Omar Latrach

TECHNICAL ARTICLE

PUBLISHED · OCTOBER 6, 2026

# Kubernetes Extension Interfaces: CNI, CSI, and CRI

Understand how CRI, CNI, and CSI let Kubernetes delegate container execution, Pod networking, and storage—and learn how to observe each boundary as an administrator.




This is Learn post 25 of the 54-post Certified Kubernetes Administrator (CKA) preparation path. You already know that kubelet reconciles assigned Pods, containerd can execute them, and a CNI add-on is needed before a kubeadm cluster becomes fully ready. This lesson connects those facts through the extension contracts that keep Kubernetes independent of any one runtime, network, or storage provider.

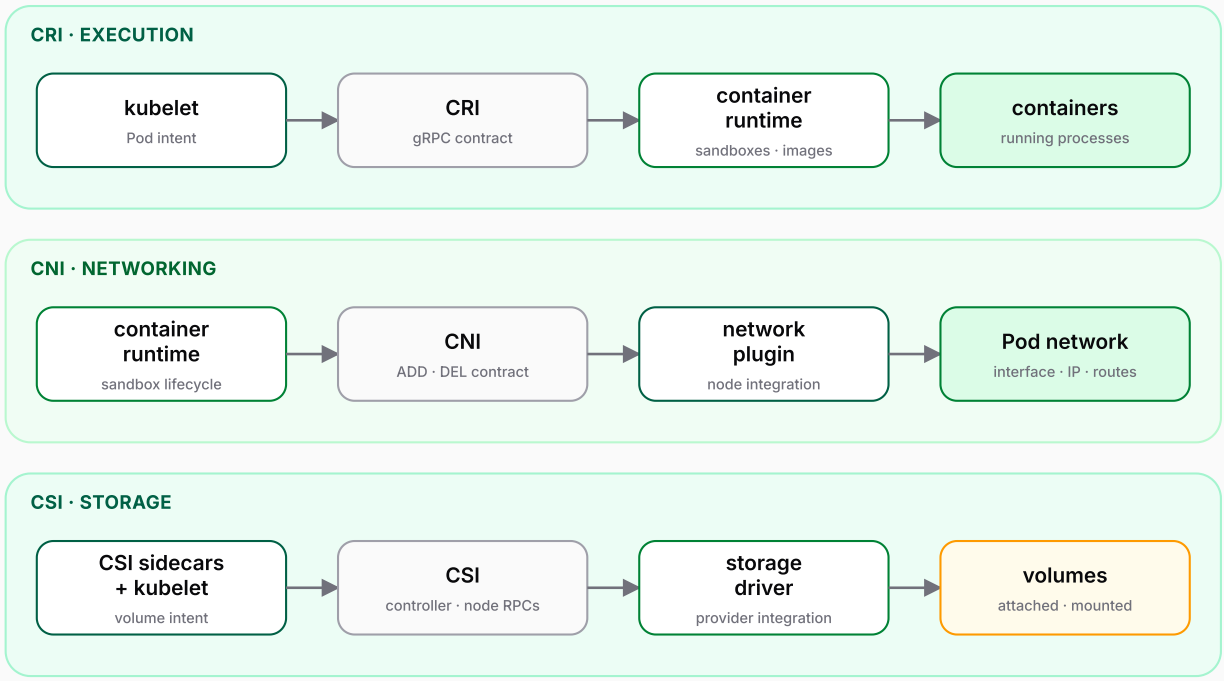
The important administrator skill is not memorizing a catalog of products. It is recognizing the boundary that owns a result: Container Runtime Interface (CRI) for execution, Container Network Interface (CNI) for Pod network setup, and Container Storage Interface (CSI) for storage operations.

## What you'll learn

- Explain the problem solved by CRI, CNI, and CSI and identify the components on each side of every contract.
- Follow a Pod from kubelet intent to a runtime sandbox, network configuration, containers, and optional storage.
- Inspect the runtime boundary with Node status and crictl.
- Observe CNI results in a normal Pod and distinguish them from Services, DNS, and NetworkPolicy.
- Read CSIDriver and CSINode resources without getting ahead of the later persistent-storage lessons.

## The mental model: three replaceable adapters

> Kubernetes declares what a Pod needs; extension interfaces standardize how specialized implementations deliver execution, networking, and storage. The contract stays recognizable even when the implementation changes.




**Three extension contracts connect Kubernetes intent to node capabilities**

Remember the nouns by their result: CRI runs, CNI connects, and CSI mounts. Each interface separates Kubernetes from a replaceable implementation.

These interfaces are integration boundaries, not three workload controllers. They do not replace the scheduler, kubelet, or the API server. Instead, they let those Kubernetes components rely on stable behavior without embedding provider-specific code for every runtime, network fabric, or storage system.

Because the contract and implementation are separate, replacing an implementation does not change the meaning of a Pod. It does change node software, configuration, compatibility requirements, and operational evidence—which is why administrators must know where to look.

# Interface, implementation, and API resource are different things

the three boundaries · text

| Interface | Connects | Typical implementation result |
| --- | --- | --- |
| CRI | kubelet ↔ container runtime | Pod sandbox, images, containers |
| CNI | container runtime ↔ network plugin | interface, Pod IP, routes, cleanup |
| CSI | CSI sidecars/kubelet ↔ storage driver | provision, attach, stage, mount |

An interface defines calls and expected behavior. An implementation supplies that behavior: containerd or CRI-O for CRI, a compatible network provider for CNI, and a vendor or platform driver for CSI. Product names are not interchangeable with the interface names.

The Kubernetes API does not expose a standard CRI object or a standard CNI object that lists the active runtime or network provider. CRI is visible indirectly through Node status and directly through the node-local socket. CNI is visible through its add-on workloads, node files, Pod results, Events, and logs. CSI is different: Kubernetes defines cluster-scoped CSIDriver and CSINode API resources for driver behavior and per-node registration.

# One Pod can cross all three boundaries

After the scheduler binds a Pod to a node, that node's kubelet observes the assignment. Kubelet asks the runtime through CRI to prepare a Pod sandbox and containers. As part of sandbox networking, the runtime uses the configured CNI integration. If the Pod requests a CSI-backed volume, storage control-plane sidecars and the node's kubelet coordinate with the CSI driver so the volume is ready at the container mount path.

These dependencies converge before the application can work, but do not reduce them to one fixed command sequence. Image pulling, volume preparation, sandbox creation, and reconciliation can overlap or retry. The useful model is ownership: CRI produces runnable containers, CNI produces usable Pod networking, and CSI produces usable storage.

A Pod with no CSI-backed volume still uses CRI and normally CNI. A static control-plane Pod also uses the runtime and node networking behavior even though no scheduler selected its node. Interfaces are activated by the capabilities a Pod needs, not by its controller type.

# CRI: kubelet asks; the runtime executes

CRI is the gRPC protocol between kubelet and a container runtime on the same node. Kubelet is the client. The runtime exposes runtime and image services through an endpoint, commonly a Unix domain socket on Linux. This keeps kubelet focused on Pod reconciliation while the runtime handles images, sandboxes, containers, and low-level runtime integration.

The earlier node-preparation lesson configured containerd at unix:///run/containerd/containerd.sock. That path is not a Kubernetes API URL. It is local to the node, and the API server does not call it. If the socket or CRI service fails, kubelet loses its execution boundary even if the API server remains healthy.

### See the runtime identity reported by kubelet

Start from the cluster API when you want an inventory view. Node status reports the runtime and kubelet versions that each kubelet has published.

```bash
kubectl get nodes \
  -o custom-
columns='NODE:.metadata.name,RUNTIME:.status.nodeInfo.containerRuntimeVersion,KUBELET:.status.nodeInfo.kubeletVersio
n'
```

```text
NODE       RUNTIME               KUBELET
cp1        containerd://2.x.y    v1.x.y
worker-1   containerd://2.x.y    v1.x.y
```

The exact versions vary. The useful observation is that kubelet identifies the CRI implementation it is using. This is reported state, not a live socket test, so move to the node when you need to prove the boundary currently answers.

## Use crictl at the CRI layer

crictl is the administrator-facing client for a CRI-compatible runtime. Run it on the node and use the same endpoint as kubelet. The commands below ask for runtime information, Pod sandboxes, and containers—including exited containers.

```bash
sudo crictl \
  --runtime-endpoint unix:///run/containerd/containerd.sock \
  info

sudo crictl \
  --runtime-endpoint unix:///run/containerd/containerd.sock \
  pods

sudo crictl \
  --runtime-endpoint unix:///run/containerd/containerd.sock \
  ps -a
```

A structured info response proves that the CRI service answered. crictl pods lists sandboxes—the runtime boundary around a Pod—while crictl ps -a lists containers inside those sandboxes. kubectl reports API objects; crictl reports runtime objects on one node. Their identifiers and lifecycles are related but not identical.

Do not substitute a vendor tool and assume you tested CRI. For example, containerd's ctr can inspect containerd internals, but crictl exercises the CRI service kubelet depends on. Likewise, Docker Engine alone is not a CRI endpoint; Kubernetes requires a CRI-compatible runtime or adapter.

## CNI: the Pod sandbox receives a network identity

CNI standardizes node-local network setup and teardown. On current Kubernetes releases, the container runtime manages the CNI integration. When the runtime prepares a Pod sandbox, it uses the selected CNI configuration and plugins to create the required network result; when the sandbox is removed, it requests cleanup.

A provider usually also deploys Kubernetes workloads such as a per-node DaemonSet and sometimes controller components. Those long-running components prepare node networking or program the provider's data plane. The CNI call is still the per-sandbox contract. The provider implementation and the interface call are related, but they are not the same process.

The deeper packet path belongs to post 27. For now, remember the observable result: a normal Pod has a network namespace, loopback, one or more interfaces, an address, and routes appropriate to the provider. CNI establishes that foundation; it does not define Service virtual IP behavior or DNS records.

## Inspect the installed network add-on

Begin with kube-system to find node agents and controllers. Names and labels are provider-specific, so inspect what the cluster actually runs instead of assuming a product name.

```bash
kubectl get daemonsets,deployments \
  --namespace kube-system

kubectl get pods \
  --namespace kube-system \
  --output wide
```

A networking DaemonSet normally wants one node agent on every eligible node. Wide Pod output shows placement and readiness. A healthy DaemonSet is useful evidence, but the definitive result for a workload is whether its sandbox receives working network configuration.

On a Linux node, these are common CNI locations. The runtime and provider documentation remain authoritative because packaging can change them.

```bash
sudo ls -1 /etc/cni/net.d
sudo ls -1 /opt/cni/bin | sed -n '1,20p'
```

Configuration under /etc/cni/net.d tells the runtime which CNI chain to use; executables are commonly installed under /opt/cni/bin. Treat the files as node configuration managed by the selected provider. Do not combine configurations from two primary network providers on the same node.

## Observe a normal CNI result

Create one ordinary Pod so the runtime must create a sandbox and arrange its network. This is a demonstration of normal behavior, not a troubleshooting challenge.

```bash
kubectl create namespace interface-demo

kubectl run web \
  --namespace interface-demo \
  --image=nginx:1.27.4 \
  --labels=app=web \
  --restart=Never

kubectl wait \
  --namespace interface-demo \
  --for=condition=Ready pod/web \
  --timeout=90s

kubectl get pod web \
  --namespace interface-demo \
  --output wide
```

```text
NAME   READY   STATUS    RESTARTS   IP           NODE
web    1/1     Running   0          10.244.1.17   worker-1
```

Addresses and nodes vary. The stable relationship is that the scheduler selected worker-1, kubelet asked its runtime to create the sandbox, and the CNI integration returned network configuration including the Pod IP. The runtime could then start the nginx container inside that sandbox.

If sandbox networking cannot be created, inspect the Pod's Events before looking inside the application container. A FailedCreatePodSandBox event mentioning CNI points below the application layer; the container may not exist yet.

```bash
kubectl describe pod web \
  --namespace interface-demo

kubectl get events \
  --namespace interface-demo \
  --sort-by=.metadata.creationTimestamp
```

### Do not confuse CNI with adjacent networking layers

```text
CNI             Gives Pod sandboxes their network connectivity
kube-proxy      Implements Service forwarding on nodes in many clusters
CoreDNS         Answers cluster DNS queries
NetworkPolicy   Declares allowed traffic when the network provider enforces it
```

One networking provider may participate in more than one row, but the responsibilities remain distinct. Posts 27–32 develop Pod connectivity, Services, EndpointSlices, DNS, NetworkPolicy, Ingress, and Gateway API in that order.

## CSI: storage drivers stay outside Kubernetes core

CSI lets storage systems integrate through a standard driver interface rather than requiring provider code to be compiled into Kubernetes. A typical driver has a controller component and a node component. They implement different parts of the same storage lifecycle.

The controller component commonly runs as a Deployment or StatefulSet with Kubernetes CSI sidecars. Sidecars watch Kubernetes API objects and call the driver's CSI controller service for operations such as provisioning or attachment when the driver supports them. Kubernetes control-plane components do not directly open the driver's CSI socket.

The node component normally runs as a DaemonSet. Its node-driver-registrar sidecar registers the driver's Unix socket with kubelet. Kubelet can then call the CSI node service to stage and publish a volume on that node, making the mounted path available to the Pod.

## Read the CSI API resources

Unlike CRI and CNI, CSI has standard discovery objects in storage.k8s.io/v1. List the resource types first, then inspect the drivers and per-node registrations present in the cluster.

```bash
kubectl api-resources \
  --api-group=storage.k8s.io | grep '^csi'

kubectl get csidrivers.storage.k8s.io \
  -o custom-columns='DRIVER:.metadata.name,ATTACH_REQUIRED:.spec.attachRequired,MODES:.spec.volumeLifecycleModes'

kubectl get csinodes.storage.k8s.io \
  -o custom-columns='NODE:.metadata.name,DRIVERS:.spec.drivers[*].name'
```

```text
DRIVER         ATTACH_REQUIRED   MODES
<driver-name>   true             [Persistent]

NODE       DRIVERS
cp1        <driver-name>
worker-1   <driver-name>
```

Provider names, fields, and supported modes vary. A CSIDriver object describes cluster-wide driver behavior that Kubernetes needs to know. A CSINode object has the same name as its Node and lists CSI drivers registered with kubelet there. Seeing a driver cluster-wide does not prove that its node service is registered on every node.

Inspect one node when placement matters. The node ID is the storage provider's identity for that Kubernetes node; it does not have to equal the Kubernetes Node name.

```bash
kubectl get csinode worker-1 \
  -o jsonpath='{range .spec.drivers[*]}{.name}{"\t"}{.nodeID}{"\n"}{end}'
```

```text
<driver-name>    <provider-node-id>
```

Then find the driver's controller and node workloads using the installation namespace and labels documented by the provider. CSI Pod names are not standardized, so discovery should follow the installed add-on rather than a memorized grep pattern.

```bash
kubectl get daemonsets,deployments \
  --all-namespaces

kubectl get pods \
  --all-namespaces \
  --output wide
```

PersistentVolumes, PersistentVolumeClaims, StorageClasses, access modes, reclaim policies, and dynamic provisioning are intentionally deferred to posts 33–35. At this point, learn the integration boundary: CSI drivers translate Kubernetes storage intent into provider operations and node mounts.

# Read the demonstration across layers

The interface-demo Pod gives you one object to correlate. Ask the API for its node and addresses, then connect to that node and find the runtime sandbox with crictl.

```bash
kubectl get pod web \
  --namespace interface-demo \
  -o custom-columns='POD:.metadata.name,NODE:.spec.nodeName,POD_IP:.status.podIP,HOST_IP:.status.hostIP'

# Run on the node printed above.
sudo crictl \
  --runtime-endpoint unix:///run/containerd/containerd.sock \
  pods --name web
```

```text
POD   NODE      POD_IP       HOST_IP
web   worker-1   10.244.1.17   192.0.2.21

POD ID       CREATED       STATE   NAME   NAMESPACE
<id>         <time>        Ready   web    interface-demo
```

The API view and runtime view meet at the same Pod name, namespace, and node. The runtime's Ready sandbox is evidence from CRI; the Pod IP is a result of CNI integration. This Pod requests no CSI volume, so CSI is not part of its startup path. That absence is useful: interfaces participate only when their capability is needed.

Remove the demonstration namespace when you are finished. Kubernetes deletes the Pod, kubelet asks the runtime to stop and remove its containers and sandbox, and the runtime requests network cleanup through CNI.

```bash
kubectl delete namespace interface-demo
```

# Choose evidence from the failing boundary

```text
Question                       Start with
------------------------------ ----------------------------------------------
Which runtime does the node use? Node .status.nodeInfo.containerRuntimeVersion
Does CRI answer locally?         crictl info; kubelet and runtime service logs
Did sandbox networking fail?     Pod Events; CNI agent and runtime logs
Which CSI drivers are registered? CSIDriver and CSINode resources
Did a CSI mount fail?            Pod Events; CSI node/controller Pod logs
```

This is a layer map, not a claim that every failure has one cause. A Pod stuck before container creation can reflect runtime, networking, or volume preparation. Read the Event message, identify the named boundary, and then move to node-local or add-on evidence.

kubectl logs reads a container's logs through kubelet and the runtime. If the application container was never created, there may be no application log to read. Pod Events, kubelet logs, runtime evidence, and extension-component logs become the appropriate sources instead. Systematic troubleshooting is developed later in posts 36 and 37–48.

# Important distinctions to keep

- CRI is the kubelet-to-runtime protocol. containerd and CRI-O are implementations; a container image runtime such as runc operates below that boundary.
- CNI configures Pod sandbox networking. kube-proxy, CoreDNS, Services, and NetworkPolicy have adjacent but different responsibilities.
- CSI has controller-side and node-side operations. A controller component being healthy does not prove registration or mounting works on a particular node.
- DaemonSet is a deployment pattern, not an interface. CNI agents and CSI node plugins may both use DaemonSets for different reasons.
- A Kubernetes upgrade does not automatically upgrade the runtime, CNI provider, or CSI driver. Check each implementation's compatibility policy before changing cluster versions.

- Installing two competing primary implementations does not provide redundancy. Each node needs one coherent runtime endpoint and one coherent primary Pod-network configuration; CSI can legitimately have multiple drivers for different storage systems.

## Administrator checklist

- Name the desired result first: container execution, Pod networking, or storage attachment and mounting.
- Separate the interface from its installed implementation and version.
- Use kubectl for API state, crictl for the node's CRI state, and provider workloads plus node files for implementation evidence.
- Correlate Pod name, namespace, node, sandbox, network result, and any driver registration instead of reading one layer in isolation.
- Verify compatibility and rollout guidance from the implementation provider before installs or upgrades.

## Where this fits in the series

Posts 16–20 introduced worker-node components, configured containerd, installed CNI during kubeadm bootstrap, joined nodes, and planned compatible upgrades. This lesson turns those product-level steps into a reusable interface model. Post 26 next covers a different extension mechanism: CustomResourceDefinitions and Operators extend the Kubernetes API itself. Posts 27–32 then develop networking behavior, and posts 33–35 develop the storage resources that drive CSI workflows.

## Official documentation

Container Runtime Interface (CRI) defines the kubelet-to-runtime protocol and current API requirements.

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

Network Plugins explains current Kubernetes CNI integration and network-plugin requirements.

Compute, Storage, and Networking Extensions places network and storage plugins in Kubernetes' wider extension model.

Deploying a CSI Driver on Kubernetes explains controller components, node components, sidecars, registration, and kubelet communication.

CSIDriver API reference documents the cluster-scoped object that describes driver behavior.

CSINode API reference documents per-node CSI driver registration information.