Omar Latrach

TECHNICAL ARTICLE

PUBLISHED · SEPTEMBER 21, 2026

Taints and Tolerations

Understand how node taints repel Pods, how matching tolerations remove that barrier, and how NoSchedule, PreferNoSchedule, and NoExecute change placement and eviction behavior.

This is Learn post 12 of the 54-post Certified Kubernetes Administrator (CKA) preparation path. Post 11 showed how the scheduler filters nodes and how a nodeSelector narrows the eligible set. Taints add the opposite kind of signal: a node can repel Pods that have not explicitly declared that they tolerate its condition or purpose.

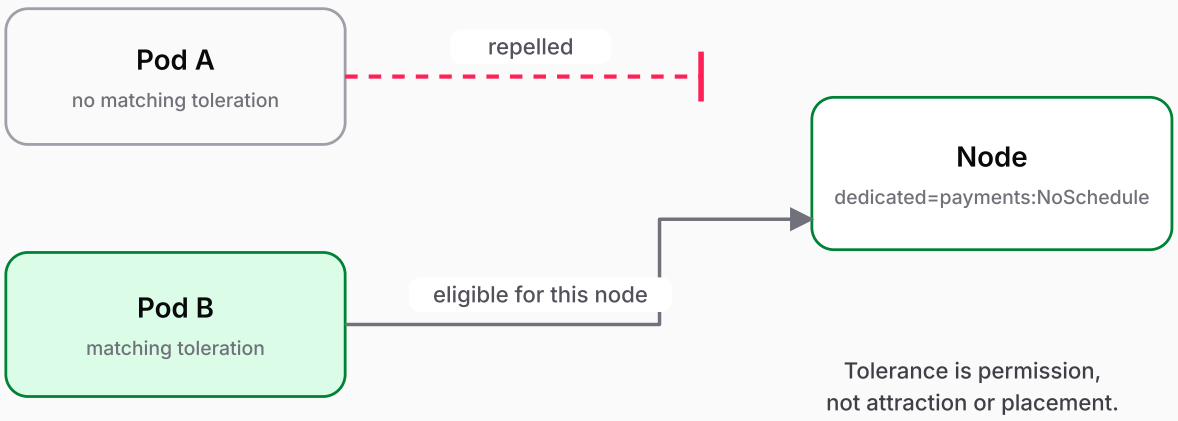
This mechanism protects specialized or restricted nodes from unsuitable workloads and can also remove Pods from a node under specific conditions. The essential skill is to reason from the node's taints and the Pod's tolerations to the resulting scheduling or eviction behavior.

What you'll learn

- Read a taint as a key, optional value, and effect on a node.
- Write Equal and Exists tolerations that match deliberately.
- Predict the different behavior of NoSchedule, PreferNoSchedule, and NoExecute.
- Observe taints, tolerations, FailedScheduling events, and assignments with kubectl.
- Separate permission to use a node from attraction to that node.

The mental model: repel, then permit

> A taint says “keep out.” A matching toleration says “this Pod may pass.” It does not say “place this Pod here.”



A toleration removes one scheduling barrier

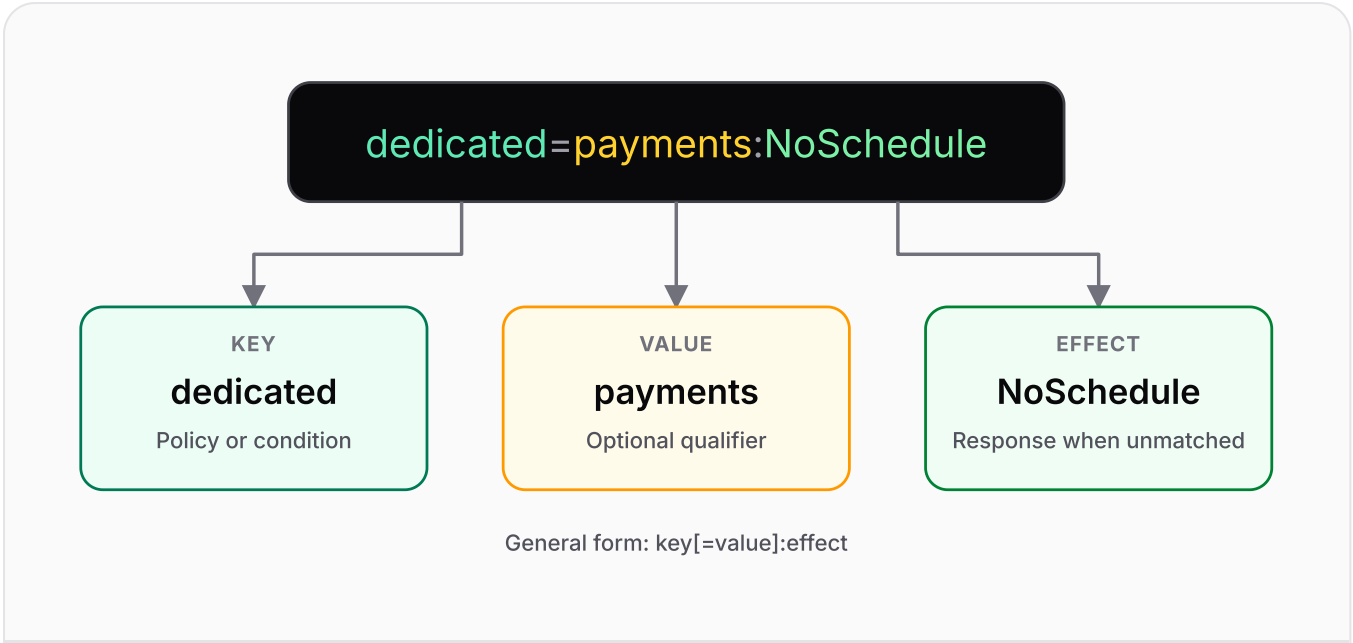
The scheduler ignores a node taint only when the Pod has a matching toleration. Passing that check makes the node possible, not mandatory.

Taints belong to Nodes. Tolerations belong to Pods. During scheduling, Kubernetes compares the candidate node's taints with the incoming Pod's tolerations. Each matching toleration neutralizes one matching taint for that Pod; any relevant taints left unmatched keep their effects.

This is why a toleration alone never reserves or selects a node. After the taint check passes, resource requests, node selectors, and all other scheduler rules still apply. Post 13 will add richer attraction and spreading rules.

A taint has three parts

The kubectl form is key=value:effect. The key identifies the condition or policy, the optional value refines it, and the effect defines what Kubernetes does when a Pod lacks a matching toleration.



Anatomy of a node taint

A taint uses the form key[=value]:effect. The value is optional; the effect determines the consequence of an unmatched taint.

Administrators add and remove node taints with kubectl taint. Use an exact, meaningful key and effect so the policy is easy to inspect later.

```bash
kubectl taint node worker-1 dedicated=payments:NoSchedule
kubectl taint node worker-1 dedicated=payments:NoSchedule-
```

The trailing hyphen in the second command removes that taint; it is not part of the effect name. Removing a taint changes future scheduling decisions but does not move Pods that are already bound.

A toleration must match the taint

A toleration matches when the key and effect match and its operator accepts the taint's value. Equal requires the values to match. Exists ignores the value and matches any taint value with the same key and effect.

```yaml
# Exact value match
tolerations:
  - key: dedicated
    operator: Equal
    value: payments
    effect: NoSchedule
```

```yaml
# Match any value for this key and effect
tolerations:
  - key: dedicated
    operator: Exists
    effect: NoSchedule
```

Equal is the default operator when operator is omitted, but writing it explicitly makes teaching and review clearer. Do not set value with Exists. Also remember that effect participates in matching: tolerating dedicated=payments:NoSchedule does not automatically tolerate the same key and value with NoExecute.

A node can have several taints and a Pod can have several tolerations. Think of the comparison as subtraction: remove every taint matched by a toleration, then evaluate the effects of what remains. One unmatched NoSchedule or NoExecute taint is enough to prevent new placement on that node.

The three effects answer different questions

NoSchedule: reject new placement

The scheduler does not place a new Pod on the node while an unmatched NoSchedule taint remains. Pods already running there are not evicted. This is a hard scheduling filter.

PreferNoSchedule: avoid when practical

The scheduler tries to avoid a node with an unmatched PreferNoSchedule taint, but the rule is soft. The Pod can still be placed there when the overall scheduling decision favors or requires it. Existing Pods are not evicted.

NoExecute: reject new placement and affect bound Pods

An unmatched NoExecute taint prevents new placement and triggers eviction of Pods already bound to the node. A matching toleration without a time limit lets the Pod remain while the taint exists. A matching toleration with tolerationSeconds grants a limited stay.

```yaml
tolerations:
  - key: maintenance
    operator: Equal
    value: planned
    effect: NoExecute
    tolerationSeconds: 300
```

If maintenance=planned:NoExecute appears on the node, this Pod may remain bound for 300 seconds. If the taint is removed before the period expires, eviction is avoided. The time limit applies only to NoExecute tolerations.

> NoSchedule changes eligibility for new placement. NoExecute can change the fate of Pods already on the node.

Worked demonstration: protect one labeled node

Use a disposable cluster with at least one schedulable worker. The label from post 11 confines both demonstration Pods to one known node, so the taint—not some unrelated candidate node—explains the different outcomes.

```bash
kubectl create namespace cka-taints
kubectl get nodes -o wide
export TARGET_NODE=<worker-node-name>
kubectl label node "$TARGET_NODE" training=taints
kubectl taint node "$TARGET_NODE" dedicated=payments:NoSchedule
```

Inspect the stored node configuration instead of relying on memory. The custom column shows the taint list exactly as the API holds it.

```bash
kubectl get node "$TARGET_NODE" \
  -o custom-columns='NAME:.metadata.name,TAINTS:.spec.taints'
kubectl describe node "$TARGET_NODE"
```

In describe output, find the Taints line. It should include dedicated=payments:NoSchedule. The label makes the node the only candidate for these examples; the taint decides whether each Pod may use that candidate.

A Pod without the toleration remains unassigned

blocked.yaml · yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: blocked
  namespace: cka-taints
spec:
  nodeSelector:
    training: taints
  containers:
    - name: web
      image: nginx:1.27
```

```bash
kubectl apply -f blocked.yaml
kubectl get pod blocked -n cka-taints -o wide
kubectl describe pod blocked -n cka-taints
```

The Pod should be Pending with an empty NODE column. Its events explain that the labeled candidate had an untolerated taint. Counts and wording vary with cluster size, but FailedScheduling and untolerated taint are the evidence that matters.

```plaintext
Type     Reason            From               Message
Warning  FailedScheduling  default-scheduler  0/3 nodes are available: 1 node(s) had untolerated taint {dedicated: payments}, 2 node(s) didn't match Pod's node affinity/selector.
```

A matching toleration removes the barrier

allowed.yaml · yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: allowed
  namespace: cka-taints
spec:
  nodeSelector:
    training: taints
  tolerations:
    - key: dedicated
      operator: Equal
      value: payments
      effect: NoSchedule
  containers:
    - name: web
      image: nginx:1.27
```

```bash
kubectl apply -f allowed.yaml
kubectl get pods -n cka-taints \
  -o custom-columns='NAME:.metadata.name,NODE:.spec.nodeName,PHASE:.status.phase'
```

allowed can bind to TARGET_NODE because its toleration matches all three relevant parts of the taint. blocked remains unassigned. The Pods differ only in permission to pass the taint filter.

```plaintext
NAME      NODE       PHASE
allowed   worker-1   Running
blocked   <none>     Pending
```

The node name is illustrative; your output shows TARGET_NODE. Remove nodeSelector from allowed and the toleration would still permit the tainted node, but it would not attract the Pod there. The scheduler could choose any other feasible node.

Tolerations in workload controllers

A Deployment, DaemonSet, StatefulSet, Job, or CronJob does not run on a node itself. Put tolerations in the Pod template so every created Pod receives them.

```yaml
spec:
  template:
    spec:
      tolerations:
        - key: dedicated
          operator: Equal
          value: payments
          effect: NoSchedule
      containers:
        - name: app
          image: nginx:1.27
```

Updating a controller's Pod template creates the desired tolerations on replacement Pods; it does not rewrite immutable fields on existing Pods in place. For a Deployment, a template change triggers a rollout because the Pod template changed.

NoExecute deserves operational caution

Adding a NoExecute taint is not a harmless scheduling experiment: it can evict every bound Pod that lacks a matching toleration. Before applying one, list the node's Pods and inspect critical workloads. Use only a disposable lab node for demonstrations.

```bash
kubectl get pods -A --field-selector spec.nodeName=<node-name>
kubectl get pod <pod-name> -n <namespace> -o yaml
```

When a controller-managed Pod is evicted, its controller can create a replacement, which then needs another eligible node. A standalone Pod has no workload controller to restore it. The taint mechanism changes Pod eligibility or continued binding; controllers separately maintain desired state.

Kubernetes also manages taints

Not every taint was added by an administrator. The control plane represents several node conditions with well-known taints such as node.kubernetes.io/not-ready, node.kubernetes.io/unreachable, node.kubernetes.io/memory-pressure, and node.kubernetes.io/disk-pressure.

The scheduler evaluates those taints when considering new Pods. NoExecute taints for not-ready and unreachable nodes also participate in taint-based eviction. Kubernetes normally adds 300-second tolerations for those two conditions to Pods unless they already specify them, allowing brief node failures to recover before eviction.

Many kubeadm clusters also protect control-plane nodes with node-role.kubernetes.io/control-plane:NoSchedule. Inspect the taint and understand why ordinary workloads remain off that node; do not remove it casually to make a Pending Pod disappear.

```bash
kubectl get nodes \
  -o custom-columns='NAME:.metadata.name,TAINTS:.spec.taints'
kubectl describe node <node-name>
```

Tainting, cordoning, and selecting are different controls

A taint is selective: Pods with matching tolerations can pass. Cordoning a node marks it unschedulable for ordinary new workloads; it is an administrative availability switch rather than a per-workload permission rule. Neither NoSchedule nor cordon evicts existing Pods by itself.

A nodeSelector constrains placement by requiring node labels; a toleration only stops a matching taint from repelling the Pod. To dedicate nodes in both directions, administrators commonly combine a taint and toleration with a label-based placement rule. Post 13 expands the placement side with node affinity.

Taints are scheduling and eviction controls, not security boundaries. They do not authorize access, isolate network traffic, or prevent a privileged actor from changing a Pod specification.

The nodeName exception from post 11

A manually supplied spec.nodeName bypasses scheduler filtering, so a NoSchedule taint does not stop that direct assignment. A NoExecute taint still affects the bound Pod and can trigger eviction when no matching toleration exists. This is another reason nodeName is not a routine placement tool.

A compact investigation workflow

When a Pod is unassigned, compare both sides of the relationship: the candidate nodes' taints and the Pod's stored tolerations. Then read events for the scheduler's conclusion.

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

# What tolerations does the stored Pod have?
kubectl get pod <pod-name> -n <namespace> -o yaml

# What taints are on the nodes?
kubectl get nodes -o custom-columns='NAME:.metadata.name,TAINTS:.spec.taints'

# What did the scheduler report?
kubectl describe pod <pod-name> -n <namespace>
kubectl get events -n <namespace> --sort-by=.metadata.creationTimestamp
```

Match key, value, operator, and effect carefully. If all taints are tolerated but the Pod remains unassigned, another filter—such as nodeSelector or resource fit—can still be responsible.

Official references

Taints and Tolerations documents matching, effects, multiple-taint evaluation, condition taints, and taint-based eviction.

kubectl taint reference provides the supported command syntax for adding, replacing, selecting, and removing node taints.

Clean up the demonstration

```bash
kubectl delete namespace cka-taints
kubectl taint node "$TARGET_NODE" dedicated=payments:NoSchedule-
kubectl label node "$TARGET_NODE" training-
```

Delete the workload namespace first, then remove the demonstration taint and label. Confirm the node's Taints field no longer contains dedicated=payments:NoSchedule.