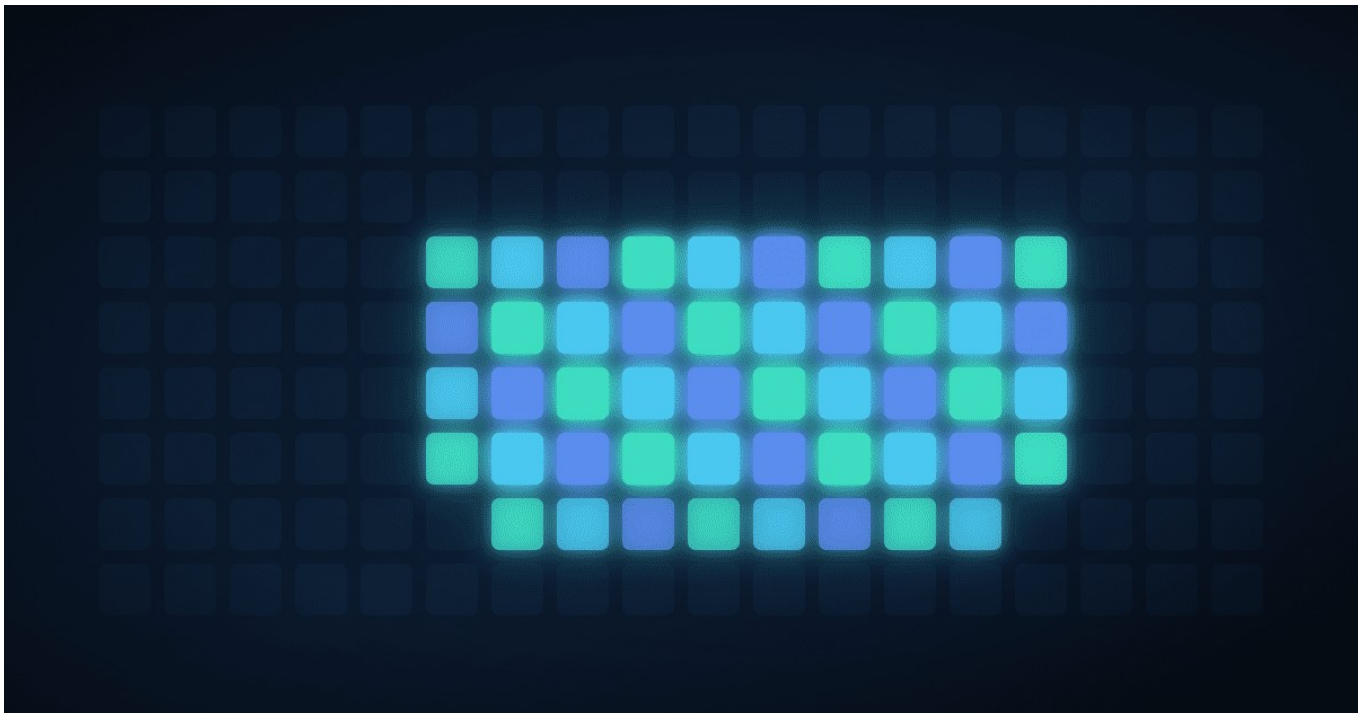


Spark on Kubernetes Reserves CPU It Never Uses. Here's the Overcommit Fix.

Julien Laurenceau | Published 2026-05-31 | Updated 2026-10-01

<https://pepitedata.com/spark-kubernetes-cpu-overcommit-operator/>

PepiteData | LinkedIn: Julien Laurenceau | X: @Julien_Lau



This is a failure mode I fix for clients who run Spark on Kubernetes. The [Spark tuning, benchmark and scale service](#) page describes the engagement.

On paper, the cluster was full. In practice, the CPUs were barely working, and new Spark executors were stuck in Pending. If you run Spark on Kubernetes, you have almost certainly seen the same thing: a compute plane that reports itself as saturated while the actual processors sit at 30 to 50% utilization. You are paying for nodes you cannot use.

This post explains why that happens, why Kubernetes will not fix it for you natively, and the operator I now use to get the cluster back.

The symptom: a cluster that is full and idle at the same time

Kubernetes schedules pods against **requests**, not against real usage. The scheduler reads a node's allocatable CPU and memory from the kubelet (which gets them from cAdvisor as physical values) and packs pods until the sum of their *requests* reaches that capacity. It does not care that those pods are only using half of what they reserved.

So you end up in the worst of both worlds: the scheduler refuses to place more work because the node is "full" on reservations, while top on that same node shows cores doing nothing. Pending executors, idle silicon, and a capacity-planning conversation that makes no sense.

Why Spark does this: requests = limits = Guaranteed QoS

Spark is the main offender here, and it is by design. When you ask for `spark.executor.cores=N`, Spark assumes those N cores will be used at 100% by parallel compute, so by default it sets the executor pod's CPU **request equal to its limit**. That gives the pod the **Guaranteed** QoS class, and a hard reservation of N cores on the node.

The assumption is wrong for most real workloads. Spark jobs do a lot of I/O (shuffle, object-storage reads, network) and during those phases the executor is not burning CPU. This is exactly the storage-bound behaviour I set out to measure when I [modernized the TPCx-HS benchmark for Spark on Kubernetes](#). Across a fleet of jobs, effective CPU usage rarely exceeds ~50% of what was reserved. So a Spark-backed Kubernetes cluster reserves roughly twice the CPU it actually needs.

There is a second, sharper edge to this. Spark's `spark.executor.cores` only accepts **integers** (you cannot ask it for "2.5 cores"), which is a long-standing constraint tracked in [SPARK-32744](#). Spark does expose `spark.kubernetes.executor.request.cores` to set a fractional pod request independently of the limit, but that is a *per-job* knob: it lives in whoever submits the job, it is easy to forget, and you cannot enforce it as a cluster-wide policy. What a cluster operator actually wants is a single lever that applies to every workload, regardless of how careful each team is with its submit flags.

I asked Kubernetes for a native fix. The answer was no.

The clean solution would be a node-level overcommit factor: tell the cluster "treat these worker nodes as if they had 2× the CPU," exactly like OpenStack or VMware have done for years. So I opened an issue with SIG-node asking for it: [kubernetes/kubernetes#132286: Support resources overcommit for cpu memory](#).

The feedback was honest and, frankly, fair. Overcommit at the node level ties into [KEP-5224 \(Node resource discovery\)](#): a move from static to dynamic node resources that is a large, long-horizon effort. As one maintainer put it, "adding overcommit support is a huge task which is unlikely to be implemented anytime soon." The issue eventually aged out and was closed as *not planned*.

Native overcommit is not coming soon. But the problem is real today, and it turns out someone had already built the workaround.

The remediation: rewrite requests, leave limits alone

On that same issue, engineers from Inditex pointed me to an operator they built and run in production: [InditexTech/k8s-overcommit-operator](#). It is the pragmatic version of what I was asking Kubernetes for: implemented where you actually can implement it: in a **mutating admission webhook**.

The mechanism is deliberately simple. When a pod is created, the webhook intercepts it and rewrites its CPU and memory **requests** to a configurable fraction of its **limits**. It never touches the limits. So a Spark executor declared with a 4-core limit and a 4-core request becomes, with a 0.5 CPU ratio:

- The **request** is 2 cores: what the scheduler reserves
- The **limit** is 4 cores: what the executor can still burst to when it is actually computing

You configure it with two CRDs. A singleton `Overcommit` object (it must be named `cluster`) declares which label selects an overcommit class:

```
apiVersion: overcommit.inditex.dev/v1alpha1
kind: Overcommit
metadata:
  name: cluster
spec:
  overcommitLabel: inditex.com/overcommit-class
```

Then one or more `OvercommitClass` objects define the actual ratios. `cpuOvercommit: 0.2` means the request becomes 20% of the limit. Critical namespaces are protected with a regex, and one class can be marked as the cluster default:

```
apiVersion: overcommit.inditex.dev/v1alpha1
kind: OvercommitClass
metadata:
  name: high
spec:
  cpuOvercommit: 0.2      # request = 20% of CPU limit
  memoryOvercommit: 0.8  # request = 80% of memory limit
  excludedNamespaces: ".*^(openshift|k8s-overcommit|kube).*).*"
  isDefault: true
```

Selection is label-driven with a clear precedence: a **pod** label wins over a **namespace** label, which wins over the **default** class. That lets you overcommit aggressively on a batch-Spark namespace while leaving latency-sensitive services untouched. Installation is Helm or OLM, and system namespaces (`kube-*`, the operator's own, etc.) are excluded out of the box so you do not accidentally starve the control plane.

What about Spark's integer-cores constraint?

Spark's integer-cores constraint does not block this approach: the restriction lives in Spark's *configuration layer*, not in Kubernetes. This was my first worry, so I [asked the maintainers directly](#): if Spark refuses fractional cores, will it choke when the request becomes 0.4? The answer is the elegant part of this approach.

The operator rewrites the resulting **Pod** object at admission time, and Kubernetes is perfectly happy with fractional CPU on a Pod, even though `spark.executor.cores` will not let you express it. Spark never sees the rewritten value; it only set the limit.

One practical caveat from that thread: pick your ratios and limits so the resulting request is sensible. With `cpuOvercommit: 0.5`, a 4-core limit yields a clean 2-core request; a 3-core limit yields 1.5. Kubernetes accepts both, but choosing limits that land on tidy numbers keeps capacity math easy to reason about.

What actually changed: Pending executors started scheduling

The immediate, visible effect is the one I cared about: **pods that were stuck Pending started scheduling**. Because each executor now reserves a fraction of its old footprint, the scheduler fits roughly twice as many executors onto the same nodes, and the reservation math finally lines up with the ~50% real CPU usage those executors were always going to have.

Nothing about the workload changed. No node was added. The cluster simply stopped lying about being full. The same hardware now absorbs the queued work instead of leaving it pending while CPUs idle.

The trade-offs you must understand first

Overcommit is a real engineering decision, not a free switch: pods move from Guaranteed to Burstable QoS and get evicted first under node pressure, and overcommitted memory can get OOM-killed. It compensates for over-reservation instead of right-sizing your executors, and its mutating admission

webhook sits on every pod creation. A few things you have to be clear-eyed about:

- **Your QoS class changes.** Pods move from Guaranteed to Burstable (request < limit). Under node pressure, Burstable pods are evicted before Guaranteed ones. For batch Spark that is usually acceptable; for stateful or latency-critical services it may not be.
- **CPU is forgiving, memory is not.** CPU is a compressible resource: overcommit it and the worst case is throttling. Memory is incompressible: if overcommitted pods all climb toward their limits at once, the kernel OOM-kills them. Keep memoryOvercommit conservative (or close to 1.0) and be far more aggressive on CPU.
- **It is a cluster lever, not a tuning replacement.** Overcommit compensates for workloads that over-reserve; it does not right-size them. If your executors are genuinely mis-sized, fix that too: overcommit just stops you from paying for the gap in the meantime.
- **The webhook is in the critical path.** A mutating admission webhook sits on every pod creation. Validate its availability and failure policy before you trust it in production.

Where this fits

This remediation fits any Spark/Kubernetes compute plane stuck at low utilization: one of the most common and most expensive patterns I see when reviewing data platforms. The fix is rarely "buy more nodes"; it is usually understanding where reservations and reality have diverged, and applying the right lever. Overcommit is one of them.

If your data platform is over-provisioned, slow, or hitting scheduling walls and nobody can quite explain why, that is exactly the kind of problem I diagnose. See the [data platform performance audit](#), or [book an expert call](#) if you want a focused hour on one specific bottleneck.

Update 2026-09-29. This post is now part of [Performance and Cost Engineering for Data Platforms](#), the field guide to performance and cost engineering.

Related posts

- [Performance and Cost Engineering for Data Platforms](#)
- [Apache Spark smoke tests on Kubernetes](#)
- [Benchmark Best Practices, Part 1: Amdahl's law caps your scaling claim](#)
- [When the lakehouse works but the operating model breaks](#)

PDF snapshot of the public article. The online version is authoritative; links remain clickable.