl is probing known endpoints means its runtime endpoint is not configured explicitly. Confirm the endpoint used by kubelet, then configure crictl consistently; post 17 covers that host setup. Do not select a socket merely because a file with a familiar name exists.

kube-proxy: Service state becomes node rules

Pods have changing addresses, while a Kubernetes Service provides a stable virtual destination. kube-proxy watches Service and EndpointSlice objects, then reconciles the node's forwarding rules so traffic for a Service can reach an eligible backend. Services

and EndpointSlices receive their full treatment in post 28; here the key point is ownership on the worker node.



How Service state becomes a node data path

kube-proxy synchronizes rules from API state; packets then follow the node's programmed forwarding path.

Despite its name, kube-proxy commonly programs packet-processing rules and is not a userspace application proxy through which every byte must pass. On Linux it can use backends such as iptables or nftables; the configured mode and Kubernetes version determine the exact mechanism.

kube-proxy is also not the Container Network Interface (CNI) plugin. The CNI plugin establishes Pod network connectivity; kube-proxy implements the usual Service virtual-IP behavior on top of that connectivity. Extension interfaces appear in post 25, and the networking model appears in post 27.

kube-proxy is optional because some network implementations replace its Service data plane. Do not infer that a cluster is broken merely because no kube-proxy Pods exist; first determine which architecture the cluster intentionally uses.

Observe kube-proxy in a kubeadm cluster

kubeadm normally deploys kube-proxy as a DaemonSet in kube-system, giving each eligible node one kube-proxy Pod. The DaemonSet owns distribution; the kubelet and runtime on each node execute that Pod.

```bash
kubectl get daemonset kube-proxy -n kube-system
kubectl get pods -n kube-system -l k8s-app=kube-proxy -o wide
```

```plaintext
NAME        DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE
kube-proxy  2         2         2       2            2

NAME              READY   STATUS    NODE
kube-proxy-7m2qd  1/1     Running   worker1
kube-proxy-w8kz4  1/1     Running   worker2
```

Desired equals the number of nodes on which the DaemonSet should run; Ready confirms the Pods report ready. Include -o wide because node placement is the point of this observation.

```bash
PROXY_POD=$(kubectl get pod -n kube-system \
  -l k8s-app=kube-proxy \
  --field-selector spec.nodeName="$NODE" \
  -o jsonpath='{.items[0].metadata.name}')

kubectl logs -n kube-system "$PROXY_POD" --tail=50
```

The field selector chooses the kube-proxy instance for the node already under inspection. Its logs can show startup mode and synchronization errors. The cluster's actual configuration remains authoritative; on kubeadm, it is commonly stored in the kube-proxy ConfigMap.

```bash
kubectl get configmap kube-proxy -n kube-system \
  -o go-template='{{index .data "config.conf"}}' | grep '^mode:'
```

An explicit value identifies the configured proxy backend. An empty value delegates selection to kube-proxy's version-specific default, so avoid claiming a mode from memory; inspect the configuration and startup logs for the cluster in front of you.

Do not confuse the three ownership loops

- kubelet asks, “Are the Pods assigned to my node realized and reported correctly?”
- The runtime answers, “Which sandboxes, containers, and images exist on this host, and what state are they in?”
- kube-proxy asks, “Do this node's Service forwarding rules match the current Services and EndpointSlices?”

The questions overlap operationally but not conceptually. A running container proves the runtime completed local work. It does not prove kubelet can still report to the API, and it does not prove a Service virtual IP is routed correctly.

Worked demonstration: follow one Pod onto a worker

This demonstration creates one ordinary Pod, observes its API assignment, then inspects the matching runtime objects on that node. It assumes you can connect to the selected Linux worker and that crictl is configured there.

node-demo.yaml · yaml

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: worker-anatomy
---
apiVersion: v1
kind: Pod
metadata:
  name: node-demo
  namespace: worker-anatomy
  labels:
    app: node-demo
spec:
  containers:
    - name: web
      image: nginx:1.27-alpine
```

The manifest deliberately contains no spec.nodeName. The scheduler performs the placement taught in post 11; this lesson begins at the handoff to the chosen node.

bash

```bash
kubectl apply -f node-demo.yaml
kubectl get pod node-demo -n worker-anatomy -o wide -w
```

plaintext

```
NAME        READY   STATUS              RESTARTS   IP       NODE
node-demo   0/1     ContainerCreating   0          <none>   worker1
node-demo   1/1     Running             0          10.244.1.8   worker1
```

The first row already names worker1, so scheduling is complete. ContainerCreating means node-side realization is still in progress. When the second row appears, kubelet has observed runtime success and reported the Pod Running. End the watch with Ctrl-C.

```bash
NODE=$(kubectl get pod node-demo -n worker-anatomy \
  -o jsonpath='{.spec.nodeName}')
POD_UID=$(kubectl get pod node-demo -n worker-anatomy \
  -o jsonpath='{.metadata.uid}')

printf 'node=%s\npod uid=%s\n' "$NODE" "$POD_UID"
```

Connect to the node printed above. The exact connection method depends on the environment. Query the runtime there by Pod name and namespace, then inspect the containers associated with the returned sandbox.

```bash
sudo crictl pods --name node-demo --namespace worker-anatomy
sudo crictl ps --name web
```

```plaintext
POD ID        CREATED          STATE   NAME       NAMESPACE
7f3a...       35 seconds ago   Ready   node-demo  worker-anatomy

CONTAINER     IMAGE      CREATED          STATE     NAME   POD ID
4b21...       a6f2...    34 seconds ago   Running   web    7f3a...
```

The IDs and timestamps will differ. The important relationship is one CRI Pod sandbox named node-demo containing the web container. kubectl saw the API object; crictl sees the runtime objects that realize it on worker1.

```bash
CONTAINER_ID=$(sudo crictl ps --name web -q | head -n 1)
sudo crictl inspect "$CONTAINER_ID" | sed -n '1,80p'
sudo crictl logs "$CONTAINER_ID" | tail -n 20
```

inspect exposes detailed runtime metadata and state; logs reads this container's CRI log stream. For routine application work, kubectl describe and kubectl logs remain the portable first choices. Reach for crictl when the node-runtime boundary itself is what you need to observe.

```bash
kubectl delete namespace worker-anatomy
```

Deletion reverses the same path: the API object disappears, kubelet observes that the assigned Pod is no longer desired, and the runtime removes the Pod's local containers and sandbox.

A reusable observation order

1. Use kubectl to establish API facts: assignment, Pod status, Node conditions, events, and the reported runtime.
2. On the selected host, use systemctl and journalctl to establish whether kubelet and the runtime service are active and what they report locally.
3. Use crictl to establish whether the expected sandbox, container, image, and runtime state exist.
4. If the issue concerns a Service path, inspect the node's kube-proxy instance and its actual configured mode rather than treating Pod health as proof of Service health.

This is an evidence order, not a troubleshooting challenge. It prevents a common category error: looking in scheduler logs for a runtime failure, or restarting kubelet because a Service rule is stale.

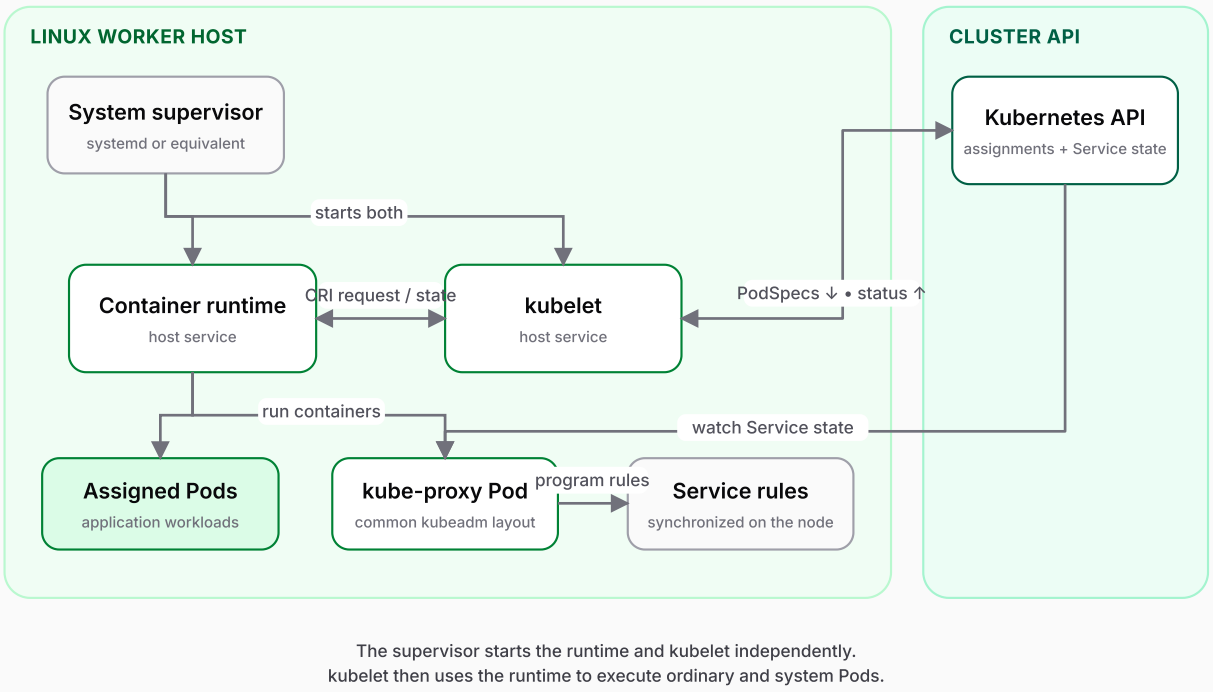
Read symptoms as ownership clues

- A Pod with no node name has not completed scheduling; no kubelet owns its execution yet.

- A Pod with a node name but no running container moves attention to kubelet, runtime, image, volume, and node-local evidence.
- A stale Node Lease or Unknown readiness points to the kubelet-to-API heartbeat path.
- Healthy direct Pod access alongside a node-specific Service failure can point toward kube-proxy or node forwarding state.

These clues narrow the responsible layer; they do not replace evidence. Posts 36 and 45 later turn these signals into structured troubleshooting workflows.

The worker-node startup dependency



Worker node startup: host services before system Pods

The supervisor starts the execution foundation; kubelet then uses the runtime to realize both application and system Pods.

This dependency chain explains why kubelet and the runtime are usually host services while kube-proxy can itself run as a Pod. The node needs its execution foundation before it can run ordinary system Pods. The exact installation and configuration

sequence belongs to the next article.

What to remember

- The scheduler assigns a Pod; kubelet realizes that assignment on the selected node.
- kubelet reconciles Pod intent and reports Pod and Node state; it delegates container execution through CRI.
- The runtime manages local sandboxes, images, and containers; crictl exposes that CRI layer.
- kube-proxy watches Services and EndpointSlices and normally reconciles node packet-forwarding rules; it does not schedule Pods or provide the Pod network.
- A Node object and its Lease are API evidence about the machine; host commands reveal the local services behind that evidence.
- Choose the observation layer from the ownership boundary: kubectl for API state, systemctl and journalctl for host services, and crictl for runtime state.

Official references

Kubernetes Components summarizes the roles of kubelet, kube-proxy, and the container runtime.

Nodes explains Node registration, status, conditions, capacity, and heartbeats.

Leases documents the lightweight heartbeat objects renewed by kubelets.

Container Runtime Interface defines the kubelet-to-runtime gRPC boundary.

Debugging Kubernetes nodes with crictl documents CRI endpoint configuration and runtime inspection commands.

Virtual IPs and Service Proxies describes how kube-proxy synchronizes Service and EndpointSlice state into node forwarding behavior.

Local files and paths used by the kubelet explains common Linux configuration, state, socket, and manifest locations without treating them as universal APIs.