



Kueue Quotas, Cohorts and Fair Share for GPU Clusters

Plan and build private AI compute with GPU Smith. We connect workload requirements with GPU hardware, networking, deployment and operating decisions so your infrastructure fits the work it must perform.

Published by GPU Smith · 2026-10-29 · kueue clusterqueue gpu quota · kueue cohort borrowing · kueue fair sharing preemption · kueue resourceflavor gpu types · kubernetes gpu quota multi-tenant · kueue borrowinglimit lendinglimit

Original article: <https://gpusmith.com/articles/en/kueue-quotas-fair-share-gpu-multi-tenancy>



In this article

1. The short answer
 2. The multi-tenant GPU quota problem
 3. The Kueue object model
 4. ResourceFlavors for GPU models
 5. ClusterQueue quotas
 6. Cohorts and hierarchy
 7. Preemption policies
 8. Worked example: three teams, 64 H100s
 9. Fair sharing
 10. Reference designs
 11. Accounting and chargeback
 12. Failure modes
 13. Rollout checklist
 14. Sources
-

The short answer

To share a GPU cluster between teams with Kueue, create one ResourceFlavor per GPU model and give each team a ClusterQueue with a nominalQuota of GPUs. Put the ClusterQueues in one cohort so idle GPUs can be borrowed. Cap bursts with borrowingLimit, protect a floor with lendingLimit, and set reclaimWithinCohort so teams can take their quota back. Add fair sharing when first-come, first-served borrowing stops being acceptable. As of 2026-10-28 the latest release is Kueue v0.20.3, published 2026-10-22, alongside v0.19.10 on the previous minor line ([Kueue releases](#)). Since v0.20.0 Kueue no longer serves `kueue.x-k8s.io/v1beta1` ([v0.20.0 release notes](#)), so every manifest below is v1beta2.

This article covers quota policy. Gang placement is covered in [gang scheduling on Kubernetes](#), and splitting one device between workloads in [MIG, MPS and time-slicing](#).

The multi-tenant GPU quota problem

Hard per-team GPU slices leave GPUs idle while other queues grow. No slices at all lets the first submitter take the cluster. Kueue "manages quotas and how jobs consume them": it decides when a job waits, is admitted or is preempted. Pod-to-node placement stays with kube-scheduler ([Kueue overview](#)). For each team and GPU model you are really choosing three things:

- **a guaranteed floor**, the GPUs the team gets back quickly;
- **a burst cap**, how far above the floor it may go when others are idle;
- **a reclaim cost**, the running work you are willing to destroy to hand quota back.

The Kueue object model

Object	Scope	Role in a GPU design
ResourceFlavor	Cluster	One per GPU model or node pool; binds quota to nodes (ResourceFlavor)
ClusterQueue	Cluster	"Governs a pool of resources, defining usage limits and Fair Sharing rules" (ClusterQueue); usually one per tenant
LocalQueue	Namespa ce	What users submit to; points at one ClusterQueue (LocalQueue)
Workload	Namespa ce	Unit of admission; created by Kueue for each Job (Workload)
Cohort	Cluster	Group of ClusterQueues that share quota; can hold its own quota and form a tree (Cohort)
WorkloadPriorityClass	Cluster	Queueing and preemption priority, independent of pod priority (WorkloadPriorityClass)

Jobs opt in through the labels `kueue.x-k8s.io/queue-name` and, optionally, `kueue.x-k8s.io/priority-class` ([Run a Job](#); [WorkloadPriorityClass](#)).

ResourceFlavors for GPU models

Every NVIDIA GPU is requested as `nvidia.com/gpu`. The Kueue docs call this out: with T4 and A100 GPUs in one cluster, the system "may not differentiate between them" ([ResourceFlavor](#)). The fix is one flavor per hardware type:

- `spec.nodeLabels` ties a flavor to nodes. When a Workload is admitted, Kueue adds these labels to its pod template's `nodeSelector`, so the pods run where the quota was charged.
- `spec.tolerations` is injected into admitted pods, which lets them land on tainted GPU nodes automatically.
- `spec.nodeTaints` works the other way round. Only Workloads that already tolerate the taint can use the flavor, and Kueue does not inject that toleration. This suits pools that users must opt into (same source).

NVIDIA GPU Feature Discovery labels each node with `nvidia.com/gpu.product`, with example values such as `A100-SXM4-40GB` and `Tesla-T4`. The value "may be modified by the device plugin if a sharing strategy is employed", and it carries MIG information when MIG is enabled ([GPU Feature Discovery README](#)). GFD builds it from the NVML device name with spaces turned into hyphens ([resource.go](#)), so an H100 SXM 80 GB node reports `NVIDIA-H100-80GB-HBM3` and an A100 SXM4 80 GB node `NVIDIA-A100-SXM4-80GB`. Copy the exact string from `kubectl get nodes -L nvidia.com/gpu.product`. Check it again after sharing or driver changes; see [GPU Operator and DRA upgrade gate](#).

```
# Assumption: GPU nodes are tainted nvidia.com/gpu:NoSchedule.
# NVIDIA-H100-80GB-HBM3 is the H100 SXM 80 GB value; copy yours from the node labels.
apiVersion: kueue.x-k8s.io/v1beta2
kind: ResourceFlavor
```

```

metadata:
  name: h100
spec:
  nodeLabels:
    nvidia.com/gpu.product: NVIDIA-H100-80GB-HBM3
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule

```

Create an a100 flavor the same way.

ClusterQueue quotas

A ClusterQueue lists resourceGroups. Each group has coveredResources and an ordered list of flavors with a quota per resource. When a flavor maps to a node pool, put cpu, memory and nvidia.com/gpu in the same group so all three are charged to the same flavor (ClusterQueue). The CRD also requires every flavor to list every covered resource ([clusterqueue_types.go, v0.20.3](#)). The three per-resource fields are ([v1beta2 API reference](#)):

- **nominalQuota**: what the queue can use at any time. It should reflect capacity "after discounting resources consumed by system components and pods not managed by kueue".
- **borrowingLimit**: the most the queue may borrow from unused quota in its cohort, which caps it at $\text{nominalQuota} + \text{borrowingLimit}$. If unset, borrowing is unlimited. It must be null when the queue has no cohort.
- **lendingLimit**: the most unused quota the queue lends out. The queue keeps $\text{nominalQuota} - \text{lendingLimit}$ for exclusive use. The webhook rejects a lendingLimit larger than nominalQuota ([clusterqueue_webhook.go, v0.20.3](#)). The docs list lending limits as stable since v0.17 (ClusterQueue).

Kueue tries flavors in list order and takes the first one with enough unused quota in the queue or its cohort. Listing h100 before a100 therefore makes A100 the fallback. By default (`whenCanBorrow: MayStopSearch`), Kueue stops at the first flavor that fits by borrowing. Set `flavorFungibility.whenCanBorrow: TryNextFlavor` if teams should prefer in-quota A100s over borrowed H100s (same source). `queueingStrategy` orders Workloads by priority, then creation time. Under `StrictFIFO`, an older Workload that does not fit blocks newer ones. Under `BestEffortFIFO`, the default, newer Workloads that fit can go ahead.

Cohorts and hierarchy

ClusterQueues join a cohort through `spec.cohortName`. Some prose on the ClusterQueue page still says `.spec.cohort`, but the v1beta2 examples and the API reference use `cohortName`. A request fits by borrowing when it exceeds the queue's unused `nominalQuota` but stays within both (1) the cohort's total unused `nominalQuota` and (2) the queue's unused `nominalQuota + borrowingLimit`. A queue can borrow only for flavors it defines, even when it defines them at `nominalQuota: 0` (ClusterQueue). The "dedicated and fallback

flavors" layout in [Administer Cluster Quotas](#) uses this rule. Each team gets dedicated quota on one flavor and zero on another, and a separate shared ClusterQueue owns that second flavor. It is a clean way to run a shared A100 pool next to dedicated H100 slices.

A Cohort can also be an object in its own right. Its `nominalQuota` is "additional resources on top of those defined by ClusterQueues within the Cohort", which works as a shared pool ([Cohort](#)). Cohorts form trees through `spec.parentName`. On a child Cohort, `borrowingLimit` and `lendingLimit` limit what the whole subtree borrows from or lends to its parent. These limits are rejected on a root Cohort ([API reference](#)). A typical tree has one root per cluster, a child Cohort per business unit, and team ClusterQueues underneath.

Preemption policies

Borrowing does not return quota on its own. `reclaimWithinCohort` defaults to `Never`, so a lender waits until the borrower's jobs finish ([ClusterQueue](#)).

Field	Values	Effect
<code>reclaimWithinCohort</code>	<code>Never</code> (default), <code>LowerPriority</code> , <code>Any</code>	Preempt Workloads in other ClusterQueues that are above their nominal quota
<code>borrowWithinCohort</code>	<code>policy</code> : <code>Never</code> (default) / <code>LowerPriority</code> , <code>maxPriorityThreshold</code>	Preempt even when the incoming Workload must borrow; classic preemption only; needs <code>reclaimWithinCohort</code> other than <code>Never</code>
<code>withinClusterQueue</code>	<code>Never</code> (default), <code>LowerPriority</code> , <code>LowerOrNewerEqualPriority</code>	Preempt lower-priority (or newer equal-priority) Workloads in the same queue

Source: [ClusterQueue](#). Under classic preemption, "ClusterQueues in a cohort borrow resources in a first-come first-served fashion". Victims come first from borrowing queues, then by lowest priority, then by most recent admission, and Kueue trims the set to the minimum that still fits. Preempted Workloads receive `Evicted` and `Preempted` conditions that identify the preemptor ([Preemption](#)). For GPU training, preemption costs whatever ran since the last checkpoint. Make checkpoint frequency part of each tenant's quota contract.

Worked example: three teams, 64 H100s

All numbers are **assumptions**. The cluster has 64 H100 GPUs (8 nodes × 8) in flavor `h100`, and the A100 pool is full, so nothing falls back to it. Three ClusterQueues share cohort `ml-platform`, all with `reclaimWithinCohort`: `Any`. Research runs 8-GPU jobs.

ClusterQueue	<code>nominalQuota</code>	<code>borrowingLimit</code>	<code>lendingLimit</code>	Exclusive floor
research	32	16	unset	0
product-training	24	unset	12	$24 - 12 = 12$
eval	8	8	unset	0
Total	64			

The nominal quotas add up to the physical 64 GPUs. Kueue does not enforce that, but if the sum is too high, admitted pods sit unschedulable.

Step 0: research bursts. Product-training is using 6 GPUs and eval 0.

- Product-training can lend 12, because its 6 GPUs fit inside its 12-GPU exclusive floor. Eval can lend 8. Available to borrow: $12 + 8 = 20$.
- Research ceiling: $32 + \min(16, 20) = 48$ GPUs, which is six 8-GPU jobs.
- Allocated: $48 + 6 + 0 = 54$. Idle: 10. Six are product-training's unused floor ($12 - 6$) and four are cut off by research's borrowingLimit ($20 - 16$).

Step 1: product-training submits a 16-GPU job. $6 + 16 = 22 \leq 24$, so the job is within nominal and may reclaim. Ten GPUs are free, so 6 more are needed. Research is the only queue above nominal ($48 > 32$), and preempting one 8-GPU research job is enough. Result: research 40, product-training 22, eval 0. Total 62, free 2.

Step 2: eval submits an 8-GPU job. That is within its nominal of 8. Six more GPUs are needed, so one more research job is preempted. Result: research 32 (borrowing 0), product-training 22, eval 8. The 2 free GPUs are product-training's, and research may borrow them because $12 - (22 - 12) = 2$ are still lendable.

With the defaults (`reclaimWithinCohort: Never`), the 16-GPU job in step 1 waits until two research jobs finish. On multi-day runs, that wait is where most "Kueue took our GPUs" complaints come from.

`lendingLimit` buys latency, not throughput. Product-training can always start 6 more GPUs instantly, with no preemption and no lost checkpoints, and pays for it with 6 idle GPUs in step 0. This walk-through applies the documented rules by hand. It is not a measurement, so replay it on a staging cluster.

The research queue, with CPU and memory at an assumed 12 CPU and 120Gi per GPU:

```
apiVersion: kueue.x-k8s.io/v1beta2
kind: ClusterQueue
metadata:
  name: research
spec:
  cohortName: ml-platform
  namespaceSelector:
    matchLabels:
      tenant: research
  queueingStrategy: BestEffortFIFO
  preemption:
    reclaimWithinCohort: Any
    withinClusterQueue: LowerPriority
  resourceGroups:
  - coveredResources: ["cpu", "memory", "nvidia.com/gpu"]
    flavors:
    - name: h100
      resources:
      - name: cpu
        nominalQuota: 384
      - name: memory
```

```

    nominalQuota: 3840Gi
  - name: nvidia.com/gpu
    nominalQuota: 32
    borrowingLimit: 16
  - name: a100          # fallback: borrowed from a shared A100 ClusterQueue
    resources:
      - name: cpu
        nominalQuota: 0
      - name: memory
        nominalQuota: 0
      - name: nvidia.com/gpu
        nominalQuota: 0
---
apiVersion: kueue.x-k8s.io/v1beta2
kind: LocalQueue
metadata:
  namespace: research
  name: research-gpu
spec:
  clusterQueue: research

```

Product-training uses `nominalQuota: 24` and `lendingLimit: 12` on `nvidia.com/gpu`. Give `cpu` and `memory` proportional lending limits (144 and 1440Gi) so the floor reserves whole nodes' worth of resources, not just GPUs.

Fair sharing

Classic preemption guarantees nominal quota but leaves idle quota with whoever borrowed it first. Fair sharing preempts "until the preempting ClusterQueue obtains an equal or weighted share of the borrowable resources". It has been stable since v0.7 ([Preemption](#)). In the v1beta2 Configuration, the `fairSharing` block holds only `preemptionStrategies` ([Configuration API](#)). v0.15 removed the old `enable` field and made `preemptionStrategies` required ([v0.15.0 release notes](#)). The manager turns fair sharing on when the block is present ([main.go](#), [v0.20.3](#)).

```

apiVersion: config.kueue.x-k8s.io/v1beta2
kind: Configuration
metrics:
  enableClusterQueueResources: true
fairSharing:
  preemptionStrategies: [LessThanOrEqualToFinalShare, LessThanInitialShare]

```

Only three lists are accepted: `[LessThanOrEqualToFinalShare]`, `[LessThanInitialShare]`, or both in that order. Any other list "fails configuration validation and Kueue does not start". Fair sharing works with `reclaimWithinCohort` and `withinClusterQueue`, but `borrowWithinCohort` is classic-only ([Preemption](#)).

A queue's share is "the maximum of the ratios of usage above nominal quota to the lendable resources in the Cohort, among all the resources", divided by `spec.fairSharing.weight` ([API reference](#)). This is a Dominant Resource Share, or DRS ([Fair Sharing](#)). Kueue admits from the lowest share first and preempts from the

highest. In v0.20.3 the ratio is computed per thousand, and `.status.fairSharing.weightedShare` reports it rounded up to an integer ([fair_sharing.go](#), v0.20.3). A weight of zero makes any borrowing an infinite share.

Worked example (assumptions). One flavor, 64 GPUs and no lending limits, so `lendable` = 64. Queues A and B have nominal 24 each, and C has nominal 16 and is idle. A runs 36 GPUs (borrowing 12) and B runs 28 (borrowing 4), so the cluster is full. Weights are 1.

- Share A = $12/64 = 187.5$ per thousand. Share B = $4/64 = 62.5$.
- **B submits a 4-GPU job.** B's share with the job admitted is $8/64 = 125$. A's share without one of its 4-GPU jobs is also $8/64 = 125$. Since $125 \leq 125$, `LessThanOrEqualToFinalShare` holds and the A job is preempted. Both queues end at 32 GPUs.
- **B submits an 8-GPU job instead.** B with the job: $12/64 = 187.5$. A after losing 8 GPUs: 62.5. $187.5 \leq 62.5$ fails, and `LessThanInitialShare` ($187.5 < 187.5$) also fails, so B waits. Flipping the imbalance gains nothing. The Fair Sharing page proves that two workloads in different `ClusterQueues` cannot keep preempting each other.
- **With A at weight 2,** B's 4-GPU job would need $125 \leq 125/2 = 62.5$, which fails. A keeps its borrowed GPUs.

In a Cohort tree, the strategies compare the queues' "AlmostLCA" ancestors rather than the queues themselves ([Preemption](#)). Cohort weights of 0.75 and 0.25 make two orgs "trend towards using 75%" and 25% of common resources ([Cohort](#)).

Admission Fair Sharing (AFS) is for many `LocalQueues` feeding one `ClusterQueue`. It orders Workloads by the decayed historical usage of their `LocalQueue`, and an entry penalty stops a tenant from flooding the queue before its usage catches up. It is beta and on by default since v0.15. A `ClusterQueue` opts in with `spec.admissionScope.admissionMode: UsageBasedAdmissionFairSharing`, and `usageHalfLifeTime`, `usageSamplingInterval`, `resourceWeights` and a per-`LocalQueue` `fairSharing.weight` tune it ([Admission Fair Sharing](#)). In v0.20, usage counts from quota reservation, and disabling `AdmissionFairSharing` without also disabling `AdmissionFairSharingAnchorAtQuotaReservation` is rejected at startup ([v0.20.0 release notes](#)).

Reference designs

Pattern	Cohort	Limits	Preemption	Use when
Static slices	none	n/a (borrowingLimit must be null)	<code>withinClusterQueue</code>	Contractual isolation; accept idle GPUs
Borrow-heavy	flat	none	<code>reclaimWithinCohort: Any</code>	Research with frequent checkpoints
Guaranteed floor	flat	<code>lendingLimit</code> on latency-sensitive, <code>borrowingLimit</code> on bursty	<code>reclaimWithinCohort: Any</code>	Mixed production and research (example above)

Pattern	Cohort	Limits	Preemption	Use when
Strict priority	flat	as needed	<code>reclaimWithinCohort: LowerPriority + borrowWithinCohort + WorkloadPriorityClasses</code>	Release-critical runs jump the queue; classic only
Weighted fair share	tree with <code>fairSharing.weight</code>	optional	<code>reclaimWithinCohort: Any + strategies</code>	Many teams contesting idle capacity
Shared queue + AFS	single queue	n/a	<code>withinClusterQueue</code>	Many small users or notebooks

These patterns combine. A common shape is a weighted Cohort tree at the top, floors on production queues, and AFS inside a shared research queue.

Accounting and chargeback

Kueue measures quota held, not GPU work done. A job holding 8 GPUs at low SM activity is still charged 8, so pair Kueue's numbers with device telemetry; see [GPU utilization targets](#). Four per-flavor gauges are exported only when `metrics.enableClusterQueueResources` is set: `kueue_cluster_queue_resource_usage`, `kueue_cluster_queue_nominal_quota`, `kueue_cluster_queue_borrowing_limit` and `kueue_cluster_queue_lending_limit`. They are labelled cohort, cluster_queue, flavor, resource and replica_role ([Kueue metrics](#)). The docs still mark LocalQueue metrics such as `kueue_local_queue_resource_usage` as alpha behind `LocalQueueMetrics`, but in the v0.20.3 source that gate has been Beta and on by default since v0.17 ([kube_features.go](#), v0.20.3).

Reserved GPU-hours per queue and flavor over 30 days:

```
sum by (cluster_queue, flavor) (
  sum_over_time(kueue_cluster_queue_resource_usage{resource="nvidia.com/gpu", replica_role!="follower"}
    [30d:5m])
) * 5 / 60
```

Each 5-minute sample is a GPU count. Multiplying the sum of samples by 5 gives GPU-minutes, and dividing by 60 gives GPU-hours. For borrowed GPU-hours, subtract `kueue_cluster_queue_nominal_quota` and clamp at zero. `replica_role` is leader, follower or standalone ([metrics.go](#), v0.20.3); the filter drops followers so an HA install is not double counted. For tenant SLOs, track `kueue_admission_wait_time_seconds` and `kueue_preempted_workloads_total` (same source).

Failure modes

- **Lent quota never returns.** `reclaimWithinCohort` was left at Never.
- **Head-of-line blocking.** Under StrictFIFO, one 64-GPU job that does not fit blocks everything behind it. The reverse risk under BestEffortFIFO is our inference: small jobs keep fitting and large ones wait. Use priorities or a separate queue for large jobs.

- **Borrowing storms.** Without `borrowingLimit`, one team can absorb the whole idle pool, and the owners' return then preempts many large jobs at once. Cap borrowing at what you can afford to lose.
- **Quota that does not match the hardware.** Nominal sums above allocatable GPUs, or `nodeLabels` that no longer match nodes, produce admitted pods that stay Pending. The [gang scheduling article](#) covers `waitForPodsReady` and topology.
- **Configuration that will not start.** An unsupported strategy list, `borrowWithinCohort` combined with fair sharing, or disabling AFS alone in v0.20 is rejected ([Preemption](#); [v0.20.0 release notes](#)).
- **Zero weights.** A queue with weight 0 always loses as soon as it borrows.

Rollout checklist

1. Pin a version: v0.20.3, or v0.19.10 on the previous minor, as of October 2026. Read every .0 release note you cross, and run the v1beta2 migration script if you are coming from v1beta1 ([v0.20.0 release notes](#)).
2. Take an inventory of GPU node labels and taints, then create one `ResourceFlavor` per model and confirm that it matches real nodes.
3. Set nominal quotas from allocatable GPUs minus system usage, and check the sum for each flavor.
4. Write each tenant's contract down: nominal quota, burst cap, floor, preemption exposure and checkpoint interval.
5. Enable `enableClusterQueueResources` and build the GPU-hour and admission-wait dashboards before go-live.
6. Replay a representative week of jobs in staging, including a reclaim drill like steps 1 and 2 above. Kueue has no built-in dry-run admission mode that we could verify.
7. Onboard tenants one namespace at a time with `namespaceSelector`, and keep `stopPolicy`: Hold ready for any queue that misbehaves ([ClusterQueue](#)).
8. Add fair sharing only after classic reclaim works and tenants have agreed on weights.

GPU Smith helps platform teams design and review GPU quota and fairness policies for private clusters; [contact us](#).

Sources

1. Kueue release v0.20.3 (with v0.19.10). [kubernetes-sigs/kueue](#), GitHub, 2026-10-22. <https://github.com/kubernetes-sigs/kueue/releases/tag/v0.20.3>
2. Kueue v0.20.0 release notes. GitHub, 2026-09-30. <https://github.com/kubernetes-sigs/kueue/releases/tag/v0.20.0>
3. Kueue v0.15.0 release notes. GitHub, 2025-11-28. <https://github.com/kubernetes-sigs/kueue/releases/tag/v0.15.0>
4. Overview. Kueue docs. <https://kueue.sigs.k8s.io/docs/overview/>
5. Cluster Queue. Kueue docs, modified 2026-08-19. https://kueue.sigs.k8s.io/docs/concepts/cluster_queue/

6. Resource Flavor. Kueue docs, modified 2026-10-07.
https://kueue.sigs.k8s.io/docs/concepts/resource_flavor/
7. Local Queue. Kueue docs, modified 2026-08-19. https://kueue.sigs.k8s.io/docs/concepts/local_queue/
8. Workload. Kueue docs. <https://kueue.sigs.k8s.io/docs/concepts/workload/>
9. Cohort. Kueue docs, modified 2026-08-19. <https://kueue.sigs.k8s.io/docs/concepts/cohort/>
10. Preemption. Kueue docs, modified 2026-09-30. <https://kueue.sigs.k8s.io/docs/concepts/preemption/>
11. Fair Sharing. Kueue docs, modified 2025-09-23. https://kueue.sigs.k8s.io/docs/concepts/fair_sharing/
12. Admission Fair Sharing. Kueue docs, modified 2026-09-22.
https://kueue.sigs.k8s.io/docs/concepts/admission_fair_sharing/
13. Workload Priority Class. Kueue docs. https://kueue.sigs.k8s.io/docs/concepts/workload_priority_class/
14. Administer Cluster Quotas. Kueue docs, modified 2026-09-25.
https://kueue.sigs.k8s.io/docs/tasks/manage/administer_cluster_quotas/
15. Run A Kubernetes Job. Kueue docs. <https://kueue.sigs.k8s.io/docs/tasks/run/jobs/>
16. Kueue API v1beta2 reference. Kueue docs. <https://kueue.sigs.k8s.io/docs/reference/kueue.v1beta2/>
17. Kueue Configuration API v1beta2 reference. Kueue docs.
<https://kueue.sigs.k8s.io/docs/reference/kueue-config.v1beta2/>
18. Prometheus Metrics. Kueue docs. <https://kueue.sigs.k8s.io/docs/reference/metrics/>
19. clusterqueue_types.go, v0.20.3. GitHub. https://github.com/kubernetes-sigs/kueue/blob/v0.20.3/apis/kueue/v1beta2/clusterqueue_types.go
20. clusterqueue_webhook.go, v0.20.3. GitHub. https://github.com/kubernetes-sigs/kueue/blob/v0.20.3/pkg/webhooks/clusterqueue_webhook.go
21. fair_sharing.go, v0.20.3. GitHub. https://github.com/kubernetes-sigs/kueue/blob/v0.20.3/pkg/cache/scheduler/fair_sharing.go
22. main.go, v0.20.3. GitHub. <https://github.com/kubernetes-sigs/kueue/blob/v0.20.3/cmd/kueue/main.go>
23. kube_features.go, v0.20.3. GitHub. https://github.com/kubernetes-sigs/kueue/blob/v0.20.3/pkg/features/kube_features.go
24. GPU Feature Discovery README. NVIDIA k8s-device-plugin, GitHub. <https://github.com/NVIDIA/k8s-device-plugin/blob/main/docs/gpu-feature-discovery/README.md>
25. internal/lm/resource.go (product label sanitization). NVIDIA k8s-device-plugin, GitHub.
<https://github.com/NVIDIA/k8s-device-plugin/blob/main/internal/lm/resource.go>
26. pkg/metrics/metrics.go, v0.20.3 (replica_role values). GitHub. <https://github.com/kubernetes-sigs/kueue/blob/v0.20.3/pkg/metrics/metrics.go>

Source links

Links cited in this article, in order of first appearance. Full addresses are provided for readers using extracted text.

1. <https://github.com/kubernetes-sigs/kueue/releases/tag/v0.20.3>
2. <https://github.com/kubernetes-sigs/kueue/releases/tag/v0.20.0>
3. <https://kueue.sigs.k8s.io/docs/overview>
4. https://kueue.sigs.k8s.io/docs/concepts/resource_flavor
5. https://kueue.sigs.k8s.io/docs/concepts/cluster_queue

6. https://kueue.sigs.k8s.io/docs/concepts/local_queue
7. <https://kueue.sigs.k8s.io/docs/concepts/workload>
8. <https://kueue.sigs.k8s.io/docs/concepts/cohort>
9. https://kueue.sigs.k8s.io/docs/concepts/workload_priority_class
10. <https://kueue.sigs.k8s.io/docs/tasks/run/jobs>
11. <https://github.com/NVIDIA/k8s-device-plugin/blob/main/docs/gpu-feature-discovery/README.md>
12. <https://github.com/NVIDIA/k8s-device-plugin/blob/main/internal/lm/resource.go>
13. https://github.com/kubernetes-sigs/kueue/blob/v0.20.3/apis/kueue/v1beta2/clusterqueue_types.go
14. <https://kueue.sigs.k8s.io/docs/reference/kueue.v1beta2>
15. https://github.com/kubernetes-sigs/kueue/blob/v0.20.3/pkg/webhooks/clusterqueue_webhook.go
16. https://kueue.sigs.k8s.io/docs/tasks/manage/administer_cluster_quotas
17. <https://kueue.sigs.k8s.io/docs/concepts/preemption>
18. <https://kueue.sigs.k8s.io/docs/reference/kueue-config.v1beta2>
19. <https://github.com/kubernetes-sigs/kueue/releases/tag/v0.15.0>
20. <https://github.com/kubernetes-sigs/kueue/blob/v0.20.3/cmd/kueue/main.go>
21. https://kueue.sigs.k8s.io/docs/concepts/fair_sharing
22. https://github.com/kubernetes-sigs/kueue/blob/v0.20.3/pkg/cache/scheduler/fair_sharing.go
23. https://kueue.sigs.k8s.io/docs/concepts/admission_fair_sharing
24. <https://kueue.sigs.k8s.io/docs/reference/metrics>
25. https://github.com/kubernetes-sigs/kueue/blob/v0.20.3/pkg/features/kube_features.go
26. <https://github.com/kubernetes-sigs/kueue/blob/v0.20.3/pkg/metrics/metrics.go>

THE PUBLISHER

About GPU Smith

GPU Smith provides independent engineering and research for private AI infrastructure. We help technical buyers and operators reason about GPU workload sizing, hardware selection, procurement, deployment and inference operations, from the compute system to the networking, power and cooling requirements around it.

Start with the workload

Useful infrastructure decisions begin with the models, data, throughput, latency and operating constraints a team actually has. GPU Smith helps connect those requirements with system design and deployment choices. The goal is a privately controlled AI environment whose capacity and operational demands are understood before a purchasing decision.

Hardware and supplier research

Our public reference library covers GPUs, complete systems and networking components. The server-vendor directory and manufacturing research help teams investigate suppliers, compare options and follow sources behind technical claims. We distinguish manufacturer specifications from measured benchmarks and advertised capabilities from tested configurations. Reference pages are research resources, not a live inventory listing or a binding equipment quote.

Deployment and operations

GPU Smith's engineering scope includes infrastructure integration, networking, power, cooling and the practical operation of inference workloads. Our published articles explain the tradeoffs behind deployment, procurement and ongoing operations so teams can ask better questions and document decisions.

Work with GPU Smith

Explore the [hardware library](#), [GPU references](#), [networking references](#) and [vendor research](#). [Contact GPU Smith](#) with your workloads and deployment constraints to discuss a project and confirm current scope, pricing and availability.

Inclusion of a manufacturer or product in our research does not imply a partnership, certification or endorsement.

Website: <https://gpusmith.com>

This article is educational. Verify consequential medical, legal, financial, regulatory and technical decisions with appropriate professionals and the cited primary sources. Company services are described separately from the article's research findings.