Omar Latrach

TECHNICAL ARTICLE

PUBLISHED · SEPTEMBER 21, 2026

Scheduling Foundations: The Scheduler, nodeSelector, and nodeName

Understand how Kubernetes filters, scores, and binds Pods to nodes; constrain placement with nodeSelector; use nodeName safely; and read the evidence when a Pod stays unscheduled.

This is Learn post 11 of the 54-post Certified Kubernetes Administrator (CKA) preparation path. Earlier posts established that controllers create Pods and that resource requests describe each Pod's scheduling footprint. Now we follow an admitted Pod through the placement decision: which node should become responsible for running it?

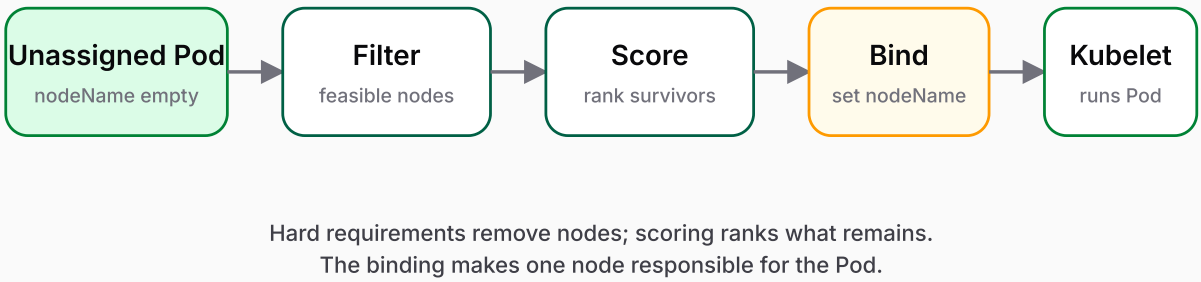
Scheduling is not container startup. The scheduler chooses a node and records that choice; the kubelet on the chosen node then works to start the Pod. That separation explains why a Pod can be assigned to a node yet still not be Running.

What you'll learn

- Trace a Pod through filtering, scoring, binding, and the handoff to kubelet.
- Use node labels and nodeSelector to constrain the set of eligible nodes.
- Recognize an unscheduled Pod and read the scheduler's evidence with kubectl.
- Distinguish a scheduler constraint from direct assignment with nodeName.
- Choose the most useful commands for explaining where a Pod was placed—or why it was not placed.

The mental model: choose, record, act

> The scheduler chooses. The API records. The kubelet acts.



Hard requirements remove nodes; scoring ranks what remains.
The binding makes one node responsible for the Pod.

A Pod moves from unassigned to owned by one node

Hard requirements decide which nodes remain possible. Scoring chooses among those feasible nodes. Binding records one node, and that node's kubelet takes over.

A newly stored Pod normally has an empty spec.nodeName. The default kube-scheduler watches for Pods in that state. It does not launch containers and it does not continuously move running Pods around; its central job is to select a node for each unscheduled Pod.

After a choice is made, the scheduler binds the Pod to the node through the API. The binding is reflected in spec.nodeName. The kubelet on that node watches for assigned Pods and begins the work of pulling images, preparing volumes and networking, and starting containers.

Read assignment separately from runtime state

The NODE column is the fastest placement check. An empty value means no node has been assigned yet; a node name means scheduling or direct assignment has already happened.

```bash
kubectl get pods -A -o wide
kubectl get pod <pod-name> -n <namespace> \
  -o jsonpath='{.spec.nodeName}{"\n"}'
```

Do not treat Pending as a synonym for unscheduled. Pending covers several pre-start situations. Check spec.nodeName: empty usually points you toward scheduling; populated points you toward the chosen node, kubelet, image, volume, or container startup path.

To list everything assigned to one node, use a field selector. This is useful before node maintenance and when investigating whether one node is unusually busy.

```bash
kubectl get pods -A \
  --field-selector spec.nodeName=<node-name> \
  -o custom-columns='NAMESPACE:.metadata.namespace,NAME:.metadata.name,PHASE:.status.phase'
```

How the scheduler reaches a decision

1. Filtering removes impossible nodes

A node is feasible only if it satisfies every hard requirement considered for the Pod. Resource requests from post 10 must fit within the node's remaining allocatable capacity. A nodeSelector must match. Other hard placement rules can also eliminate nodes; taints and affinity get their own treatment in posts 12 and 13.

Because filtering applies hard requirements, one failed requirement is enough to remove a node. If every node is removed, the Pod remains unassigned and the scheduler reports FailedScheduling events. It can try again when relevant cluster state changes.

2. Scoring ranks the feasible nodes

Filtering answers “can this Pod run here?” Scoring answers “which feasible node is the best choice?” The scheduler combines scores from its configured plugins and selects a highest-ranked node. A node does not need to be empty; it needs to pass the Pod's requirements and rank well enough among the candidates.

Resource scheduling is based on declared requests, not a snapshot of live CPU or memory use. A quiet node can still be unavailable to a Pod when existing requests have reserved its allocatable capacity. This is why kubectl top and the scheduler can appear to tell different stories without contradicting each other.

3. Binding records the winner

The scheduler submits a binding through the API, and the Pod's spec.nodeName identifies the selected node. This is the handoff point: the kubelet for that node becomes responsible for the Pod lifecycle. Scheduling succeeded even if a later image pull or container start fails.

nodeSelector constrains the choice with node labels

Nodes have labels just as Pods do. A nodeSelector is a map of required node label key-value pairs in the Pod spec. Every pair must match, so adding more entries narrows the eligible set. The scheduler still performs filtering, scoring, and binding within that set.

> nodeSelector says “choose a node from this labeled set,” not “use this exact node.”

Administrators commonly label nodes to describe real placement properties: a workload pool, hardware capability, or an organizational boundary. First inspect what the cluster already exposes.

```bash
kubectl get nodes --show-labels
kubectl get nodes -L kubernetes.io/hostname,kubernetes.io/os,kubernetes.io/arch
```

The -L form promotes selected labels into readable columns. Prefer stable, meaningful labels over encoding a particular machine name into workload configuration.

Worked demonstration: place a Deployment on a labeled node pool

This demonstration needs a cluster with at least one schedulable worker. Create an isolated namespace, choose a worker from kubectl get nodes, and give it a label that describes the intended workload pool.

```bash
kubectl create namespace cka-scheduling
kubectl get nodes -o wide
export TARGET_NODE=<worker-node-name>
kubectl label node "$TARGET_NODE" workload=payments
kubectl get node "$TARGET_NODE" -L workload
```

The last command should show payments in the WORKLOAD column. The label is cluster metadata; it does nothing by itself until a Pod asks for it.

An imperative command can generate a reliable Deployment starting point. The manifest matters here, so generate YAML locally, then add the placement field under the Pod template's spec.

```bash
kubectl create deployment payment-api \
  --image=nginx:1.27 \
  --replicas=2 \
  -n cka-scheduling \
  --dry-run=client -o yaml > payment-api.yaml
```

payment-api.yaml · yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: payment-api
  namespace: cka-scheduling
spec:
  replicas: 2
  selector:
    matchLabels:
      app: payment-api
  template:
    metadata:
      labels:
        app: payment-api
    spec:
      nodeSelector:
        workload: payments
      containers:
        - name: nginx
          image: nginx:1.27
```

Placement belongs in spec.template.spec because the Deployment creates Pods from that template. The Deployment selector under spec.selector has a different job: it identifies the Pods owned by the workload controller.

bash

```bash
kubectl apply -f payment-api.yaml
kubectl get pods -n cka-scheduling -o wide
kubectl get pods -n cka-scheduling \
  -l app=payment-api \
  -o custom-columns='NAME:.metadata.name,NODE:.spec.nodeName,PHASE:.status.phase'
```

Both replicas should show the node stored in TARGET_NODE, assuming it is the only schedulable node with workload=payments. If several nodes carry the label, the scheduler can choose among all of them. That flexibility is the point of selecting a pool rather than naming one machine.

See what happens when no node matches

The following Pod requests a label that does not exist yet. The API can store the valid object, but the scheduler cannot produce a feasible-node set.

waiting-for-label.yaml · yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: waiting-for-label
  namespace: cka-scheduling
spec:
  nodeSelector:
    accelerator: demo-gpu
  containers:
    - name: web
      image: nginx:1.27
```

bash

```bash
kubectl apply -f waiting-for-label.yaml
kubectl get pod waiting-for-label -n cka-scheduling -o wide
kubectl describe pod waiting-for-label -n cka-scheduling
```

The Pod should be Pending with an empty NODE column. Near the end of describe output, a FailedScheduling event will explain that nodes did not match the Pod's node selector. Node counts and exact wording vary by cluster; the important evidence is the event reason and message.

plaintext

```
Type     Reason            From               Message
Warning  FailedScheduling  default-scheduler  0/3 nodes are available: 3 node(s) didn't match Pod's node affinity/selector.
```

The scheduler retains responsibility for the unassigned Pod. Add the missing label and watch the same Pod become schedulable; no recreation is required.

```bash
kubectl label node "$TARGET_NODE" accelerator=demo-gpu
kubectl get pod waiting-for-label -n cka-scheduling --watch
```

Once the node matches, the scheduler can bind the Pod and the NODE column becomes populated. Stop the watch with Ctrl-C after the Pod is assigned.

Label changes affect future scheduling decisions

A nodeSelector is evaluated while the Pod is being scheduled. Removing the matching label afterward does not evict or relocate a Pod that is already bound. The existing Pod stays assigned; new or replacement Pods using the selector can remain Pending until a matching node exists.

```bash
kubectl label node "$TARGET_NODE" workload-
kubectl get pods -n cka-scheduling -l app=payment-api -o wide
kubectl scale deployment payment-api -n cka-scheduling --replicas=3
kubectl get pods -n cka-scheduling -l app=payment-api -o wide
```

The original replicas remain on the node, while the new replica should remain unassigned because no node now matches workload=payments. This is a normal demonstration of scheduling-time behavior, not a promise that Kubernetes continuously enforces node labels after binding.

Restore the label so the third replica can be scheduled before continuing.

```bash
kubectl label node "$TARGET_NODE" workload=payments
kubectl rollout status deployment/payment-api -n cka-scheduling
```

nodeName is direct assignment, not a selector

Setting spec.nodeName yourself bypasses normal scheduler selection. The scheduler ignores that Pod, and the kubelet whose node name matches the value attempts to run it. There is no filtering or scoring step to protect the choice.

direct-assignment.yaml · yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: direct-assignment
  namespace: cka-scheduling
spec:
  nodeName: worker-2 # Replace with an exact name from kubectl get nodes
  containers:
    - name: web
      image: nginx:1.27
```

This field is also where a successful scheduler binding becomes visible, but there is an important causal difference: normally the scheduler chooses and then sets the field; in this manifest, the user supplies the outcome in advance.

nodeSelector versus nodeName

- nodeSelector expresses a hard requirement over labels. It can match zero, one, or many nodes; kube-scheduler still checks the candidates and chooses one.
- nodeName supplies one exact node identity. It bypasses configured schedulers and overrides selector or affinity-based placement intent.

- With nodeSelector, no match leaves the Pod available for later scheduling when labels or capacity change. With a nonexistent nodeName, no scheduler repairs the choice and no kubelet claims the Pod.

Use nodeSelector for ordinary label-based placement. Reserve nodeName for advanced cases that intentionally require direct assignment, such as specialized components or scheduler-level workflows. It is usually the wrong way to express a workload preference.

A compact scheduling investigation

When a Pod does not start, follow the control-plane evidence in order. First establish whether a node was assigned; then inspect the specification and events that explain the decision.

```bash
# 1. Is the Pod assigned?
kubectl get pod <pod-name> -n <namespace> -o wide

# 2. What placement fields did the Pod declare?
kubectl get pod <pod-name> -n <namespace> -o yaml

# 3. Why did scheduling fail?
kubectl describe pod <pod-name> -n <namespace>

# 4. Which node labels actually exist?
kubectl get nodes --show-labels

# 5. Show recent namespace events in time order.
kubectl get events -n <namespace> --sort-by=.metadata.creationTimestamp
```

Use describe early because scheduler failures are reported as Pod events. Compare the selector's exact keys and values with node labels, and remember that label matching is case-sensitive. If the NODE column is already populated, stop blaming the scheduler for later startup failures.

Official references

Kubernetes Scheduler explains the filtering, scoring, and binding overview.

Assigning Pods to Nodes documents nodeSelector, nodeName, and the more advanced placement mechanisms covered next in the series.

Pod API reference is the authoritative field reference for spec.nodeSelector and spec.nodeName.

Clean up the demonstration

```bash
kubectl delete namespace cka-scheduling
kubectl label node "$TARGET_NODE" workload- accelerator-
```

Deleting the namespace removes all demonstration workloads. The second command removes the two custom node labels; the trailing hyphen is kubectl's label-removal syntax.