Why Your Pod Is Pending: A Deep Dive into the K8s Scheduler

medium.com/@pragalva.sapkota/why-your-pod-is-pending-a-deep-dive-into-the-k8s-scheduler-42e9f608d155

Pragalva Sapkota January 13, 2026

Introduction

We have all created Pods, seen them carry out workloads, but as fascinating as Pods are, it is essential to understand how and where those Pods are scheduled.

In this blog, I will dive deep into Kubernetes Scheduling using analogies and leanings from hands-on experimentation and studying the official documentation.

Kubernetes Scheduling

In Kubernetes, scheduling is the process of making sure Pods are matched to the most feasible **Node**, so that the **kubelet** can run them.

Node Selection in Kubernetes

The **kube-scheduler** is the default scheduler for Kubernetes and runs as a part of the control plane.

If you want to learn more about the Kubernetes architecture, you can read 👉 **[here]**

Whenever you create a Pod, the kube-scheduler goes through a rigorous process to find the most optimal Node. The workflow looks like this:

1) Queue
This is when you have created the Pod and it is in the Pending state.
The scheduler watches this queue and proceeds to find the perfect match for each Pod.

2) Filtering
Filtering is like the elimination round for Nodes that cannot, or will not be able to run the Pod efficiently.
This is done by applying hard constraints.
You can think of it as the scheduler removing everything that does not have the must-haves.

3) Scoring
Scoring is done for all Nodes that passed filtering.
The scheduler ranks them using scoring functions.
Essentially, scoring checks which Node has the most amount of good-to-haves.Examples of good-to-haves:
- Does this Node already have the container image downloaded?
- Is this Node underutilized, for example better memory availability?

4) Binding
After scoring, a winner Node is selected, and the scheduler binds the Pod to that Node.

Pod Scheduling Readiness

Sometimes we may create a Pod that is not meant to run, or even get scheduled, immediately.

Why?
Because in some cases Pods might be waiting for:
- an external database to spin up
- a CRD to be ready

In such cases, we can add Scheduling Gates to allow the Pod to exist but tell the scheduler to ignore it.That means the Pod will not be scheduled, therefore it will not run.This saves control plane resources by avoiding unnecessary scheduling loops.Example:

```
spec:
  schedulingGates:
  - name: “example.com/waiting-for-database”
```

Topology Spreading Constraints:

This is a feature in kubernetes that allows us to control how Pods are spread across various domains such as nodes, clusters, regions, zones, etc.

Imagine that you have a cluster with 10 nodes, you want to have 10 replicas of your pod. If a single node has all of the pods, and the node encounters some disaster, you will lose all of your pods. But with the help of topology spreading constraints you can avoid such incidents by spreading your pod across multiple nodes ensuring better disaster recovery. The same logic can be used for other domains as well.

1. It is the degree to which the pods can be unevenly distributed. To further understand this lets look into an example.
   If maxSkew: 1, the Scheduler tries to keep the number of pods on all nodes almost exactly equal. If Node A has 2 pods, Node B cannot have 0 (difference is 2). Node B *must* have at least 1.
2. **Topology key**
   This defines the domain that we are balancing our pods across. It can be across different nodes, or across different data centers.
3. Here we can define the strictness of our spreading constraints.
   : If the spread isn’t perfect, the pod will not be scheduled.
   : It wishes that the constraints are perfectly applied, but if it isn’t applied it still allows the pod to be scheduled.
4. This enlightens the scheduler about the pods it should check. It’s like saying only send the students outside whose name starts with A.

5. It is a 32 bit integer, which defines the priority of the pod. The higher it is the more priority it holds and vice-versa. The lowest priority pod is preempted if there is no available resource for a mission-critical pod (a pod with high priority).
6. Running a Pod isn't free; the sandbox itself needs resources. We specify this in the RuntimeClass so the scheduler accounts for it:

```
Total Request = Container Requests + Pod Overhead.
```

Node Affinity:

Think of Node Affinity as a magnet. You are giving the Pod a magnetic attraction to specific nodes based on their labels.

It is similar to node selector, where your pods can be scheduled based on labels of nodes.

The hard requirement vs soft requirement:

The "Must-Have" :

```
requiredDuringSchedulingIgnoredDuringExecution
```

It has a hard rule, which if not complied with, the pod will NEVER run. Example: I *must* run on a node with label gpu=true. If no such node exists, I will not run at all. I will stay Pending forever. This might look like a stubborn rule but has a huge use case like having a Pod that requires a specific hardware driver (like a GPU) to even start.

The "Nice-to-have":

```
preferredDuringSchedulingIgnoredDuringExecution
```

It acts like a chill parent. Imagine you have a server that you want to run in a different zone for better latency and the zone is full. In this case, it is not wise to crash the entire server because the zone was full.

Note: What does *IgnoreDuringExecution mean?*
Kubernetes promises **Stability** over **Perfection**. Once a Pod is scheduled (Execution phase), the Scheduler stops checking the affinity rules. If a Node

loses its gpu=true label, the Pod running there won't be evicted immediately. It stays until it is manually deleted or the node crashes.

Pod Affinity & Anti-Affinity (Friends vs. Rivals)

We just covered how Pods choose Nodes. Now, let's look at how Pods choose their **neighbors**.

Pod Affinity (The "Friends" Rule) Pod Affinity is essentially the "buddy system" for your containers. It allows you to tell the scheduler that a specific Pod should be placed on the same node as another existing Pod. This is primarily used for performance; for example, if you ensure your Web App and Cache Pods land on the same machine, they can communicate over localhost with lightning-fast speeds, completely avoiding network latency.

Pod Anti-Affinity (The "Rivals" Rule) On the flip side, Pod Anti-Affinity acts as a separation rule, instructing the scheduler to keep specific Pods far apart. This is critical for High Availability and disaster recovery. If you are running three replicas of an application, you don't want them all crowding onto a single server that could crash and take your entire service down. Anti-affinity forces the scheduler to spread those replicas across different nodes, ensuring that if one server fails, the others stay alive.

Taints and Tolerations (The "Keep Out" Sign)

We have talked about how Pods get attracted to Nodes (Affinity), but what if a Node wants to reject a Pod? This is where **Taints and Tolerations** come in.

Think of Taints as bug spray. If you spray a node with a Taint, Pods will naturally avoid it. The only way a Pod can land on that node is if it has a specific **Toleration** , think of this as a hazmat suit that makes the Pod immune to the spray. This is incredibly useful for reserving special hardware. For example, if you have expensive GPU nodes for AI training, you can "taint" them so that regular web-server pods don't accidentally clog them up.

The Three Effects (How strict is the spray?)

1. The strict doorman. If a Pod doesn't have the matching toleration, it is strictly forbidden from entering the node. It won't be scheduled there, period.
2. The "nice" rejection. The scheduler tries its best to avoid placing the pod here, but if there is literally nowhere else to go, it will let the pod in.
3. The bouncer. This is unique because it affects pods *already running* on the node. If you add this taint to a live node, any existing pods without the

toleration are immediately (kicked off). This is often used when a node is having hardware issues and needs to be drained for maintenance.

Conclusion

We have covered the complete lifecycle of a Kubernetes Pod. The process begins in the **Queue**, where **Scheduling Gates** may defer execution until specific conditions are met. Next, the Scheduler performs **Filtering** to eliminate nodes that do not meet resource requirements or satisfy node selectors. The remaining nodes undergo **Scoring**, where they are ranked based on active plugins, before the Pod is finally **Bound** to the optimal node. We also examined how to control placement using **Affinity** rules for co-location and **Taints** to restrict pod execution on specific nodes.

Understanding these mechanics allows you to move beyond default behaviors and actively configure the scheduler for performance, stability, and resource efficiency.

Go Hands-On

Now that you understand how it works, it's time to get your hands dirty. Here in my repo, you can find all the concepts that I have covered, which you can easily implement with some tweaks: 👉 **[here]**

Huge shout out to **KubeSimplify** who helped me clear my concepts on this topic. Please do check them out!