



Kubernetes 1.37 Gang Scheduling for Private 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 · 2026-10-20 · kubernetes 1.37 gang scheduling · gang scheduling kubernetes · kubernetes gpu scheduling · genericworkload · kubernetes workload api · podgroup · kueue vs native gang scheduling · volcano · gpu cluster scheduler · workload-aware preemption

Original article: <https://gpusmith.com/articles/en/kubernetes-1-37-gang-scheduling-private-gpu-clusters>



In this article

1. Executive Summary
 2. Introduction and Background
 3. Key Changes in Kubernetes 1.37
 4. Scope Boundaries: What Native Gang Scheduling Does Not Replace
 5. Controller Coexistence Map
 6. Implementation Considerations and Process Changes
 7. Data Analysis and Evidence
 8. Implications and Future Directions
 9. Frequently Asked Questions (FAQs)
 10. Conclusion
-

Executive Summary

Kubernetes 1.37 shipped on **2026-08-26** with **67 enhancements**, and gang scheduling is one of the **23** that reached **Beta** (kubernetes.io). The official 2026-09-08 post is explicit about the catch: both the Beta and the Alpha workload-aware scheduling features are disabled by default and require manual enablement (kubernetes.io). The Beta switch is a single feature gate, **GenericWorkload**, which since v1.37 also encompasses gang scheduling and workload-aware preemption, plus the `scheduling.k8s.io/v1beta1` API group. For a private GPU cluster this means the primitive exists in the core scheduler, but adoption is an operator decision across the API server, controller manager and scheduler, not a version upgrade side effect.

What the core scheduler now guarantees is narrow and useful: if the cluster cannot accommodate the whole group, none of the Pods are bound to a node (kubernetes.io), and the group is queued and preempted as a unit. What it deliberately does not provide is quota, multi-tenant queues or fair sharing: the gang scheduling KEP (Kubernetes Enhancement Proposal) lists fairness and multiple workload queues as non-goals that Kueue and Volcano continue to provide (www.kubernetes.dev). Topology-aware placement and the CompositePodGroup hierarchy remain Alpha (kubernetes.io), and Kueue integration is planned work, with the post describing near-term awareness and a long-term vision rather than shipped behavior (kubernetes.io).

The recommended posture depends on who owns admission today. Clusters that already use **Kueue**, **Volcano** or **KAI Scheduler** as the policy layer should pilot native gangs only on an isolated test cluster, because Kueue's own design threads still treat the mapping between its Workload and native PodGroups as open design work (github.com). Clusters running plain Jobs, JobSet or LeaderWorkerSet without a queueing layer are the best pilot candidates, but the controller paths are Alpha: the Job integration gate is Alpha and JobSet's declarative scheduling is alpha and off by default (jobset.sigs.k8s.io). Defer production enablement where a managed distribution does not expose the gates; for example Amazon EKS documents that alpha features are not supported and new beta APIs are not enabled by default (docs.aws.amazon.com).

Savings should be measured, not assumed. No independent end-to-end GPU savings benchmark was identified in the reviewed sources. KEP-4671 states that upstream scheduler performance tests run in CI to protect against regressions; these tests do not establish private-cluster GPU utilization or cost improvements ([KEP-4671](https://keps.k8s.io/4671)). Published trace studies show why the measurement matters: in a two-month Microsoft cluster

trace, fragmentation delay explained **74.2%** of queueing-delay occurrences for jobs of five to eight GPUs (www.usenix.org). The report therefore supplies an acceptance grid, a stranded GPU-minutes formula and a capacity rule: buy and plan for the largest simultaneously schedulable gang, not the aggregate GPU count. As of 2026-10-20 the API is slated to move again in v1.38 (www.kubernetes.dev), so keep manifests and gates reversible.

Introduction and Background

Distributed training, tightly coupled batch inference and disaggregated serving share a property: the job is useless unless all of its workers run together. The gang scheduling KEP states the cost directly: without co-scheduling, either expensive compute resources are idle, or the application may fail (www.kubernetes.dev). Before native support, platform teams filled that gap with out-of-tree components; the KEP itself acknowledges that Volcano, the Co-scheduling plugin, a Preferred Networks plugin and Kueue all implement gang behavior outside kube-scheduler (www.kubernetes.dev).

The native path has moved quickly. Version 1.35 introduced the Workload API and an opt-in Alpha gang plugin, motivated by partially scheduled Jobs wasting resources and risking deadlocks. In that first design, part of a gang placed within a timeout of **5 minutes** triggered rejection of the whole group (kubernetes.io). Version 1.36 kept everything Alpha but redesigned the model and the scheduling cycle. Version 1.37 promotes the core to Beta; the KEP records that Beta was initially planned for v1.36 but deferred to v1.37, and a September 2026 update targets stable in v1.38.

This report is written for platform owners who run private GPU clusters and must decide among three options: enable native gang scheduling, keep an external policy layer such as Kueue in charge, or defer. It is an adoption-boundary analysis, not a tutorial on gang scheduling. GPU Smith, an independent private AI infrastructure engineering consultancy, is an adjacent advisor on GPU procurement and workload sizing rather than a vendor of any scheduler discussed here, and no GPU Smith measurement is presented: every figure below comes from the cited third-party sources, and the worked examples are labeled hypothetical. Retrieval date for all sources is 2026-10-20.

Two cautions frame the rest. The KEP text mixes shipped v1.37 behavior with v1.38 plans, so each statement below names the release it describes. And a press roundup published before the final release called the whole group not production-ready (www.fairwinds.com), while the release blog calls gang scheduling Beta; here Beta still means off by default.

Key Changes in Kubernetes 1.37

Beta versus Alpha: the feature gate map

Table 1 below consolidates the gates, the components that need them and the API surface, drawing on the release post and the feature-gate reference. The release post names the components for each gate (kubernetes.io).

Capability	Stage in 1.37	Gate (default off)	Components named in the 1.37 post	API version
Workload and PodGroup APIs, gang scheduling, workload-aware preemption	Beta	GenericWorkload (kubernetes.io)	kube-apiserver, kube-controller-manager, kube-scheduler	scheduling.k8s.io/v1beta1 (kubernetes.io)
DRA (Dynamic Resource Allocation) ResourceClaims shared by a PodGroup	Beta (kubernetes.io)	DRAWorkloadResourceClaims	kube-apiserver, kube-controller-manager, kube-scheduler, kubelet	v1beta1 PodGroup field
Topology-aware workload scheduling	Alpha (kubernetes.io)	TopologyAwareWorkloadScheduling	kube-apiserver, kube-scheduler	PodGroup constraints
CompositePodGroup (gang of gangs)	Alpha (kubernetes.io)	CompositePodGroup (also needs the topology gate on the controller manager)	kube-apiserver, kube-controller-manager, kube-scheduler	scheduling.k8s.io/v1alpha3
Job controller integration (.spec.scheduling)	Alpha (kubernetes.io)	WorkloadWithJob	kube-apiserver, kube-controller-manager	Job batch/v1 field
PodGroup preemptionPolicy	Alpha	PodGroupPreemptionPolicy	kube-apiserver, kube-scheduler	PodGroup field

The table shows why enablement is a coordinated change: a Beta gate that defaults to false still needs the API group, matching flags on every named component, and restarts. The separate v1.36 gates no longer exist: GangScheduling was removed in 1.37 and merged into GenericWorkload, and the preemption gate was folded in because gang scheduling without workload-aware preemption was judged not valuable enough (www.kubernetes.dev). Disabling the gate on the scheduler therefore disables the preemption changes and gang scheduling together (www.kubernetes.dev).

API versions and the v1beta1 transition

The core Workload and PodGroup APIs were promoted to v1beta1, described as one step away from General Availability. For anyone who tested Alpha, the transition is breaking: v1alpha2 has been entirely replaced by v1alpha3, with changes around disruptionMode (kubernetes.io). The KEP is blunter about operations, telling operators to delete v1alpha2 resources before upgrading because they are unsupported in v1.37 (www.kubernetes.dev), and a roundup repeats the same warning (www.fairwinds.com). The disruptionMode values were also renamed during promotion so PodGroups and CompositePodGroups use consistent names.

The current core Workload and PodGroup API version in v1.37 is scheduling.k8s.io/v1beta1 ([Kubernetes v1.37 scheduling post](https://kubernetes.io/blog/2021/07/21/kubernetes-1.37-scheduling-post/)). KEP-4671 plans promotion to scheduling.k8s.io/v1 in v1.38; this is roadmap intent, not a shipped v1.37 API ([KEP-4671](https://github.com/kubernetes/enhancements/blob/master/keps/4671/WorkloadWithJob.md)). These describe successive stages, so keep manifests adaptable by templating apiVersion. Separately, Kubernetes deprecation policy says beta API versions are deprecated no more than 9 months or 3 minor releases after introduction (kubernetes.io), whereas alpha versions may be removed in any release without notice. That asymmetry is the practical reason to keep Alpha features (CompositePodGroup, topology) out of production manifests.

PodGroup as the queued and scheduled unit

Version 1.37 makes the PodGroup a first-class citizen of the scheduling queue: only the top-level PodGroup object is queued, where earlier releases queued every member Pod individually (kubernetes.io). The PodGroup stays out of the active queue until the PodGroup object exists and enough Pods have been created (kubernetes.io). The scheduler then takes a single snapshot of cluster state for the whole cycle. If it cannot place the minimum number of Pods, none are scheduled and the Pods wait in the unschedulable queue for cluster resources to free up, with standard backoff applying.

Three details matter for GPU acceptance tests:

- **Order dependence.** The default algorithm relies on Pod sorting and, for heterogeneous groups or groups with inter-Pod dependencies, may fail to find a valid placement (kubernetes.io); the v1.36 post made the same point.
- **One scheduler per gang.** All Pods of a group must share one scheduler name, so a gang cannot straddle kube-scheduler and a second scheduler (kubernetes.io).
- **Initial placement only.** The count of scheduled Pods can later drop below minCount if Pods are deleted or evicted (kubernetes.io), and the initial-scheduling condition is not updated when Pods later fail or are evicted. The minCount field became mutable in 1.37, letting controllers degrade or expand an elastic gang without interrupting scheduled Pods.

Workload-aware preemption

Preemption now treats the group as the unit. The documentation says preemption searches a cluster-wide domain instead of evaluating node by node (kubernetes.io), and victim ordering prefers PodGroups over individual Pods of equal priority. Two 1.37 changes close earlier gaps: default single-Pod preemption now respects a victim PodGroup's disruptionMode (kubernetes.io), a limitation the preemption documentation says no longer applies in v1.37. With disruptionMode: all, the scheduler is instructed that all Pods of the PodGroup have to be disrupted together (kubernetes.io). For composite hierarchies the mode defaults to single when unset.

Operators should note three limits:

- **Priority resolution.** A PodGroup whose priority class is not found is rejected.
- **Topology.** Topology-aware scheduling, as of v1.36 documentation, does not trigger workload or Pod preemption.
- **Plugins.** The preemption KEP records that the DynamicResources, NodeVolumeLimits and VolumeBinding plugins do not work well with preemption right now, and that workload-aware preemption does not improve this (www.kubernetes.dev): relevant for GPU gangs that use ResourceClaims.

DRA ResourceClaims for PodGroups

With DRAWorkloadResourceClaims, any number of Pods in a PodGroup can share a ResourceClaim reserved by the group instead of by each Pod (kubernetes.io). This lifts the practical limit created by recording individual Pods in status.reservedFor, which caps a claim at 256 Pods. For GPU fabrics this is the natural carrier for shared devices such as NVIDIA's ComputeDomain, which provisions ephemeral multi-node NVLink fabrics so

Pods on different nodes can share GPU memory ([dra-driver-nvidia-gpu.sigs.k8s.io](https://github.com/dra-driver-nvidia-gpu.sigs.k8s.io)), on Grace Blackwell hardware with Multi-Node NVLink ([dra-driver-nvidia-gpu.sigs.k8s.io](https://github.com/dra-driver-nvidia-gpu.sigs.k8s.io)); the fabric is torn down automatically when the workload finishes ([dra-driver-nvidia-gpu.sigs.k8s.io](https://github.com/dra-driver-nvidia-gpu.sigs.k8s.io)). Reservations persist in the claim until the PodGroup is deleted, even if the group has no Pods. And 1.37 changed disable behavior: when the gate is off and a claim was meant to be shared, no ResourceClaim is created at all, preventing a flood of per-Pod claims from exhausting DRA resources. The KEP adds a skew hazard: with an older kubelet, Pods referencing a PodGroup claim are scheduled, but the kubelet refuses to run them (www.kubernetes.dev).

Job controller integration and the controller building blocks

The native Job controller is the first consumer of the new controller-integration APIs and workloadbuilder library, which carry no feature gate of their own. The Job API gains an explicit `.spec.scheduling` field, and after creation it is immutable except that `gang.minCount` can change, which lets a running gang be resized. The Job controller is the only built-in controller that creates PodGroup objects from Workload templates (kubernetes.io), and an omitted `minCount` defaults to the Job's parallelism so all Pods must be schedulable together. Because the integration gate is Alpha, treat the Job path as a pilot feature.

What stays Alpha: topology and CompositePodGroup

The CompositePodGroup API expresses multi-level topology constraints, gang scheduling and preemption policies for heterogeneous groups, aimed at structures such as JobSet and LeaderWorkerSet. The scheduler commits a placement that satisfies the hierarchy's configured policies. A CompositePodGroup with basic policy admits child groups independently; with gang policy, at least `minGroupCount` child groups must be schedulable together ([Kubernetes scheduling policies](#)). Each leaf PodGroup must satisfy its own Pod-level policy. A whole-hierarchy, all-workers requirement therefore needs group and Pod quorums that require every member; atomic commitment alone does not require every descendant Pod ([Kubernetes gang scheduling documentation](#)). Topology-aware scheduling relies on node labels rather than discovering the network, per its KEP (www.kubernetes.dev), and its Beta criteria still require cluster autoscaling components to be aware of workload topology constraints. For private clusters that means rack and zone co-location must still be backed by labels you maintain and by tests you run.

Scope Boundaries: What Native Gang Scheduling Does Not Replace

The cleanest way to decide is to separate placement from policy. Native gang scheduling answers one question inside kube-scheduler: can this whole group be placed now? Kueue describes itself as a system that manages quotas and how jobs consume them (kueue.sigs.k8s.io), and states that pod-to-node scheduling remains the responsibility of kube-scheduler (kueue.sigs.k8s.io). The topology KEP likewise lists replacing external queueing and admission systems among its non-goals (www.kubernetes.dev).

Table 2 below compares responsibilities as documented by each project. It covers only claims found in the reviewed primary documentation; cells marked as not assessed are not asserted either way.

Responsibility	Native 1.37 (Beta, gate off)	Kueue	Volcano	KAI Scheduler
All-or-nothing placement	None of the Pods are bound if the group cannot fit (kubernetes.io)	Admits a workload only when quota fits; documents that aggregate quota can still leave an admitted job stuck at the scheduler layer (kueue.sigs.k8s.io)	A PodGroup with minMember; none of its pods scheduled if it cannot run (volcano.sh)	Ensures all pods in a group are scheduled simultaneously or not at all (github.com)
Queues, quota, fair share	Non-goal in the KEP (www.kubernetes.dev)	Quotas and fair sharing among tenants (kueue.sigs.k8s.io)	A Queue is a collection of PodGroups (volcano.sh)	Multi-level queue hierarchies (github.com)
Process model	In-tree kube-scheduler	Does not replace kube-scheduler (kueue.sigs.k8s.io)	Separate scheduler selected through the job's scheduler name (volcano.sh); other Pods are not gang scheduled (trainer.kubeflow.org)	Can run alongside other schedulers on the cluster (github.com); accepted into the CNCF (Cloud Native Computing Foundation) in December 2025 (www.cncf.io)
Topology	Alpha	Beta since Kueue v0.14 (kueue.sigs.k8s.io)	Not assessed in this report	Topology-aware scheduling of multi-level workloads (github.com)
GPU sharing	Not a scheduler feature; shared devices flow through DRA claims	Not assessed in this report	Not assessed in this report	Lets multiple workloads share single or multiple GPUs (github.com)
Preemption	Group-aware, cluster-wide	Cohort borrowing and preemption policies (kueue.sigs.k8s.io)	Preempt and reclaim actions in the scheduling cycle (volcano.sh)	Hierarchical queues with quotas, priorities and fairness policies

Table 2 supports a division of labor rather than a ranking. Kueue's own documentation concedes the subtlety: without topology awareness, quota-based admission can leave a job stuck indefinitely at the Kubernetes scheduler layer (kueue.sigs.k8s.io), and its topology-aware scheduling admits and reserves quota only if all Pods fit in the requested domains (kueue.sigs.k8s.io). Kueue's waitForPodsReady is described as a heuristic that does not prevent partial scheduling (kueue.sigs.k8s.io), and its block-admission option admits workloads one at a time (kueue.sigs.k8s.io). Native gangs address the placement half of that gap inside the scheduler; Kueue addresses the quota half above it.

The "Kueue vs native" question therefore usually resolves as "Kueue and native", with one rule: a single component should own each decision. The Kueue maintainers' own tracking issues show the boundary is still being negotiated:

- **Object matching.** One issue describes the goal that the Kueue Workload, PodGroup and Workload objects should match (github.com).
- **Unit mismatch.** Another records that gang units may not map one-to-one onto a native PodGroup, with an example where Kueue would admit nothing though native scheduling would place two of three groups (github.com).
- **Label conflicts.** A third notes that proprietary labels can lead kube-scheduler to reject Kueue admission decisions when they conflict with standard API specs (github.com).

Kueue also records a naming overlap, since the Kubernetes and Kueue Workload objects share the same kind name (github.com), so scripts and alerts must name the API group explicitly. A maintainer noted that maintaining both pod-group models long term is possible (github.com).

For Volcano and KAI the boundary is the scheduler process. Both are separate schedulers selected per Pod; Volcano describes itself as extending and enhancing the capabilities of the standard kube-scheduler (github.com). The scheduler-plugins project documents the general hazard of that arrangement: running multiple schedulers will inevitably encounter resource conflicts when the cluster is short of resources (scheduler-plugins.sigs.k8s.io), with the later Pod evicted by the kubelet and left hanging (scheduler-plugins.sigs.k8s.io). A cluster that keeps both kube-scheduler and a second scheduler placing onto the same GPU nodes should therefore partition nodes (labels, taints) or route every GPU workload through exactly one scheduler; the same project calls its single-scheduler mode conflict-free because a unified scheduler is used (scheduler-plugins.sigs.k8s.io). Because native gangs require one scheduler name per PodGroup, mixed routing within one gang is rejected rather than silently tolerated.

Controller Coexistence Map

Table 3 below maps who creates the native objects, who admits, and where two controllers could both act. Status reflects project documentation retrieved on 2026-10-20.

Workload type	Who creates native Workload and PodGroup	Maturity	Admission / queue owner today	Coordination point to test
Batch Job	Job controller from <code>spec.scheduling</code>	Alpha gate	Kueue via suspend, or none	In Kueue's mapping table only the Job row has a Kueue pull request (github.com); an externally supplied scheduling group stops the v1.36 Job controller from creating its own objects
JobSet	JobSet controller when <code>spec.scheduling</code> is set (jobset.sigs.k8s.io)	Alpha and off by default (jobset.sigs.k8s.io); needs Kubernetes 1.37+ with the gates (jobset.sigs.k8s.io)	Kueue (JobSet integration documented since JobSet v0.2.3) (jobset.sigs.k8s.io)	Composite parent/leaf hierarchies are not implemented in the alpha (jobset.sigs.k8s.io)

Workload type	Who creates native Workload and PodGroup	Maturity	Admission / queue owner today	Coordination point to test
LeaderWorkerSet	LWS controller when scheduling is set	Alpha, API may change (lws.sigs.k8s.io); opt-in (lws.sigs.k8s.io)	Kueue gates each LWS group with a scheduling gate until its Workload is admitted (kueue.sigs.k8s.io)	Cannot be combined with the exclusive-topology annotation (lws.sigs.k8s.io); the LWS KEP leaves Kueue integration to follow-up (github.com)
Ray (KubeRay)	Not documented as native in the reviewed Ray pages	Gang via Volcano from KubeRay 1.6.0 for RayJob (docs.ray.io), scheduler-plugins from v1.4.0 (docs.ray.io), KAI (docs.ray.io)	The chosen batch scheduler or Kueue	YuniKorn needs an explicit gang label, otherwise it schedules without gang enforcement (docs.ray.io)
Kubeflow Trainer (TrainJob)	PodGroupPolicy creates the appropriate PodGroup (trainer.kubeflow.org)	Policy backends listed in Trainer docs	Kueue, KAI, Volcano or Coscheduling	Pick one backend per runtime
Plain Pods with Kueue pod groups	Kueue labels	Stable in Kueue	Kueue removes its scheduling gate on admission (kueue.sigs.k8s.io)	Kueue's quota gate and the scheduler's gang gate form two-level admission that must be coordinated (github.com)

The interpretive point of Table 3 is that every row has exactly one party that can create group objects and one that can hold Pods back, and trouble appears when both are active on the same Pod. The pod-gating mechanism is concrete: Kueue patches the Pods and removes its scheduling gate once the Workload is admitted (kueue.sigs.k8s.io), and the native PodGroup then sits in PreEnqueue until the minimum number of Pods exist. If Kueue were to ungate only a subset of a group's Pods, the native scheduler would keep waiting for the rest; this is an inference from the documented mechanics, not a documented incident. The Kueue design discussion lists the matching requirement as two-level admission coordination, and notes that on clusters without GenericWorkload the scheduling group field is silently ignored, which would silently lose the gang guarantee (github.com). A reviewer on a related Kueue pull request observed that, from the scheduling perspective, a native Workload resource is optional (github.com).

Kueue is officially supported and tested on **Kubernetes 1.34 or newer** (kueue.sigs.k8s.io), so 1.37 is within range, and **Kueue v0.20.0** stopped serving its own v1beta1 API, requiring migration to v1beta2 before upgrade (github.com): two API migrations in one window should be sequenced, not combined. Kueue v0.20.0 also added an alpha unscheduled-timeout that evicts admitted workloads whose Pods remain unscheduled (github.com), a backstop that should be tested together with native gang waiting.

Implementation Considerations and Process Changes

Decision summary: enable, pilot or defer

The 1.37 post encourages trying workload-aware scheduling in test clusters. Applying that to private GPU estates yields three postures.

- **Enable (non-production first):** Clusters with no queueing layer, running Jobs or JobSet, where partial starts are the main pain. Use a canary cluster, since the Job and JobSet integrations are Alpha (jobset.sigs.k8s.io).
- **Pilot beside a policy layer:** Kueue, Volcano or KAI environments. Keep the policy layer authoritative for quota and admission, and let native gangs handle in-scheduler placement only for workloads that the policy layer does not already gang. Gate the pilot on the acceptance grid below.
- **Defer:** Managed or vendor distributions that do not expose the gates, clusters pinned to older minor versions, and anything that depends on topology or CompositePodGroup, which remain Alpha. Kueue integration is future work.

GPU acceptance test grid

Before relying on any claim, prove the behaviors. *Table 4* below lists scenarios with desired gang size, cluster state and the expected outcome from the documented semantics. The sizes are test parameters, not recommendations.

Scenario	Desired gang / cluster state	Expected outcome	Evidence to capture
Insufficient full gang	8 Pods × 8 GPUs; only 7 nodes have 8 free GPUs	No Pod bound; group waits unschedulable; smaller workloads still schedule	Pending Pods hold zero allocated devices; PodGroup condition false with reason recorded
Fragmented GPUs	8 × 8 GPUs; 64 free GPUs spread as 4 per node	Unschedulable; under Kueue alone the job could be admitted yet stuck	GPU-minutes held by partial placements stay at zero
Multi-node topology	8 Pods in one rack; free capacity split across racks	With the topology gate off, the constraint is not applied; with it on (Alpha), placement confined to a domain; no preemption to satisfy it	Node labels of bound Pods; note pre-gate Pods may span domains
Preemption of a running gang	High-priority gang needs capacity held by a low-priority gang with <code>disruptionMode: all</code>	Victim gang evicted as a whole, not one Pod	Eviction timestamps for every member within one window
Single-Pod preemptor	Default preemption of one Pod against a gang victim	Respects the PodGroup disruption mode in 1.37	Compare against a 1.36 control if available
Node loss mid-run	1 of 8 nodes fails after start	Other Pods are not unassigned by the scheduler (kubernetes.io); the initial condition is not rewritten; controller restart policy decides recovery	Time to full gang restart; GPUs idle while waiting for the replacement Pod

Scenario	Desired gang / cluster state	Expected outcome	Evidence to capture
Shared ResourceClaim	Mixed CPU, GPU and NIC claim shared by the group	One claim for the PodGroup; reservation persists until the PodGroup is deleted	Claim count equals expected; none leaked after job deletion
Kubelet skew	PodGroup claim with an older kubelet	Pods scheduled but refused at the kubelet	Kubelet versions in preflight
Scheduler routing	Pods in one gang with different scheduler names	Group rejected	Event message on the Pods
Rollback	Disable GenericWorkload on the scheduler	Pod-by-Pod behavior returns (www.kubernetes.dev); existing objects remain (www.kubernetes.dev)	Pending Pod count after rollback

The grid's value lies in the evidence column. It converts "gang scheduling works" into artifacts a reviewer can check: zero devices held by pending gangs, a whole-group eviction timestamp window, and a claim count. Run each row with the policy layer on and off, because the coexistence cases (a Kueue-gated Pod group, a LeaderWorkerSet group with a gate) are where the expected behavior is least documented.

Measuring stranded GPU-minutes

Do not assume savings; measure them. Define:

`stranded GPU-minutes = sum over gangs of (GPUs allocated to member Pods × minutes with no useful training or serving progress)`

Report this total in separate buckets: GPU-minutes held by incomplete placement, GPU-minutes waiting for container startup or readiness after placement, and GPU-minutes without useful application progress after startup. For each bucket, integrate the GPUs allocated to member Pods over the relevant interval; record binding, readiness and application-progress timestamps separately.

For an initially unplaced gang that cannot meet `minCount` and remains unbound, the incomplete-placement bucket should be zero ([Kubernetes gang scheduling documentation](#)). This does not promise zero time between the first binding and full startup: the kubelet creates containers after scheduling ([Kubernetes Pod lifecycle documentation](#)). Retain startup, readiness and useful-progress measurements even when initial placement succeeds. The total can also remain non-zero after node loss, because running Pods stay assigned. Track four companion measures:

- **Scheduling latency.** Time from PodGroup creation to the last Pod bound; the KEP warns Pod startup service level indicators may shift because gang Pods wait for all members to be created (www.kubernetes.dev).
- **Failed or retried admissions.** Count per queue and per workload type.
- **Recovery time.** Mean time to recover after a member failure.
- **Idle-but-unschedulable GPUs.** Free devices that cannot fit any pending gang.

(Hypothetical Example) A 16-node training gang with 8 GPUs per node places 12 Pods under a Pod-by-Pod scheduler and holds them for 25 minutes waiting for the rest. That is 96 GPUs × 25 minutes = **2,400 stranded GPU-minutes**, or 40 GPU-hours. At an illustrative list rate of \$12.29 per GPU-hour (derived below from a

public cloud price, not from any private cluster cost basis), the stranded cost is $40 \times \$12.29 = \text{\$491.60}$ for one event. The same event under an enforced gang binds zero Pods and records zero stranded GPU-minutes, though the 128 GPUs may sit idle: that idle capacity belongs in the idle-but-unschedulable measure, not in the stranded figure.

Rollback and API conversion plan

Keep every change reversible and sequenced.

1. **Upgrade order.** Control-plane components must not be newer than the API server and may be one minor version older. For the preemption changes, upgrade kube-apiserver before kube-scheduler (www.kubernetes.dev); on downgrade, downgrade kube-scheduler first (www.kubernetes.dev).
2. **Set new fields late.** For core scheduling, wait until all control-plane instances are upgraded before setting new fields. Shared DRA PodGroup ResourceClaims also require kubelet support: before workloads use shared claims, require compatible kubelets and enable DRAWorkloadResourceClaims on kube-apiserver, kube-controller-manager, kube-scheduler and the participating kubelets ([Kubernetes DRA API documentation](#); [KEP-5729 version skew strategy](#)).
3. **Delete Alpha objects.** Remove v1alpha2 Workload and PodGroup objects before upgrading.
4. **Enable in one change set.** Alpha API versions are disabled by default and must be explicitly enabled in the API server configuration, with restarts to follow.
5. **Rolling-upgrade fallback.** If the active scheduler has the feature disabled, behavior falls back to the status quo (www.kubernetes.dev): flip every scheduler replica together.
6. **Disable cleanly.** Disabling the gate leaves existing Workload and PodGroup objects in place (www.kubernetes.dev); for Jobs, with the gate off on the API server the field is cleared (www.kubernetes.dev).
7. **DRA downgrade.** The feature stops working on a release without support (www.kubernetes.dev), so tear down PodGroup claims first.
8. **Version-controlled manifests.** Template apiVersion and the scheduling blocks, since the KEP points to a promotion in v1.38 (www.kubernetes.dev).

Managed Kubernetes availability

As of 2026-10-20 the evidence for managed services is mostly negative: the reviewed pages do not document gang scheduling or the GenericWorkload gate. **Amazon EKS** lists 1.37 as released on **October 1, 2026** in its version table (docs.aws.amazon.com), but its documentation says alpha features are not supported (docs.aws.amazon.com) and that new beta APIs are not enabled in clusters by default (docs.aws.amazon.com). **Google GKE** announced 1.37 in the Rapid channel (cloud.google.com); the documented route to alpha gates is the alpha cluster, which enables all alpha APIs (cloud.google.com) but is automatically deleted after 30 days (cloud.google.com), so it suits tests only. The **AKS** calendar shows 1.37 GA in October 2026 (learn.microsoft.com), and its release notes record the preview rollout in September (github.com); no explicit GA statement was found. **Red Hat OpenShift** Container Platform 4.22 uses Kubernetes 1.35 (docs.redhat.com), and its technology-preview feature set cannot be undone and prevents minor version updates (docs.okd.io). The RKE2 releases page lists a 1.37 build, but no gate documentation

was reviewed (github.com). Upstream lists 1.37 as actively supported, with 1.37.1 the latest patch confirmed on the retrieval date. Private clusters on upstream kubeadm or similar distributions are the realistic early adopters, because they control component flags directly.

Data Analysis and Evidence

The upstream gang-policy performance-test configuration defines gangs with a minimum Pod count ([Kubernetes benchmark configuration](#)). These scheduler tests should not be treated as evidence of private-cluster GPU utilization or cost improvements. No independent end-to-end GPU savings benchmark was identified in the reviewed sources. The older trace studies below quantify the problem the feature targets; none measures native 1.37 gang scheduling.

- **Fragmentation delay (Microsoft, 2019).** A USENIX ATC paper analyzed a two-month trace from a multi-tenant GPU cluster. It reported average hardware utilization of allocated GPUs around **52%** (www.usenix.org), and fewer than 4.5% of servers completely empty when two thirds of GPUs were in use (www.usenix.org). Fragmentation delay accounted for 74.2% of queueing-delay occurrences in one job-size band (www.usenix.org) and around 80% of waiting time for multi-GPU jobs (www.usenix.org). The cluster used a different scheduler, so the numbers describe the problem, not the 1.37 feature.
- **Gang requirements at scale (Alibaba, 2022).** The NSDI paper characterized a production cluster with over 6,000 GPUs (www.usenix.org) and reported that **around 85%** of task instances carried gang-scheduling requirements (www.usenix.org).
- **Fragmentation after packing (Alibaba, 2023).** A later ATC paper warned that GPU sharing can leave hundreds of GPUs unable to be allocated (www.usenix.org), and in one trace-driven cluster reported about 500 GPUs unallocated despite apparent capacity (www.usenix.org). This is a fragmentation example, not a gang-scheduling one.
- **Ecosystem context (CNCF, January 2026).** The annual survey reports **82%** of container users running Kubernetes in production (www.cncf.io) and 66% of organizations hosting generative AI models using Kubernetes for some or all inference (www.cncf.io), from 628 respondents (www.linuxfoundation.org). The survey does not provide a gang-scheduling figure.

A reference price anchors the stranded-cost arithmetic. Azure describes its ND H100 v5 virtual machine as one VM with eight NVIDIA H100 Tensor Core GPUs (learn.microsoft.com), and its public retail price API returned **USD 98.32 per hour** for the East US pay-as-you-go Linux meter on 2026-10-20 (prices.azure.com). Dividing by eight gives **\$12.29 per GPU-hour**, or about \$0.205 per GPU-minute; an eight-GPU node held idle for 10 minutes costs about \$16.39. That is a cloud list rate used purely as a unit; a private cluster should substitute its own amortized cost per GPU-hour from procurement, power and facility records.

(Hypothetical Example) Capacity planning. A 160-GPU cluster has 20 nodes of 8 GPUs in four racks of five. A training team needs gangs of four whole nodes (32 GPUs) inside one rack. At a moment of steady load, rack A has 3 whole free nodes (24 GPUs), rack B has 2 (16), rack C has 1 whole node plus four nodes with 4 free GPUs each (24 free), and rack D has five nodes with 2 free GPUs each (10). Aggregate free GPUs are $24 + 16 + 24 + 10 = 74$, more than two 32-GPU gangs. The largest simultaneously schedulable gang within a rack

is only 24 GPUs (rack A's three whole nodes), so **zero** 32-GPU gangs can start. Aggregate GPU count says 74 free; schedulable-gang analysis says none. Repeating this calculation against real inventory snapshots, weekly, is a cheap diagnostic whether or not native gang scheduling is enabled.

Implications and Future Directions

Procurement. Size capacity by schedulable gangs and topology. The unit of purchase for distributed training is the largest gang you must run, expressed as whole nodes inside one failure and network domain, plus headroom for the long-running small jobs that fragment those nodes. Native gang scheduling makes this arithmetic more honest, because an unplaceable gang now stays pending instead of holding devices, but it does not create capacity. The Hypothetical Example above shows how 74 free GPUs can coexist with zero startable 32-GPU gangs; the corrective levers are node reservation policy, placement constraints, and, where justified, additional nodes in the right rack, not aggregate GPU count alone. Topology constraints in the core scheduler rely on node labels, so labeling quality becomes a procurement-adjacent control.

Roadmap. The working group post lists, for v1.38, GA graduation of the Workload and PodGroup APIs, Beta for topology-aware scheduling and CompositePodGroup, and closer Kueue alignment, with a long-term vision of Kueue using the native machinery for gang and topology placement. Kueue's tracked work already includes parsing native Workload, PodGroup and CompositePodGroup objects. Expect the coexistence guidance in this report to date quickly; recheck the Kueue release notes and the controller documentation before each minor upgrade.

Operational posture. Treat the feature as three independent decisions: scheduler placement (Beta), controller adoption (mostly Alpha) and policy-layer integration (in design), each with its own rollback. GPU Smith's advisory view is that the decision record should name the owner of each decision (quota, admission, placement, preemption, restart) and the metric that proves it.

Frequently Asked Questions (FAQs)

What is Kubernetes native gang scheduling? An all-or-nothing scheduling mode for Pods grouped in a PodGroup: if the cluster cannot accommodate the group, none of the Pods are bound, which keeps distributed training from holding devices while waiting for peers.

Is it enabled in Kubernetes 1.37 by default? No. Beta features here are disabled by default and require manual enablement.

Which feature gates are needed? GenericWorkload for the **Beta** core, plus DRAWorkloadResourceClaims for shared claims; Alpha features add their own gates (Table 1).

What are the Kubernetes 1.37 release features relevant to GPUs? Beyond gang scheduling, the release contains 67 enhancements in total; the workload-related ones are the Workload API, workload-aware preemption, DRA claims for PodGroups, CompositePodGroup and controller integration APIs.

Does the Workload API replace Kueue? No. Fairness and multiple queues are documented non-goals for kube-scheduler, and Kueue itself keeps pod-to-node scheduling with kube-scheduler.

Volcano vs kube-scheduler gang scheduling? Volcano is a separate scheduler selected by scheduler name (volcano.sh), with its own queues (volcano.sh); native gangs run inside kube-scheduler and need a single scheduler name per group. Do not let both place onto the same nodes without partitioning.

What is the best private GPU cluster scheduler for AI training? The evidence does not support a single answer. KAI Scheduler offers gang scheduling, hierarchical queues and GPU sharing; Kueue offers quota and fair sharing above kube-scheduler (kueue.sigs.k8s.io); native 1.37 offers atomic placement without policy. Choose by which decisions you need one component to own.

Conclusion

Kubernetes 1.37 gives private GPU operators a real, core-supported all-or-nothing placement primitive, with group-aware preemption, but it arrives disabled by default and surrounded by Alpha features. The official post calls out the gate, the API group move to v1beta1 and the status of topology, composite groups and Kueue alignment. The pragmatic adoption boundary is therefore this: pilot native gangs where no policy layer exists and the controllers can be tested; keep Kueue, Volcano or KAI authoritative for quota, queues and fairness; and defer production use of topology and composite hierarchies until they leave Alpha.

The deciding artifact is evidence, not preference. Run the acceptance grid, record stranded GPU-minutes, scheduling latency, failed admissions and recovery time, and keep gates and manifests reversible given a stable API step expected in v1.38. Capacity decisions should follow the largest simultaneously schedulable gang, since aggregate GPU count can look healthy while no gang can start. Re-check this analysis against the next release notes.

Source links

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

1. <https://kubernetes.io/blog/2026/08/26/kubernetes-v1-37-release>
2. <https://kubernetes.io/blog/2026/09/08/kubernetes-v1-37-advancing-workload-aware-scheduling>
3. <https://kubernetes.io/docs/concepts/scheduling-eviction/gang-scheduling>
4. <https://www.kubernetes.dev/resources/keps/4671>
5. <https://github.com/kubernetes-sigs/kueue/issues/14201>
6. https://jobset.sigs.k8s.io/docs/workload-aware-scheduling/declarative_scheduling
7. <https://docs.aws.amazon.com/eks/latest/userguide/kubernetes-versions.html>
8. <https://www.usenix.org/system/files/atc19-jeon.pdf>
9. <https://kubernetes.io/blog/2025/12/29/kubernetes-v1-35-introducing-workload-aware-scheduling>
10. <https://www.fairwinds.com/blog/kubernetes-1.37-is-here-pod-rightsizing-dra-and-gang-scheduling>
11. <https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates>
12. <https://kubernetes.io/docs/concepts/resource-management/dynamic-resource-allocation/dra-api>
13. <https://kubernetes.io/docs/concepts/workloads/compositepodgroup-api>
14. <https://kubernetes.io/docs/concepts/workloads/workload-api>

15. <https://www.kubernetes.dev/resources/keps/5710>
16. <https://kubernetes.io/docs/reference/using-api/deprecation-policy>
17. <https://kubernetes.io/docs/concepts/scheduling-eviction/podgroup-scheduling>
18. <https://kubernetes.io/docs/concepts/scheduling-eviction/workload-aware-preemption>
19. <https://kubernetes.io/docs/concepts/workloads/workload-api/disruption-and-priority>
20. <https://kubernetes.io/docs/concepts/workloads/podgroup-api>
21. <https://dra-driver-nvidia-gpu.sigs.k8s.io/docs>
22. <https://dra-driver-nvidia-gpu.sigs.k8s.io/docs/prerequisites>
23. <https://www.kubernetes.dev/resources/keps/5729>
24. <https://kubernetes.io/docs/concepts/workloads/workload-api/policies>
25. <https://www.kubernetes.dev/resources/keps/5732>
26. <https://kueue.sigs.k8s.io/docs/overview>
27. https://kueue.sigs.k8s.io/docs/concepts/all_or_nothing
28. <https://volcano.sh/en/docs/podgroup>
29. <https://github.com/NVIDIA/KAI-Scheduler>
30. <https://volcano.sh/en/docs/queue>
31. <https://volcano.sh/en/docs/vcjob>
32. <https://trainer.kubeflow.org/en/latest/operator-guides/job-scheduling/volcano.html>
33. <https://www.cncf.io/projects/kai-scheduler>
34. https://kueue.sigs.k8s.io/docs/concepts/topology_aware_scheduling
35. <https://kueue.sigs.k8s.io/docs/concepts/preemption>
36. https://volcano.sh/en/docs/scheduler_introduction
37. <https://github.com/kubernetes-sigs/kueue/issues/13707>
38. <https://github.com/kubernetes-sigs/kueue/issues/13150>
39. <https://github.com/kubernetes-sigs/kueue/issues/12385>
40. <https://github.com/volcano-sh/volcano>
41. <https://scheduler-plugins.sigs.k8s.io/docs/user-guide/installation>
42. <https://jobset.sigs.k8s.io/docs/workload-aware-scheduling>
43. <https://jobset.sigs.k8s.io/docs/overview>
44. <https://lws.sigs.k8s.io/docs/overview>
45. <https://lws.sigs.k8s.io/docs/concepts/leaderworkerset/gang-scheduling>
46. <https://kueue.sigs.k8s.io/docs/tasks/run/leaderworkerset>
47. <https://github.com/kubernetes-sigs/lws/tree/main/keps/666-gang-scheduling-in-lws>
48. <https://docs.ray.io/en/latest/cluster/kubernetes/k8s-ecosystem/volcano.html>
49. <https://docs.ray.io/en/latest/cluster/kubernetes/k8s-ecosystem/scheduler-plugins.html>
50. <https://docs.ray.io/en/latest/cluster/kubernetes/k8s-ecosystem/kai-scheduler.html>
51. <https://docs.ray.io/en/latest/cluster/kubernetes/k8s-ecosystem/yunikorn.html>

52. <https://trainer.kubeflow.org/en/latest/operator-guides/job-scheduling/overview.html>
53. https://kueue.sigs.k8s.io/docs/tasks/run/plain_pods
54. <https://github.com/kubernetes-sigs/kueue/pull/13643>
55. <https://kueue.sigs.k8s.io/docs/installation>
56. <https://github.com/kubernetes-sigs/kueue/releases/tag/v0.20.0>
57. <https://kubernetes.io/blog/2026/05/13/kubernetes-v1-36-advancing-workload-aware-scheduling>
58. <https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle>
59. <https://www.kubernetes.dev/resources/keps/5547>
60. <https://cloud.google.com/kubernetes-engine/docs/release-notes>
61. <https://cloud.google.com/kubernetes-engine/docs/concepts/alpha-clusters>
62. <https://learn.microsoft.com/en-us/azure/aks/supported-kubernetes-versions>
63. <https://github.com/Azure/AKS/releases>
64. https://docs.redhat.com/en/documentation/openshift_container_platform/4.22/html-single/release_notes/index
65. <https://docs.okd.io/4.22/nodes/clusters/nodes-cluster-enabling-features.html>
66. <https://github.com/rancher/rke2/releases>
67. https://github.com/kubernetes/kubernetes/blob/a6d4708e816967a7d0c64155d6a702ab19463607/test/integration/scheduler_perf/podgroup/gangscheduling/performance-config.yaml
68. <https://www.usenix.org/system/files/nsdi22-paper-weng.pdf>
69. <https://www.usenix.org/system/files/atc23-weng.pdf>
70. <https://www.cncf.io/announcements/2026/01/20/kubernetes-established-as-the-de-facto-operating-system-for-ai-as-production-use-hits-82-in-2025-cncf-annual-cloud-native-survey>
71. https://www.linuxfoundation.org/hubfs/Research%20Reports/CNCF_Annual_Survey_Report_1.15.26.pdf
72. <https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/gpu-accelerated/ndh100v5-series>
73. [https://prices.azure.com/api/retail/prices?
%24filter=armSkuName%20eq%20%27Standard_ND96isr_H100_v5%27%20and%20armRegionName%20eq%20%27eastus%27%20and%20priceType%20eq%20%27Consumption%27](https://prices.azure.com/api/retail/prices?%24filter=armSkuName%20eq%20%27Standard_ND96isr_H100_v5%27%20and%20armRegionName%20eq%20%27eastus%27%20and%20priceType%20eq%20%27Consumption%27)

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.