BY FENIL GAJJAR

KUBERNETES DAILY TASKS

KUBERNETES
TAINTS TOLERATIONS & NODE SELECTORS

- COMPEREHENSIVE GUIDE
- THEORY + PRACTICAL TASKS
- REAL TIME SCENARIO TASKS
- END TO END DOC

100% SOLID

☎ CONTACT US

fenilgajjar.devops@gmail.com
linkedin.com/in/fenil-gajjar
gitub.com/Fenil-Gajjar

🚀 Mastering Kubernetes: Taints, Tolerations, Labels & Selectors 🎯 | From Core Concepts to Real-World Deployments ⚙️

👋 Welcome!

Hey there!
Thank you so much for stopping by and exploring this detailed doc on **Kubernetes Taints, Tolerations, Node Labels & Selectors**. 🚀

Whether you're a DevOps engineer, SRE, platform engineer, or a curious learner diving into Kubernetes infrastructure, you're in the right place! 🙌

💬 About This Doc

This isn't just theory — this guide is a **deep dive from fundamentals to production-grade use cases**. We'll walk through:

- 🎓 **Understanding** what taints, tolerations, labels, and selectors really do under the hood.
- ⚙️ **How they work together** to influence pod scheduling, isolation, and workload control.
- 🛠️ **Real-world implementation tasks** using `kops`-based Kubernetes clusters.
- 🚑 **Troubleshooting tips** and 🧠 **best practices** you can apply in enterprise-grade clusters.

If you're managing a multi-tenant cluster, deploying ML workloads, isolating nodes for platform services, or just trying to cleanly control where your pods land — this doc is for you.

📚 Part of a Bigger Series

This document is part of my growing DevOps content series where I share hands-on knowledge, practical scenarios, and production patterns across:

- ☁️ **AWS**
- 🔧 **Ansible**
- ☸️ **Kubernetes**
- 📦 **Infrastructure as Code**
- And more cloud-native tooling and practices

You can follow my content, series updates, code samples, and more on:

🔗 **LinkedIn**: linkedin.com/in/fenil-gajjar
🔗 **GitHub**: *github.com/Fenil-Gajjar*

🎯 Stay Tuned!

Thank you again for reading — and **extra gratitude** if you've checked out my previous docs as well. Your support means a lot! 🙏

📥 **More deep dives are coming soon**, so stay connected and get ready for more content just like this — clean, practical, and made for real DevOps engineers.

Let’s keep learning and building together.

Happy Clustering! ☸️🔥

📌 What Are Taints and Tolerations in Kubernetes?

In Kubernetes, **taints** and **tolerations** work together to **control which pods can be scheduled on which nodes**. This mechanism provides **fine-grained control** over **pod placement** by allowing certain nodes to repel pods unless those pods explicitly "tolerate" the condition.

🔹 Taints – Think of them as Node-Level Restrictions

A **taint** is applied to a node. It marks the node as **unsuitable for general scheduling**. Once a node is tainted, **no pods** will be scheduled onto it **unless they have a matching toleration**.

A taint consists of three parts:

```
<key>=<value>:<effect>
```

Example:

```
key=env, value=prod, effect=NoSchedule
```

This means:

> Don't schedule any pod on this node unless it tolerates a taint with key `env`, value `prod`, and effect `NoSchedule`.

🔹 Tolerations – Think of them as Pod-Level Permissions

A **toleration** is applied to a **pod**. It tells Kubernetes:

> “I am okay with being scheduled on a node that has a matching taint.”

Tolerations don’t force scheduling onto tainted nodes. They just allow it.

YAML example:

```
tolerations:

  - key: "env"

    operator: "Equal"

    value: "prod"

    effect: "NoSchedule"
```

🔹 Effects of Taints

There are **three types of effects** that define how Kubernetes should treat a taint:

Effect	Behavior
`NoSchedule`	Do **not** schedule pods on the node unless they **tolerate** the taint

`PreferNoSchedule`	Try to avoid scheduling pods here, but not strictly enforced
`NoExecute`	Existing pods that don’t tolerate this taint will be **evicted**

🔸 Use Cases (When and Why)

- **Dedicated nodes**: Isolate workloads (e.g., run only GPU workloads or sensitive apps).
- **Environment isolation**: Separate `dev`, `test`, and `prod` workloads.
- **Cluster maintenance**: Temporarily cordon nodes for draining or updates.
- **Spot instances**: Taint ephemeral/spot instances to prevent critical workloads from landing there.

🔍 Why Do We Need Taints and Tolerations in Kubernetes?

Kubernetes by default tries to **utilize all nodes evenly** and will schedule pods wherever there are available resources. But in real-world clusters, **not all nodes are created equal**—and **not all workloads should run everywhere**.

That's where **taints and tolerations** come in. They give platform engineers the ability to **control where pods can and cannot run**. This is **not a resource constraint**, but rather a **policy constraint**.

⚠️ Without Taints and Tolerations:

Imagine a cluster where:

- Some nodes are **GPU-enabled** for ML workloads.
- Some nodes are **spot instances**, cheaper but **less reliable**.
- Some nodes are **reserved for production traffic only**.

If there's no control:

- Regular, low-priority workloads could land on **expensive GPU nodes**.
- Critical production pods could be scheduled on **spot nodes** and lost.
- Dev/test workloads might run on **production-isolated** nodes.

This leads to:

- **Wasted resources**
- **Performance issues**
- **Security/risk exposure**
- **Unpredictable behavior**

✅ With Taints and Tolerations:

Taints let you **mark nodes** for specific purposes.
Tolerations let **only approved pods** be scheduled on those nodes.

This results in:

- **Workload isolation** (e.g., dev vs. prod)
- **Node specialization** (e.g., GPU, memory-optimized, ARM nodes)
- **Improved reliability** (e.g., avoid unreliable nodes)
- **Policy enforcement** without tight coupling to pod specs

⚙️ How Do Taints and Tolerations Work in Kubernetes?

At a high level, **taints and tolerations work together** to allow **node-level filtering**. They don't assign pods to specific nodes — instead, they define **where pods are not allowed** to run, unless they **tolerate** the restriction.

Let’s break it down in detail:

🔸 1. Taints Are Applied on Nodes

A **taint** is a property you attach to a **node** to indicate that it should repel all pods — **except those that can tolerate it**.

The taint uses this format:

```
<key>=<value>:<effect>
```

Example:

```
kubectl taint nodes node-1 environment=prod:NoSchedule
```

This command applies a taint with:

- `key`: `environment`
- `value`: `prod`

- `effect: NoSchedule`

Effect types:

Effect	Description
`NoSchedule`	Pods that do not tolerate the taint **will not be scheduled** on the node.
`PreferNoSchedule`	Kubernetes will try to avoid scheduling non-tolerating pods on the node, but **won’t strictly block them**.
`NoExecute`	Not only blocks scheduling — it **evicts running pods** that don’t tolerate the taint.

🔸 2. Tolerations Are Applied on Pods

A **toleration** is specified in the pod's manifest. It signals that the pod **understands and accepts** the taint on a node and is willing to run there.

Example YAML:

```
apiVersion: v1

kind: Pod

metadata:

  name: nginx
```

```
spec:

  containers:

  - name: nginx

    image: nginx

  tolerations:

  - key: "environment"

    operator: "Equal"

    value: "prod"

    effect: "NoSchedule"
```

This pod will **tolerate the taint** we applied to `node-1`.

Important note:
Tolerations don’t force Kubernetes to place a pod on a tainted node — they **only allow it**.

🔍 How Kubernetes Uses This Mechanism Internally

When Kubernetes schedules a pod:

1. It looks at **all available nodes**.

2. It filters out any node with a taint that the pod **does not tolerate**.

3. From the remaining nodes, it uses other scheduling policies (like resource availability, affinity, etc.) to pick a node.

So:

- **No matching toleration → pod excluded from tainted node**
- **Matching toleration → node considered as a valid candidate**

🧪 Example: Combined Flow

Let's say we have:

Node A:

```
kubectl taint nodes node-a app=ml:NoSchedule
```

Pod:

```
tolerations:
- key: "app"
  operator: "Equal"
  value: "ml"
  effect: "NoSchedule"
```

👉 Result: Pod **can** be scheduled onto `node-a`.

⚠️ Using `NoExecute` with TolerationSeconds

`NoExecute` is a powerful taint effect. It doesn't just block new pods — it will **evict running ones** if they don't tolerate it.

You can also specify how **long** a pod is allowed to stay after the taint appears.

```
tolerations:
- key: "maintenance"
  operator: "Equal"
  value: "true"
  effect: "NoExecute"
  tolerationSeconds: 300
```

➡️ This pod will stay on the tainted node for 300 seconds, then be evicted.

🔧 How to Add/Remove Taints

Add a taint:

```
kubectl taint nodes node-1 key=value:effect
```

Remove a taint:

```
kubectl taint nodes node-1 key:effect-
```

Example:

```
kubectl taint nodes node-1 environment=prod:NoSchedule

kubectl taint nodes node-1 environment:NoSchedule-  #
Removes it
```

🔍 Operator Field in Tolerations

The `operator` field controls how the key/value pair is interpreted.

Operator	Description
`Equal`	Key and value must **both** match
`Exists`	Only the key needs to exist (value ignored)

Example with `Exists`:

```
tolerations:

- key: "dedicated"

  operator: "Exists"
  effect: "NoSchedule"
```

This pod tolerates **any taint** with key `dedicated` — regardless of value.

🧭 How Does Scheduling Work in Kubernetes (in the Context of Taints and Tolerations)?

To answer it simply:

> **Tolerations alone do *not* guarantee that a pod will be scheduled onto a tainted node.**

They only **allow** scheduling onto that node — they **don't force** or **prioritize** it.

Let's break it down in stages:

🔹 1. The Scheduler's Job

Kubernetes has a **default scheduler** that is responsible for:

- Filtering nodes where a pod *can run*
- Scoring the remaining nodes
- Picking the most appropriate one based on scores

The scheduler evaluates:

- Taints & tolerations
- Node affinity / anti-affinity
- Resource requests (CPU/memory)
- NodeSelector or node pools
- Topology spread constraints
- Custom scheduler logic, if configured

🔸 2. Filtering Stage: Taints Come First

During the **filtering** phase, taints play a crucial role.

- **If a node has a taint** that a pod **does not tolerate →** ✅ Node is *excluded* from the candidate list.
- **If a node has a taint** that a pod **does tolerate →** ✅ Node is *included* in the candidate list.
- Nodes **without taints →** ✅ Always included.

👉 Taints are a **gatekeeper**. Tolerations are like **keys** that open those gates.

🔸 3. Scoring Stage: Tolerations Are NOT a Factor

Once Kubernetes has a filtered list of nodes where the pod *can* be scheduled:

- It assigns scores to those nodes based on resource availability, topology, affinity rules, etc.
- **Taints/tolerations have no impact here.**
- The pod could still be scheduled on a *non-tainted* node if it scores higher.

❌ So What Doesn’t Happen?

If a pod has a toleration for a taint:

- Kubernetes **won’t prefer** tainted nodes.
- Kubernetes **won’t guarantee** that the pod will run there.
- It might never run on the tainted node **unless no better nodes are available**.

This often surprises people.

✅ How to Ensure a Pod Runs on a Tainted Node?

If you want a pod to be *specifically scheduled* onto a tainted node, you **combine**:

1. **Taint on the node**
2. **Toleration on the pod**
3. **NodeSelector or node affinity** on the pod to *target* the desired node

Example: Guaranteed Scheduling onto a Tainted Node

Taint the node:

```
kubectl taint nodes gpu-node gpu=true:NoSchedule
```

Pod manifest:

```
spec:
  nodeSelector:
    gpu: "true"
  tolerations:
  - key: "gpu"
    operator: "Equal"
    value: "true"
    effect: "NoSchedule"
```

Now the pod can *only* run on nodes with label `gpu=true`, and it tolerates the taint ➔ **it will be scheduled there.**

🔐 Why Workloads Don't Get Scheduled on Control Plane Nodes by Default?

Yes, **Kubernetes control plane nodes come with a default taint**, which **prevents regular workloads from being scheduled** on them.

This is a deliberate architectural choice to **protect the stability and performance of the cluster**.

🧠 Deep Dive: The Justification

The **control plane** is the **brain of the Kubernetes cluster**. It runs critical components like:

- `kube-apiserver`
- `etcd` (Kubernetes' database)
- `kube-scheduler`
- `kube-controller-manager`

- Cloud controller and DNS (in many setups)

These components must always be **highly available**, **lightweight**, and **protected**.

If user workloads were to be scheduled on the same nodes:

- Resource starvation (CPU/memory) could bring down `etcd` or `kube-apiserver`
- Cluster-wide instability could follow
- Even a misbehaving pod could block access to the whole cluster

That's why Kubernetes **by design** applies a **default taint** to control plane nodes.

📌 The Default Taint on Control Plane Nodes

On most Kubernetes installations (including `kubeadm`, `kops`, EKS, etc.), control plane nodes have the following default taint:

```
node-role.kubernetes.io/control-plane=:NoSchedule
```

Or, in older setups:

```
node-role.kubernetes.io/master=:NoSchedule
```

This taint:

- Has no key or value
- Has `NoSchedule` effect
- Repels any pod **unless it has a matching toleration**

🔎 How You Can Verify This

Run:

```
kubectl describe node <control-plane-node-name>
```

You’ll see a taint like:

```
Taints:
  node-role.kubernetes.io/control-plane:NoSchedule
```

Or with `kubectl get nodes -o json`:

```
"taints": [
  {
    "effect": "NoSchedule",
    "key": "node-role.kubernetes.io/control-plane"
  }

]
```

👷 Can You Schedule Pods on Control Plane Nodes?

Yes — but you must **explicitly override** the taint.

To do that, the pod must include a **toleration** for the taint:

```
tolerations:

- key: "node-role.kubernetes.io/control-plane"

  effect: "NoSchedule"

  operator: "Exists"
```

But even with this toleration, **you also need to steer the pod** to that node (since Kubernetes will not *prefer* tainted nodes). So you typically also use:

- `nodeSelector`
- or `nodeName`

Example:

```
spec:

  tolerations:

  - key: "node-role.kubernetes.io/control-plane"

    effect: "NoSchedule"

    operator: "Exists"

  nodeSelector:

    node-role.kubernetes.io/control-plane: ""
```

⚠️ Should You Do This in Production?

Strongly discouraged in production environments unless:

- You're on a **very small dev/test cluster**
- Or running **high-availability control planes** with sufficient isolation

In production:

- Keep control plane clean
- Keep workloads off it
- Use taints + separate node pools to control workload placement

🎯 Taint Effects in Kubernetes

Before diving into each, here's a quick reference of the three effects:

Effect Type	What it Does
`NoSchedule`	Prevents scheduling of non-tolerating pods on a node
`PreferNoSchedule`	Tries to avoid scheduling, but allows it if necessary
`NoExecute`	Prevents scheduling **and** evicts already running non-tolerating pods

Now, let's go deep into:

🔍 `NoSchedule` — Deep Dive

📌 What is `NoSchedule`?

The `NoSchedule` effect is used to **strictly prevent pods from being scheduled on a node**, unless those pods **explicitly tolerate** the taint.

> ⛔ If a pod does *not* tolerate the taint ➜ it will *never* be scheduled on that node.

🧠 When to Use `NoSchedule`

Use `NoSchedule` when you want to **enforce a hard boundary** between workloads and nodes. This is the most deterministic and strict form of scheduling exclusion.

✅ Common real-world examples:

- Keep dev workloads off prod nodes
- Ensure only GPU-intensive apps run on GPU nodes
- Prevent pods from landing on spot or preemptible instances unless intended
- Reserve nodes for specific tenants or environments

🛠️ Syntax and Example

Taint a node:

```
kubectl taint nodes node-a dedicated=prod:NoSchedule
```

This means:
👉 "Only pods that tolerate `dedicated=prod` with effect `NoSchedule` can be scheduled here."

Now the pod needs this toleration to be allowed:

```
tolerations:
- key: "dedicated"
  operator: "Equal"
  value: "prod"
  effect: "NoSchedule"
```

➡️ This pod **now tolerates** the taint and *may* be scheduled on that node.

❌ What Happens if Pod Lacks the Toleration?

- The Kubernetes scheduler sees the node
- Filters it **out** from consideration
- Pod won’t be scheduled there — **ever**

No retries, no random behavior — this is a **hard rule**.

⚙️ Real-World Scenario Example

Problem:

You want to reserve certain nodes (`node-group-prod`) **strictly for production traffic only**.

Solution:

Taint the nodes:

```
kubectl taint nodes node-group-prod
dedicated=prod:NoSchedule
```

Add toleration to prod-only workloads:

```
tolerations:
- key: "dedicated"
  operator: "Equal"
  value: "prod"
  effect: "NoSchedule"
```

(Optional) Add a `nodeSelector`:

```
nodeSelector:
```

1. `dedicated: prod`

🌥️ `PreferNoSchedule` — Deep Dive

📌 What is `PreferNoSchedule`?

`PreferNoSchedule` is a **soft taint** — it tells Kubernetes:

> “Try not to schedule pods here **if possible**, but it’s okay if there’s no better alternative.”

It is **advisory**, not mandatory.

🧠 When to Use `PreferNoSchedule`?

Use `PreferNoSchedule` when you want to:

- **Discourage** scheduling pods onto certain nodes
- Apply **soft separation** instead of hard enforcement
- Create **preference rules** without blocking workloads

✅ Common use cases:

- Signal that a node is lightly reserved for a type of workload (e.g., batch jobs)
- Prefer that test pods avoid production nodes, but allow them during high resource pressure

- Let Kubernetes make the final decision if needed, while guiding its preference

🛠️ Syntax and Example

Apply the taint:

```
kubectl taint nodes node-a workload=batch:PreferNoSchedule
```

This means:
👉 "Avoid scheduling here unless no other suitable nodes are available."

Pod Manifest (with or without toleration):

- 🔹 Pod **without** toleration:
  - Kubernetes **will try** to avoid the tainted node.
  - If **no other nodes are available**, it **can still schedule** here.

🔹 Pod **with** toleration:

```
tolerations:
- key: "workload"
  operator: "Equal"
  value: "batch"
  effect: "PreferNoSchedule"
```

-
    - This doesn’t force scheduling either.
    - It only says, “I’m okay running on this node if needed.”

📊 Behind the Scenes: How It Works

- The Kubernetes scheduler filters nodes with `NoSchedule` taints **out entirely** if not tolerated.
- But for `PreferNoSchedule`:
    - All nodes are kept in the candidate pool.
    - Nodes with `PreferNoSchedule` are given a **lower score**.
    - Scheduler still considers them but tries to choose others first.

This makes it ideal for **low-priority guidance**.

⚠️ Important Notes

- If **all nodes are tainted with `PreferNoSchedule`**, the pod **will be scheduled anyway**.
- It **does not protect** the node as strictly as `NoSchedule`.
- It is not suitable when **guaranteed isolation** is required.

🧪 Real-World Scenario

Problem:

You want to **prioritize critical apps** on certain nodes, but allow non-critical apps to run there **only if no other nodes are available**.

Solution:

Taint the preferred node group:

```
kubectl taint nodes prod-nodes
critical=true:PreferNoSchedule
```

1. **Deploy normal workloads without any toleration.**

Result:

- Scheduler avoids `prod-nodes` for normal apps.

- But during heavy load, it will still schedule them if needed.

🧠 Advanced Tip: Combine with Pod Priorities

`PreferNoSchedule` becomes powerful when used alongside **Pod PriorityClasses**.

- Critical pods ➜ tolerate the taint and have higher priority
- Low-priority pods ➜ no toleration, and get scheduled elsewhere

This allows you to implement **best-effort node preferences** based on workload importance.

🔥 `NoExecute` — Deep Dive

📌 What is `NoExecute`?

`NoExecute` is the **strictest and most aggressive taint effect** in Kubernetes.

It does two things:

1. **Prevents scheduling** of new pods on the tainted node (like `NoSchedule`)
2. **Evicts already running pods** from the node if they don't tolerate the taint

> In short: `NoExecute` both **blocks new pods** and **kicks out existing ones** unless they have a toleration.

⚠️ Why It's Powerful (and Dangerous)

This effect is used when:

- You want to **forcefully evacuate** a node (e.g. for maintenance or failures)
- You want to **ensure only specific pods can ever run or stay** on that node
- You want to **respond dynamically to node health conditions** (e.g. taints from the node controller)

It’s also used automatically by Kubernetes under certain system conditions — more on that below.

🧠 Behavior Summary

Pod State	Has Toleration	Effect
Not yet scheduled	❌	Pod won’t be scheduled
Not yet scheduled	✅	Pod may be scheduled
Already running	❌	Pod gets **evicted immediately**
Already running	✅	Pod **stays**, until toleration duration expires (if set)

🛠️ Syntax and Example

Taint a node:

```
kubectl taint nodes node-a role=critical:NoExecute
```

This taint means:

> “Only pods that explicitly tolerate this taint will be allowed to run or stay.”

Pod With Infinite Toleration:

```
tolerations:
- key: "role"
  operator: "Equal"
  value: "critical"
  effect: "NoExecute"
```

This pod:

- Will be allowed to **stay running**
- Will be allowed to **be scheduled**

Pod With Temporary Toleration:

```
tolerations:
- key: "role"
  operator: "Equal"
  value: "critical"
  effect: "NoExecute"
  tolerationSeconds: 60
```

This pod:

- Will be **evicted after 60 seconds**
- Useful for **graceful shutdown**, not instant eviction

Pod With No Toleration:

- Will be **evicted immediately** if already running
- Will **not be scheduled** onto the tainted node at all

🚨 Kubernetes Itself Uses `NoExecute`

Kubernetes uses `NoExecute` taints internally, such as:

Node NotReady Taint

```
node.kubernetes.io/not-ready:NoExecute
```

1.
    - Applied when a node becomes unreachable or unresponsive.
    - Non-tolerating pods are evicted.

Node Unreachable Taint

```
node.kubernetes.io/unreachable:NoExecute
```

2.
    - Used when the kubelet stops reporting.
    - Scheduler evicts pods if not tolerated.

In both cases, if you want your pods to **remain** during transient failures, add tolerations like:

```
tolerations:
- key: "node.kubernetes.io/unreachable"
  operator: "Exists"
  effect: "NoExecute"
  tolerationSeconds: 300
```

This pod will remain for 5 minutes, allowing time for the node to recover.

🧪 Real-World Use Case

Scenario:

You're managing a **dedicated compute node** in a data-intensive cluster. Only long-running analytics workloads should run there, and **they must be drained quickly** when needed.

Solution:

Taint the node with NoExecute:

```
kubectl taint nodes analytics-node batch=only:NoExecute
```

Analytics workloads tolerate it:

```
tolerations:
- key: "batch"
  value: "only"
  effect: "NoExecute"
```

All other pods get instantly evicted if mistakenly scheduled.

🎯 When & Why Do We Use Taints and Tolerations in Production Kubernetes Clusters?

Taints and tolerations are **production-grade scheduling controls**. They are used to:

- Protect infrastructure components
- Isolate sensitive workloads
- Optimize resource usage
- Enforce multi-tenancy and security boundaries

Let’s explore **real production scenarios** where these are used — with practical justification.

🔒 1. Protecting Critical Infrastructure Nodes

🧩 Situation:

You have dedicated nodes for:

- Control plane components

- Ingress controllers
- Core DNS
- Monitoring (Prometheus, Loki)

🎯 Why Taints?

You **don't want** regular app workloads or dev/test pods running on these nodes — they can starve system resources or impact reliability.

✅ Example:

```
kubectl taint nodes node-monitor
dedicated=monitoring:NoSchedule
```

Now only Prometheus pods that tolerate this taint will be scheduled here.

🔄 2. Node-Level Isolation for Different Environments

🧩 Situation:

You run **multiple environments** (dev, staging, prod) in the **same cluster**, but you want each to use its own node pool.

🎯 Why Taints?

- Prevent noisy neighbor issues
- Enforce performance separation
- Reduce risk of cross-env impact

✅ Example:

```
kubectl taint nodes prod-nodes env=prod:NoSchedule
```

Then prod-only workloads get a toleration like:

```
tolerations:
- key: "env"
  value: "prod"
  effect: "NoSchedule"
```

This ensures only prod workloads run on prod nodes.

🎮 3. Prioritizing GPU / Specialized Hardware Nodes

🧩 Situation:

You have a pool of GPU-enabled nodes, or nodes with NVMe or FPGAs.
You want **only ML/AI workloads** to land on them — not regular backend pods.

🎯 Why Taints?

Prevents accidental usage of expensive nodes. Ensures hardware is available for critical jobs.

✅ Example:

```
kubectl taint nodes gpu-node gpu=true:NoSchedule
```

Then your ML training job can tolerate this taint:

```
tolerations:
- key: "gpu"
  value: "true"
  effect: "NoSchedule"
```

🔁 4. Evicting Pods from Unhealthy Nodes

🧩 Situation:

A node becomes `NotReady` or unreachable. You want pods to **automatically be evicted** and rescheduled elsewhere.

🎯 Why Taints?

Kubernetes automatically applies `NoExecute` taints:

- `node.kubernetes.io/unreachable`
- `node.kubernetes.io/not-ready`

You can add tolerations with `tolerationSeconds` to **delay eviction** and allow the node to recover:

```
tolerations:
- key: "node.kubernetes.io/not-ready"
  operator: "Exists"
  effect: "NoExecute"
  tolerationSeconds: 300
```

This keeps pods on the node for 5 minutes during transient issues.

⚖️ 5. Enforcing Pod Placement for Licensing / Compliance

🧩 Situation:

Certain applications (e.g., Oracle DB, SAP) must run on **licensed or pinned hardware nodes**.

🎯 Why Taints?

You can restrict those nodes and **only allow licensed apps** with matching tolerations.

✅ Example:

```
kubectl taint nodes oracle-host license=oracle:NoSchedule
```

Only Oracle pods tolerate this taint.

🛡️ 6. Multi-Tenant Clusters — Tenant Isolation

🧩 Situation:

You run a **multi-team** or **multi-tenant** cluster.
You need to enforce soft or hard isolation between teams for:

- Compliance
- Cost visibility

- Debuggability

🎯 Why Taints?

- Reserve node groups for specific teams
- Prevent cross-team interference
- Combine with RBAC, network policies, and quotas

✅ Example:

```
kubectl taint nodes team-a-node tenant=team-a:NoSchedule
```

Team A workloads get a toleration:

```
tolerations:
- key: "tenant"
  value: "team-a"
  effect: "NoSchedule"
```

🔄 7. Graceful Drain or Rolling Update Strategy

🧩 Situation:

You need to **prepare a node for maintenance** (e.g., OS patching, autoscaling, draining).

🎯 Why Taints?

You can dynamically taint the node with `NoExecute` to begin gracefully evicting workloads.

✅ Example:

```
kubectl taint nodes node-to-update
upgrade=maintenance:NoExecute
```

Pods without tolerations are evicted.

Pods with `tolerationSeconds` get time to shut down cleanly.

🔁 Common Patterns for Using Taints & Tolerations in Real-World Kubernetes Setups

These patterns are drawn from **actual production practices** across organizations that run **multi-tenant**, **resource-intensive**, or **security-sensitive** workloads.

🧍 Multi-Tenant Clusters (Team/Dept/Env Isolation)

🎯 Goal:

Isolate workloads for different:

- Teams
- Environments (prod/stage/dev)
- Customers (in managed platforms)

✅ Pattern:

Taint nodes per tenant/environment:

```
kubectl taint nodes node-1 tenant=team-a:NoSchedule

kubectl taint nodes node-2 tenant=team-b:NoSchedule
```

Workloads tolerate their taints:

```
tolerations:
- key: "tenant"
  value: "team-a"
  effect: "NoSchedule"
```

- Combine with:
    - **Namespaces per tenant**
    - **ResourceQuotas**
    - **NetworkPolicies**
    - **RBAC**

🔐 **Benefit:**

- Prevents tenant pods from running on wrong nodes
- Supports clear boundaries in a shared cluster

🧠 GPU / Specialized Hardware Workload Targeting

🎯 Goal:

Ensure only **ML/AI**, **data processing**, or **accelerated** workloads land on expensive hardware nodes.

✅ Pattern:

Taint GPU nodes:

```
kubectl taint nodes gpu-node accelerator=nvidia:NoSchedule
```

ML workloads tolerate:

```
tolerations:
- key: "accelerator"
  value: "nvidia"
  effect: "NoSchedule"
```

Combine with **NodeAffinity** to target node labels:

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
```

```
      - matchExpressions:
        - key: "kubernetes.io/instance-type"
          operator: In
          values:
          - "gpu-optimized"
```

🔐 **Benefit:**

- Prevents non-GPU pods from consuming costly GPU nodes

🚪 System Component Isolation (Node Isolation)

🎯 **Goal:**

Prevent application pods from sharing nodes with system or critical services (e.g., CoreDNS, Ingress, Prometheus)

✅ **Pattern:**

Taint **infra/system nodes**:

```
kubectl taint nodes system-node dedicated=system:NoSchedule
```

Allow only infra pods to tolerate it:

```
tolerations:
- key: "dedicated"
  value: "system"
  effect: "NoSchedule"
```

- Optionally combine with `nodeSelector` or `affinity`

🔐 **Benefit:**

- Guarantees that system services are isolated and not impacted by user workload spikes

🧪 Batch vs Interactive Workload Segregation

🎯 **Goal:**

Keep **non-critical**, **batch**, or **background jobs** from impacting **interactive** or **latency-sensitive** apps.

✅ **Pattern:**

Taint low-priority batch nodes:

```
kubectl taint nodes batch-node
workload=batch:PreferNoSchedule
```

Let batch workloads tolerate:

```
tolerations:

- key: "workload"

  value: "batch"

  effect: "PreferNoSchedule"
```

- Interactive workloads are **not scheduled** here by default

🔐 **Benefit:**

- Soft isolation with fallback flexibility
- Supports cost-effective autoscaling of batch workers

🛠️ Controlled Node Maintenance / Eviction

🎯 **Goal:**

Drain nodes **gracefully** without disrupting services too quickly

✅ **Pattern:**

During maintenance:

```
kubectl taint node node-x maintenance=planned:NoExecute
```

Let pods decide how long to tolerate:

```yaml
tolerations:
- key: "maintenance"
  value: "planned"
  effect: "NoExecute"
  tolerationSeconds: 180
```

- Use `tolerationSeconds` to allow graceful shutdown

🔐 **Benefit:**

- Clean rolling updates or node patching
- Avoids chaotic evictions and service drops

⚙️ Node Feature-Aware Scheduling

🎯 **Goal:**

Ensure that workloads land on nodes with specific **capabilities**, such as:

- AVX instruction sets
- SSD/NVMe
- High RAM

- Special kernel modules

✅ **Pattern:**

Taint high-capacity nodes:

```
kubectl taint nodes perf-node ssd=true:NoSchedule
```

App pods tolerate + use affinity:

```
tolerations:
- key: "ssd"
  value: "true"
  effect: "NoSchedule"

affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: "disk-type"
          operator: In
          values:
          - "nvme"
```

🔐 Benefit:

- Fine-grained workload placement
- Performance tuning via hardware-awareness

🧱 Taints and Tolerations in YAML — Structure & Syntax

🧊 📍 Taints — Defined on Nodes

Taints are added to **nodes** to **repel** pods unless they tolerate the taint.

✅ Add a taint to a node (CLI):

```
kubectl taint nodes <node-name> <key>=<value>:<effect>
```

🔎 Example:

```
kubectl taint nodes node1 env=prod:NoSchedule
```

This applies the following taint:

```
taints:
- key: "env"
  value: "prod"
  effect: "NoSchedule"
```

> 🧠 This means: “Only pods that tolerate this taint can be scheduled on this node.”

🧊 YAML Representation of Node With Taint

You can also define taints in a node YAML (for static files or custom tools):

```
apiVersion: v1
kind: Node
metadata:
  name: node1
spec:
  taints:
  - key: "env"
    value: "prod"
    effect: "NoSchedule"
```

⚙️ 📦 Tolerations — Defined on Pods

Tolerations are defined **inside pod specs**, allowing pods to be scheduled onto nodes with matching taints.

✅ Minimal YAML Structure

```
tolerations:
- key: "<taint-key>"
  operator: "Equal" | "Exists"
  value: "<taint-value>"   # Optional if operator=Exists
  effect: "NoSchedule" | "PreferNoSchedule" | "NoExecute"
  tolerationSeconds: <int> # Only used with NoExecute
```

🧪 Example: Tolerate a `NoSchedule` Taint

```
tolerations:
- key: "env"
  operator: "Equal"
  value: "prod"
  effect: "NoSchedule"
```

> ✅ This pod can now be scheduled on nodes tainted with `env=prod:NoSchedule`.

🧪 Example: Tolerate Any `NoExecute` Taint for 60 Seconds

```
tolerations:
- key: "node.kubernetes.io/unreachable"
  operator: "Exists"
  effect: "NoExecute"
```

```yaml
  tolerationSeconds: 60
```

✅ This pod will be evicted after 60 seconds of the node becoming unreachable.

🧪 Example: Match Any Taint with `Exists`

```yaml
tolerations:
- key: "env"
  operator: "Exists"
  effect: "NoSchedule"
```

✅ This tolerates any taint with key `env` and effect `NoSchedule`, regardless of value.

📄 Complete Pod Spec Example

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-app
```

```yaml
spec:
  containers:
  - name: app
    image: nginx
  tolerations:
  - key: "env"
    operator: "Equal"
    value: "prod"
    effect: "NoSchedule"
  - key: "maintenance"
    operator: "Equal"
    value: "planned"
    effect: "NoExecute"
    tolerationSeconds: 300
```

✅ This pod can:

- Be scheduled on `env=prod:NoSchedule` nodes

- Stay on `maintenance=planned:NoExecute` nodes for 5 minutes before eviction

🚫 Remove a Taint from a Node (CLI)

```
kubectl taint nodes <node-name> <key>:<effect>-
```

Example:

```
kubectl taint nodes node1 env=prod:NoSchedule-
```

❌ Removes the `env=prod:NoSchedule` taint

📘 Real-Time Kubernetes Task:

Isolate Logging Workloads to Dedicated Nodes Using Taints & Tolerations

🎯 Objective:

In a production-grade Kubernetes cluster (like your Kops-managed one), we want to **isolate logging agents** (e.g., Fluent Bit) to run **only** on specific nodes. These nodes should not run any other workloads.

📦 Use Case:

- You have a DaemonSet (Fluent Bit) that collects logs from each node.
- You want only **dedicated logging nodes** to run the Fluent Bit pods.
- You'll taint these nodes so **no other workloads** get scheduled on them.
- Fluent Bit pods will tolerate these taints.

🧱 Steps to Implement

✅ 1. Label a Node as Logging Node

> Pick one or more nodes you want to use only for logging.

```
kubectl label node ip-192-168-1-100 node-role=log
```

✅ 2. Taint the Node

```
kubectl taint nodes ip-192-168-1-100 logging=only:NoSchedule
```

📌 This tells Kubernetes:

> "Only schedule pods that tolerate `logging=only:NoSchedule` on this node."

✅ 3. Create Fluent Bit DaemonSet with Toleration + Node Selector

Here's a **fully functional DaemonSet YAML** that will:

- Tolerate the taint
- Target only the labeled `log` node

- Deploy one pod per logging node

```yaml
apiVersion: apps/v1

kind: DaemonSet

metadata:

  name: fluent-bit

  namespace: kube-logging

  labels:

    app: fluent-bit

spec:

  selector:

    matchLabels:

      app: fluent-bit

  template:

    metadata:

      labels:

        app: fluent-bit

    spec:

      tolerations:

      - key: "logging"
        operator: "Equal"
        value: "only"
        effect: "NoSchedule"

      nodeSelector:
        node-role: log

      containers:
      - name: fluent-bit
        image: fluent/fluent-bit:2.2
        resources:
          limits:
            memory: "200Mi"
            cpu: "100m"
          requests:
            memory: "100Mi"
            cpu: "50m"
        volumeMounts:
        - name: varlog
```

```
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: varlibdockercontainers
        hostPath:
          path: /var/lib/docker/containers
```

✅ 4. Apply the DaemonSet

```
kubectl create namespace kube-logging
kubectl apply -f fluent-bit-daemonset.yaml
```

✅ 5. Validate Everything

🔍 **Check the taint:**

```
kubectl describe node ip-192-168-1-100 | grep Taints
```

Should output:

```
Taints: logging=only:NoSchedule
```

🔍 **Check that Fluent Bit runs only on the logging node:**

```
kubectl get pods -n kube-logging -o wide
```

You should see pods running **only on `ip-192-168-1-100`**.

🔍 **Check that no other workloads are scheduled there:**

```
kubectl get pods --all-namespaces -o wide | grep
ip-192-168-1-100
```

Should show **only Fluent Bit**, nothing else.

🧠 Why This Matters in Production

- Keeps logging isolated from noisy neighbor workloads
- Ensures log agents are not starved for resources
- Enables fine-grained control over resource sizing and monitoring
- Used in large-scale infra at AWS, GCP, and enterprise setups

📎 Bonus Tip: Automate with Kops

If you're using Kops, you can define taints in the instance group like this:

```yaml
taints:
  - key: "logging"
    value: "only"
    effect: "NoSchedule"
```

This will taint the nodes automatically during cluster creation or rolling update.

📘 Real-Time Kubernetes Task #2:

Graceful Node Draining for Maintenance Using `NoExecute` Taints

🎯 Objective:

When a node is going into **maintenance (patching, reboot, draining)**, you want to **gracefully evict workloads** — but **only those that don't tolerate the taint**.

This allows:

- Critical pods to stay temporarily
- Others to evict cleanly
- Controlled disruption in production

📦 Use Case:

- You need to **restart or upgrade** a node without killing all pods instantly.
- You apply a **temporary taint** to evict non-critical workloads.
- Some critical workloads have a `tolerationSeconds` buffer to remain briefly and then gracefully shut down.

🛠️ Implementation Steps

✅ 1. Choose a Node for Maintenance

Example:

```
kubectl get nodes
```

Pick a node (say, `ip-192-168-1-80`) that needs to be restarted or upgraded.

✅ 2. Apply a Temporary Maintenance Taint (NoExecute)

```
kubectl taint nodes ip-192-168-1-80
maintenance=planned:NoExecute
```

This will **evict all pods** that **don't tolerate** this taint.

✅ 3. Deploy a Pod That Can Stay Temporarily (With `tolerationSeconds`)

Here's an example of a **critical pod** that can tolerate the maintenance taint for **300 seconds (5 minutes)**:

```
apiVersion: v1

kind: Pod

metadata:
```

```yaml
  name: payment-processor
  labels:
    app: payment
spec:
  containers:
  - name: main
    image: nginx
  tolerations:
  - key: "maintenance"
    operator: "Equal"
    value: "planned"
    effect: "NoExecute"
    tolerationSeconds: 300
```

✅ This pod:

- Will **not be evicted immediately**
- Will be given **5 minutes** to finish gracefully (e.g., draining a queue, flushing a cache)

✅ 4. Deploy a Pod That Gets Evicted Immediately

```
apiVersion: v1
kind: Pod
metadata:
  name: web-frontend
  labels:
    app: frontend
spec:
  containers:
  - name: web
    image: nginx
```

🚫 This pod has **no tolerations**, so when the node gets tainted, it is **evicted immediately**.

✅ 5. Observe Eviction Behavior

Use:

```
kubectl get pods -o wide
kubectl describe pod payment-processor
```

✅ You’ll see:

- `web-frontend` is terminated immediately
- `payment-processor` runs for 5 minutes, then is evicted

✅ 6. Cleanup Taint After Maintenance

Once the node is patched, remove the taint:

```
kubectl taint nodes ip-192-168-1-80
maintenance=planned:NoExecute-
```

🧠 Why This Is Production-Ready

- Used for **graceful upgrades and maintenance windows**
- Ensures **high availability** for critical services
- Prevents **violent pod termination**
- Gives control to platform engineers while respecting app SLOs

🛡️ Bonus Tip: Use with Cluster Autoscaler

Cluster autoscaler respects `NoExecute` taints and **doesn't scale down** nodes that have tainted but non-evictable pods.

📘 Real-Time Kubernetes Task #3:

Isolating GPU Workloads on Specialized Nodes Using Taints and Tolerations

🎯 Objective:

You want to schedule **machine learning (ML) or GPU-based workloads** (like TensorFlow, PyTorch, or inference APIs) **only on GPU-enabled nodes** in your cluster.

Other workloads should **never run on GPU nodes** to preserve resources and reduce scheduling conflicts.

📦 Use Case:

- You provision dedicated GPU nodes in your Kops cluster (using a separate instance group).
- You taint those nodes to prevent regular pods from landing there.
- You use tolerations on GPU-related pods so **only those can be scheduled**.

🛠️ Implementation Steps

✅ 1. Label and Taint the GPU Nodes

Let’s assume a node with GPU is: `ip-192-168-1-90`

```
kubectl label node ip-192-168-1-90 node-type=gpu
kubectl taint node ip-192-168-1-90 accelerator=nvidia:NoSchedule
```

✅ Now, only pods with the right toleration can be scheduled on this GPU node.

✅ 2. Create a GPU-Based Workload (With Toleration)

Here’s a **sample ML inference pod** that uses NVIDIA GPU and tolerates the taint.

```
apiVersion: v1
kind: Pod
metadata:
  name: image-classifier
spec:
  containers:
  - name: classifier
    image: nvcr.io/nvidia/tensorflow:23.06-tf2-py3
    resources:
      limits:
        nvidia.com/gpu: 1
  tolerations:
  - key: "accelerator"
    operator: "Equal"
    value: "nvidia"
    effect: "NoSchedule"
  nodeSelector:
    node-type: gpu
```

📌 Notes:

- `resources.limits.nvidia.com/gpu: 1` requests 1 GPU
- The toleration matches the taint on the node
- `nodeSelector` ensures it’s scheduled only on GPU-labeled nodes

✅ 3. Deploy the Pod

```
kubectl apply -f image-classifier.yaml
```

Check:

```
kubectl get pod image-classifier -o wide
```

✅ Should be running on `ip-192-168-1-90` (the GPU node)

✅ 4. Try to Schedule a Regular Pod Without Toleration

```
apiVersion: v1

kind: Pod

metadata:

  name: web-frontend

spec:

  containers:

  - name: web

    image: nginx
```

```
kubectl apply -f web-frontend.yaml
```

This pod **will not get scheduled** on the GPU node because:

- It doesn't tolerate the `accelerator=nvidia:NoSchedule` taint
- The scheduler avoids GPU nodes

✅ 5. Validate the Setup

```
kubectl get pods -o wide

kubectl describe node ip-192-168-1-90 | grep Taint
```

You'll see:

```
Taints: accelerator=nvidia:NoSchedule
```

And only GPU pods are allowed on it.

🧠 Why This Pattern Is Important in Production

- Prevents **non-GPU workloads** from wasting GPU resources
- Enforces **workload affinity** in heterogeneous node pools
- Useful for **ML workloads**, **video processing**, **AR/VR**, **AI inference APIs**
- Supports efficient **resource isolation and scaling**

💡 Kops Integration Tip:

In your Kops instance group for GPU nodes, add this:

```yaml
taints:
  - key: "accelerator"
    value: "nvidia"
    effect: "NoSchedule"
labels:
  node-type: gpu
```

This automatically taints and labels nodes during cluster update.

📘 Real-Time Kubernetes Task #4:

Isolating Team Workloads in a Multi-Tenant Cluster Using Taints and Tolerations

🎯 Objective:

In a **multi-team (multi-tenant)** Kubernetes cluster managed via Kops, ensure that workloads from different teams **run on dedicated node pools**. Prevent "noisy neighbor" issues, resource competition, or accidental deployments across teams.

📦 Use Case:

- Teams like `team-a`, `team-b`, and `platform` share the same cluster.
- Each team has its own **dedicated node group** via Kops.
- Taints are used to **repel unrelated workloads** from those nodes.
- Tolerations ensure only the owning team's workloads get scheduled on the right nodes.

🛠️ Implementation Steps

✅ 1. Set Up Node Isolation via Kops (Taint + Label)

Assume a dedicated instance group for `team-a`. In the instance group YAML (via `kops edit ig team-a`):

```
taints:

  - key: "dedicated"

    value: "team-a"

    effect: "NoSchedule"

labels:

  node-pool: team-a
```

After updating:

```
kops update cluster --yes

kops rolling-update cluster --yes
```

This:

- Taints all nodes in this group with `dedicated=team-a:NoSchedule`
- Labels them with `node-pool=team-a`

✅ Only pods that tolerate `dedicated=team-a` will be scheduled there.

✅ 2. Deploy Team-A Workloads With Toleration + NodeSelector

Here’s a sample Deployment for `team-a` that is correctly configured:

```
apiVersion: apps/v1

kind: Deployment

metadata:

  name: team-a-api

  namespace: team-a

spec:

  replicas: 2

  selector:

    matchLabels:

      app: team-a-api

  template:

    metadata:

      labels:

        app: team-a-api

    spec:

      tolerations:
```

```yaml
- key: "dedicated"
  operator: "Equal"
  value: "team-a"
  effect: "NoSchedule"
nodeSelector:
  node-pool: team-a
containers:
- name: api
  image: teama/api:latest
  resources:
    requests:
      memory: "256Mi"
      cpu: "250m"
    limits:
      memory: "512Mi"
      cpu: "500m"
```

✅ 3. Try Deploying Without Toleration (Simulate Misconfiguration)

Let’s say someone on `team-b` mistakenly tries to deploy to `team-a`’s node group:

```
apiVersion: v1
kind: Pod
metadata:
  name: rogue-pod
  namespace: team-b
spec:
  containers:
  - name: nginx
    image: nginx
  nodeSelector:
    node-pool: team-a
```

🔍 Result:

```
kubectl describe pod rogue-pod -n team-b
```

⛔ You'll see:

```
node(s) had taint {dedicated=team-a: NoSchedule}, that the
pod didn't tolerate
```

✅ This protects node isolation.

✅ 4. Validation

🔎 **Check taints on the node:**

```
kubectl describe node <node-name> | grep Taint
```

🔎 **Confirm pod placement:**

```
kubectl get pods -o wide -n team-a
```

Should show `team-a-api` pods only on `node-pool: team-a` nodes.

🧠 Why This Pattern Works in Production

- Enforces **team-level workload isolation**
- Prevents **accidental cross-scheduling**
- Works well in **shared clusters**
- Scales with your **infrastructure-as-code via Kops**

Used heavily by:

- Platform teams in mid-large orgs
- SaaS companies with shared Kubernetes clusters
- Cost optimization: isolate by environment (prod/dev/staging)

💡 Bonus Tip: Namespace + RBAC + Tainting = Real Isolation

- Namespace per team (✔️)
- RBAC scoped to namespace (✔️)
- Node taints per team (✔️)
- ResourceQuota and LimitRange (🔒)
- NetworkPolicy (optional 🔐)

Put together = multi-tenant SaaS-grade architecture.

📘 Real-Time Kubernetes Task #5:

Blue-Green Deployment Separation Using Taints and Tolerations

🎯 Objective:

Implement **blue-green deployments** in Kubernetes using **taints and tolerations** to isolate traffic between two environments during rollout, testing, and production cutover.

This method allows:

- **Two versions of the app to run simultaneously**
- Controlled traffic switching
- Full environment isolation before cutover
- Seamless rollback

📦 Use Case:

- You run both `blue` (live) and `green` (new) environments in parallel.
- Blue and Green workloads **run on separate node pools**.

- During testing, only internal teams access green.
- After validation, you switch traffic to green (e.g., via Ingress, DNS, or service label changes).

🛠️ Implementation Steps

✅ 1. Provision Dedicated Node Groups (Kops)

Create 2 instance groups via Kops:

- `blue-group` ➔ Live traffic
- `green-group` ➔ Staging/new version

In `blue-group`, add:

```
labels:
  deployment-color: blue
taints:
  - key: "env"
    value: "blue"
    effect: "NoSchedule"
```

In `green-group`, add:

```
labels:
  deployment-color: green
taints:
  - key: "env"
    value: "green"
    effect: "NoSchedule"
```

Apply and roll the cluster:

```
kops update cluster --yes
kops rolling-update cluster --yes
```

✅ 2. Deploy the Blue Version of the App

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-blue
spec:
  replicas: 2
```

```yaml
  selector:
    matchLabels:
      app: myapp
      color: blue
  template:
    metadata:
      labels:
        app: myapp
        color: blue
    spec:
      nodeSelector:
        deployment-color: blue
      tolerations:
      - key: "env"
        operator: "Equal"
        value: "blue"
        effect: "NoSchedule"
      containers:
      - name: app
```

```
        image: myorg/myapp:1.0.0
        ports:
        - containerPort: 80
```

✅ Pods only run on `blue` tainted nodes.

✅ 3. Deploy the Green Version (New Release Candidate)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-green
spec:
  replicas: 2
  selector:
    matchLabels:
      app: myapp
      color: green
  template:
    metadata:
      labels:
        app: myapp
        color: green
    spec:
      nodeSelector:
        deployment-color: green
      tolerations:
      - key: "env"
        operator: "Equal"
        value: "green"
        effect: "NoSchedule"
      containers:
      - name: app
        image: myorg/myapp:2.0.0
        ports:
        - containerPort: 80
```

✅ Pods for green version run **exclusively** on green nodes.

✅ 4. Route Traffic Strategically

Use a Service or Ingress to control traffic:

Initial Setup (Blue is Live):

```yaml
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  selector:
    app: myapp
    color: blue
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

Switch to Green (Post Validation):

Update the selector:

```yaml
spec:
  selector:
    app: myapp
    color: green
```

Or if using Ingress (e.g., with nginx or alb), point the path/backend to green version.

✅ 5. Optional Rollback Strategy

If green version fails tests or SLOs:

- Simply revert the service or Ingress back to blue
- Delete or scale down green Deployment

No need to re-schedule or rollback nodes — isolation is already in place.

✅ Validate

Check pods and nodes:

```
kubectl get pods -o wide

kubectl describe node <node-name>
```

You’ll observe:

- Pods only scheduled to the intended tainted nodes
- No workload interference between environments

🧠 Why This Pattern Works in Production

- True **environment-level isolation**
- **Quick rollback** by switching traffic
- Zero-downtime deployment pattern
- Excellent for regulated systems, canary releases, A/B testing, or client-specific upgrades

🛠️ Troubleshooting: Taints and Tolerations in Kubernetes

🧩 Problem 1: Pod is not getting scheduled (Pending state)

🔍 Symptom:

```
kubectl get pod <pod-name>

STATUS: Pending
```

🔎 Diagnostic Command:

```
kubectl describe pod <pod-name>
```

✅ Typical Output:

```
Events:

  Warning  FailedScheduling  ...  0/3 nodes are available: 3 node(s) had taint {dedicated=team-a: NoSchedule}, that the pod didn't tolerate
```

💥 Root Cause:

Pod is trying to land on tainted nodes but doesn't **have matching tolerations**.

🩹 Solution:

Add the correct toleration to the Pod spec:

```
tolerations:
- key: "dedicated"
  operator: "Equal"
  value: "team-a"
  effect: "NoSchedule"
```

🧩 Problem 2: Pod has toleration, but still not scheduled

🔍 Symptom:

Pod has a toleration, but it's still pending and not running on expected node.

🔎 Check NodeSelector or Affinity:

```
kubectl describe pod <pod-name> | grep -A 10 Tolerations
```

Also check:

```
kubectl describe node <node-name> | grep Labels
```

💥 **Root Cause:**

- Toleration is present ✅
- But Pod has `nodeSelector` or `affinity` that doesn't match the tainted node's labels ❌

🩹 **Solution:**

Make sure the `nodeSelector` matches the node's label:

```
nodeSelector:
  node-pool: team-a
```

Or check `affinity/preferredDuringScheduling` rules if using them.

🧩 Problem 3: Pod keeps restarting or getting evicted

🔍 **Symptom:**

Pod shows `Evicted`, or restarts frequently.

🔎 **Check Events or Describe Pod:**

```
kubectl get pod <pod-name>
kubectl describe pod <pod-name>
```

💥 **Root Cause:**

Node has a `NoExecute` taint — and the pod didn’t tolerate it.
So it's being evicted as soon as it lands.

🩹 **Solution:**

Add `NoExecute` toleration with optional `tolerationSeconds`:

```
tolerations:
- key: "critical"
  operator: "Equal"
  value: "true"
  effect: "NoExecute"
  tolerationSeconds: 60
```

This keeps the pod for 60 seconds before eviction.

🧩 Problem 4: Taint removed, but pods still not scheduled

🔍 **Symptom:**

Even after removing taint, pods don’t get scheduled to the node.

🔎 **Double Check Taint Removal:**

```
kubectl describe node <node-name>
```

💥 **Root Cause:**

- Taint might have been removed ❌
- But other **scheduling conditions** (like disk pressure, node unschedulable, etc.) are blocking pod placement

🩹 **Solution:**

Check full node status:

```
kubectl get nodes

kubectl describe node <node-name>
```

Look for:

- `SchedulingDisabled`
- DiskPressure
- PIDPressure
- MemoryPressure

🧩 Problem 5: Pod lands on wrong node despite taints

🔍 Symptom:

You applied a taint on the node, but a pod still lands on it.

🔎 Check Taint Effect:

```
kubectl describe node <node-name>
```

💥 Root Cause:

- Taint might be using `PreferNoSchedule`, which is a **soft** taint (scheduler *tries* to avoid but not guaranteed).
- Or pod has a **wildcard toleration**, which tolerates all taints.

🩹 Solution:

✅ Use `NoSchedule` instead of `PreferNoSchedule` for hard enforcement.

Also, avoid wildcard tolerations unless necessary:

```
tolerations:
- operator: "Exists"
```

This tolerates all taints — use carefully!

🧩 Problem 6: YAML syntax issues when defining taints/tolerations

💥 Common Mistakes:

- Wrong indentation
- `effect` missing
- Using `tolerate` instead of `tolerations`
- Misplaced under metadata

✅ Correct Toleration Block:

```yaml
tolerations:
- key: "example"
  operator: "Equal"
  value: "test"
  effect: "NoSchedule"
```

Use a linter or validate your YAML:

```
kubectl apply --dry-run=client -f myfile.yaml
```

🧪 Commands Summary for Debugging

Task	Command
List taints on node	`kubectl describe node <node>`
Describe pod for scheduling info	`kubectl describe pod <pod>`
View unschedulable pods	`kubectl get pods --field-selector status.phase=Pending`
Dry-run to validate YAML	`kubectl apply --dry-run=client -f file.yaml`

✅ Best Practices to Avoid Taint Issues

- Always pair **taints** with **labels** and use `nodeSelector` or `affinity` to schedule accurately.
- Avoid overusing `NoExecute` unless you need eviction behavior.
- Set alerts for **Pending pods** via monitoring tools (Prometheus, Datadog, etc.).

- Document every taint in your infra-as-code (e.g., Kops instance groups or Terraform).

✅ Best Practices for Taints and Tolerations in Kubernetes

🎯 1. Use Taints to Enforce Node Roles — Not Replace Scheduling Policies

- Taints **should not be used alone** for workload placement.
- Always combine with:
  - `nodeSelector`
  - `nodeAffinity`
  - node `labels`

📌 Example:

```
nodeSelector:
  workload-type: gpu

tolerations:
- key: "gpu"
  operator: "Equal"
  value: "true"

  effect: "NoSchedule"
```

🔒 This gives **hard isolation** and **explicit scheduling**, not accidental pod placement.

🎯 2. Avoid Wildcard Tolerations Unless Absolutely Required

```
tolerations:

- operator: "Exists"
```

- This tolerates **any taint**, which could lead to **pods landing on restricted nodes** (like control-plane or spot-only pools).
- Use only in **infrastructure-level DaemonSets** like `fluentd`, `cilium`, `kube-proxy`, etc.

✅ Instead, use **scoped tolerations** with key/value pairs.

🎯 3. Label Every Tainted Node Group and Use NodeSelector

- Taints block scheduling ❌
- Tolerations allow it ✅
- But without a label-selector, the scheduler may try to place pods across unrelated tainted pools.

🛡️ Pair taints with `labels` and schedule pods precisely:

```
labels:
  node-group: platform-nodes


nodeSelector:
  node-group: platform-nodes
```

🎯 4. Use `NoExecute` With Care

- `NoExecute` evicts pods **immediately or after tolerationSeconds**.
- Use in **health-related scenarios** like:
  - Node fails probe

  - Node enters disk-pressure state

✅ Add `tolerationSeconds` to avoid instant eviction during minor blips:

```
tolerations:

- key: "node.kubernetes.io/not-ready"

  effect: "NoExecute"

  tolerationSeconds: 60
```

🎯 5. Isolate Critical Workloads With Taints

For example:

- Platform-level services
- Logging agents
- System-level DaemonSets
- High-priority APIs

✅ Taint their dedicated nodes and use strict tolerations.

```
taints:

  - key: "dedicated"

    value: "platform"
    effect: "NoSchedule"
```

🎯 6. Default Node Pools Should Remain Untainted for General Workloads

- Keep your “default” or “generic” node pool **untainted**.
- Only apply taints where **isolation is required** (e.g., GPU, staging, infra).

This avoids broken scheduling for non-tolerating pods by default.

🎯 7. Document and Standardize Taints in GitOps/IaC

- Taints should live in:
  - `kops` instance group YAMLs
  - `terraform` configs
  - `Cluster API` specs

✅ Track taints the same way you do labels, annotations, node selectors, etc.

```
taints:
  - key: "workload"
    value: "ml"
    effect: "NoSchedule"
```

🎯 8. Validate Taints and Tolerations During CI

Use tools like:

- `kubeval` or `kubeconform`
- `OPA Gatekeeper` policies
- Admission controllers

✅ Example Gatekeeper policy: prevent wildcard tolerations in dev workloads.

🎯 9. Use Monitoring for Pending Pods and Evictions

Track:

- `Pending` pod counts
- `Evicted` pod reasons
- `taint_toleration_mismatch` in logs or metrics

🔍 Alert if:

- Pods are pending for > X seconds
- Node taints unexpectedly increase (e.g., due to autoscaler)

🎯 10. Use Taints for Spot Node Isolation

For cost-effective, fault-tolerant workloads:

```
taints:
- key: "spot"
  value: "true"
  effect: "NoSchedule"
```

✅ Schedule only retryable or stateless workloads with this toleration.

✅ Bonus Best Practice: Taint the Control Plane

In most Kubernetes setups (including Kops), control-plane nodes come pre-tainted:

```
node-role.kubernetes.io/control-plane:NoSchedule
```

⚠️ Never remove this taint unless:

- You’re doing single-node testing (NOT recommended in prod)
- You **intentionally** want to run workloads there (with toleration)

📌 Node Labels and Selectors in Kubernetes

🧠 What are Node Labels?

Node labels are simple key-value pairs attached to Kubernetes nodes.
They are used to **classify** and **group** nodes based on attributes like:

- Instance type
- Environment (dev/prod)
- Zone or region
- GPU/CPU specialization
- Node group name
- Spot vs on-demand

📌 Labels are **immutable per node lifecycle**, but can be changed dynamically.

```
kubectl label node ip-172-20-30-100 workload=gpu
```

✅ Syntax Example

```
labels:

  node-role.kubernetes.io/worker: ""

  instance-type: "c5.large"

  environment: "production"

  workload: "ml"
```

📦 How Do We Use Labels?

They are **used by pods to request specific nodes** via:

- `nodeSelector`
- `nodeAffinity`

This ensures **workload placement** aligns with node capability or purpose.

🎯 nodeSelector: Simple and Common

✅ What is nodeSelector?

`nodeSelector` is the **simplest way** to bind a pod to a node with specific labels.

```
spec:
  nodeSelector:
    workload: ml
    environment: production
```

This will schedule the pod **only** on nodes that have both:

- `workload=ml`
- `environment=production`

🧪 Real Example: Run a Pod on GPU Nodes

Let's assume nodes in your cluster are labeled like:

```
kubectl label node ip-172-20-50-10 gpu=true
```

Now create a pod with:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: gpu-pod
spec:
  containers:
  - name: app
    image: tensorflow/tensorflow:latest-gpu
  nodeSelector:
    gpu: "true"
```

✅ This ensures the pod runs only on GPU-labeled nodes.

🚦 Use Case Scenarios

Use Case	Label	Selector Example
Environment-specific	`env=prod` / `env=dev`	`nodeSelector: { env: prod }`
Node group isolation	`node-pool=infra`	`nodeSelector: { node-pool: infra }`
Architecture matching	`beta.kubernetes.io/arch`	`nodeSelector: { arch: amd64 }`
Region/zone targeting	`topology.kubernetes.io/zone`	`nodeSelector: { ... }`

🚀 Node Affinity (Advanced Alternative)

If you want more expressive rules (like OR/AND logic), use **Node Affinity** instead of `nodeSelector`.

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: environment
          operator: In
          values:
          - production
          - staging
```

✅ More flexibility than `nodeSelector`, especially in multi-zone, multi-tenant environments.

🛠️ Commands for Node Labels

🔍 View Node Labels

```
kubectl get nodes --show-labels
```

🏷️ Add Label to a Node

```
kubectl label node <node-name> zone=us-east-1a
```

❌ Remove Label from Node

```
kubectl label node <node-name> zone-
```

⚠️ Best Practices for Node Labels & Selectors

✅ Label Strategy

- Use **consistent naming** conventions (e.g., `env`, `node-pool`, `zone`, `workload`).

- Use **labels** for capabilities, and **taints/tolerations** for restrictions.

- Document every label in GitOps/IaC (like in Kops instance group YAMLs).

⚠️ Don't Rely on Dynamic Labels for Scheduling

- Dynamic labels (e.g., added at runtime via script) might cause flaky scheduling.

- Label nodes at **provision time** (via `Kops`, Terraform, etc.).

✅ YAML Summary Example (Pod + Node)

```yaml
# Assume this node has:
# labels: workload=ml, env=prod

apiVersion: v1
kind: Pod
metadata:
  name: ml-job
spec:
  containers:
  - name: app
    image: ml-image:latest
  nodeSelector:
    workload: ml
    env: prod
  tolerations:
  - key: "dedicated"
    operator: "Equal"
    value: "ml"
```

```
effect: "NoSchedule"
```

🔐 This pod will only run on nodes that match the labels AND tolerate the taint — perfect for isolated workloads like ML, platform services, or GPU jobs.

🚀 Taints & Tolerations + Node Labels & Selectors — Complete Concept

🧠 Understanding the Roles

Feature	Purpose
Labels	Add metadata to nodes (e.g., type, zone, purpose)
Selectors	Used in pods to **choose** nodes with specific labels
Taints	Applied to nodes to **repel** pods unless they tolerate them
Tolerations	Applied to pods so they can be **scheduled on tainted nodes**

🧩 How They Work Together

- **Labels & nodeSelectors**: Used to **target** specific node(s).
- **Taints & tolerations**: Used to **restrict** which pods can run on those nodes.

- 🔄 Use **both** to avoid accidental scheduling or unwanted behavior.

🎯 Real-World Example Scenario: Dedicated ML Workloads on GPU Nodes

🧱 1. Label the GPU Nodes

```
kubectl label node ip-10-0-0-5 workload=gpu
```

Now the node has:

```
labels:
  workload: gpu
```

🧱 2. Taint the GPU Node

```
kubectl taint nodes ip-10-0-0-5 dedicated=gpu:NoSchedule
```

Now the node has:

```
taints:
  - key: "dedicated"
    value: "gpu"
    effect: "NoSchedule"
```

💡 This prevents any pod from being scheduled **unless it tolerates** this taint.

🧱 3. Deploy Pod That Matches Label + Tolerates Taint

```
apiVersion: v1
kind: Pod
metadata:
  name: ml-job
spec:
  containers:
  - name: trainer
    image: tensorflow/tensorflow:latest-gpu
  nodeSelector:
    workload: gpu
  tolerations:
  - key: "dedicated"
    operator: "Equal"
    value: "gpu"
    effect: "NoSchedule"
```

✅ What Happens Here?

1. Kubernetes **finds nodes** with `workload=gpu` — because of the `nodeSelector`.
2. Scheduler sees the **taint** on those nodes: `dedicated=gpu:NoSchedule`.
3. Pod has a **matching toleration** — so it is allowed to schedule on those nodes.
4. Pod runs only on GPU nodes and stays isolated from the rest of the cluster.

🛡️ Why Use Both Together?

Using **only labels** means **any pod** can try to run on the node — not safe for production isolation.

Using **only taints** means **any pod with a wildcard toleration** may sneak in — not precise.

✅ Best Practice:

- Use **labels** + **nodeSelector** to **target** node

- Use **taints** + **tolerations** to **control** access

🚨 Example: Misuse Without Selector

If the pod had only toleration and no selector:

```yaml
tolerations:

- key: "dedicated"

  operator: "Equal"

  value: "gpu"

  effect: "NoSchedule"
```

❌ Scheduler might still place this pod **on any node with that taint**, even if it's not intended for GPU workloads.

This is risky in **shared or multi-tenant clusters.**

✅ Wrapping Up

Thank you so much for taking the time to read this documentation on **Kubernetes Taints, Tolerations, Node Labels & Selectors**. 🙌

We've gone beyond just definitions — diving into how these features work together to **control, optimize, and isolate workload scheduling** in real-world production clusters. From foundational concepts to practical implementation using `kops`, you now have the tools to confidently apply these patterns in your own Kubernetes environments.

💡 What's Next?

This doc is part of a larger series focused on hands-on, production-grade DevOps knowledge across:

- ☁️ **AWS Infrastructure**
- 🔧 **Ansible Automation**
- ☸️ **Kubernetes Operations**
- 📦 **Real-world Project Scenarios**
- 🚀 **Scalable Cloud-Native Practices**

If you found this helpful, make sure to check out my other write-ups — and stay tuned for even more practical, well-structured docs coming very soon.

🙏 Thank You

Once again, thank you for being here and for supporting my DevOps content journey.
Whether you're just starting out or running production workloads, I hope this guide gave you something valuable.

📌 **Follow me on [LinkedIn] and [GitHub]** for more upcoming deep dives, tutorials, and end-to-end DevOps scenarios — all with a strong focus on practical knowledge and real-world relevance.

👋 Until next time — stay curious, keep experimenting, and happy deploying!
More docs are on the way. 🔥📘