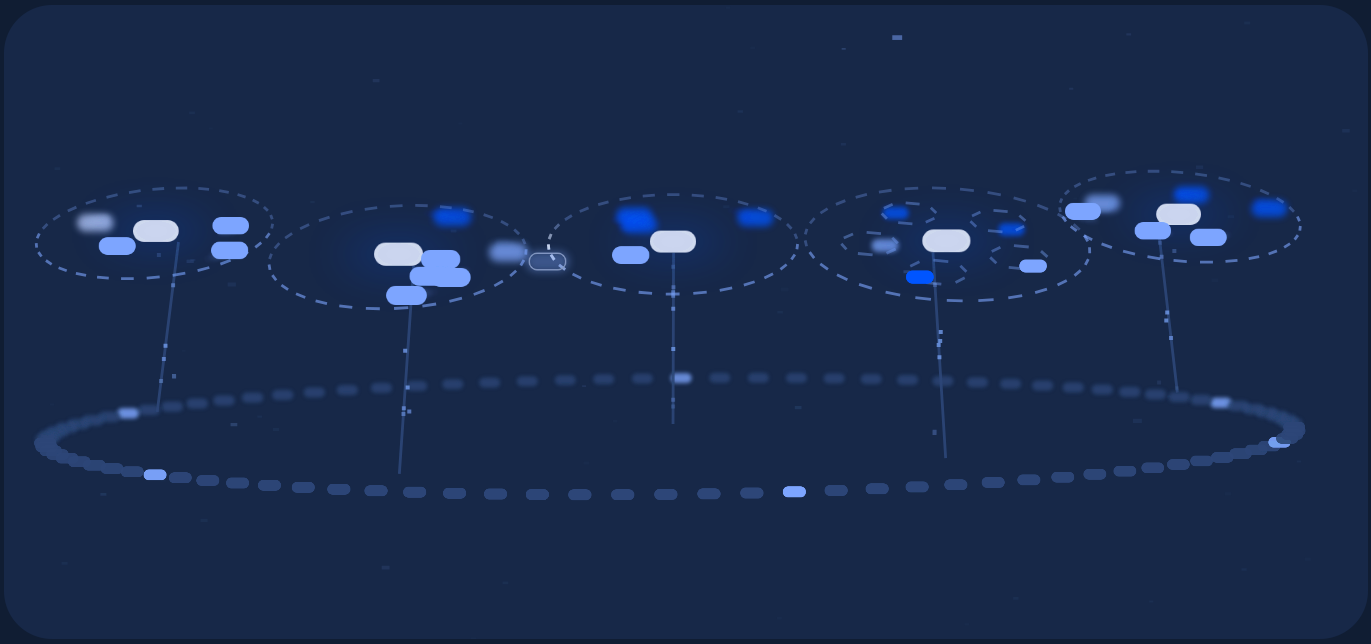
Rillor

EXPLAINER

GPU scheduling on Kubernetes: operators, sharing and node pools

Kubernetes GPU scheduling, layer by layer: device plugins and whole-GPU requests, the GPU Operator, node pools built from labels and taints, DRA claims, sharing and driver upgrades.



Rillor

Contents

01 Summary

02 Background

03 Method

04 Findings

05 Implications

06 Limits of this analysis

07 Sources

Summary

GPU scheduling on Kubernetes runs through a stack of parts: a vendor driver and a container toolkit on each node, a component that advertises the GPUs to Kubernetes, and the scheduler that places each Pod on a node that has them [1, 2, 3, 39].

- A device plugin advertises GPUs to Kubernetes as an extended resource, such as `nvidia.com/gpu`. Extended resources are whole numbers that cannot be overcommitted, and a device cannot be shared between containers, so each Pod receives whole GPUs [1, 2].
- NVIDIA's GPU Operator installs and manages the driver, the device plugin, the container toolkit, GPU node labels and monitoring on each GPU node. Release 26.7.2, published 14 October 2026, was its current release on 8 November 2026 [14, 15, 44].
- Kubernetes has no node pool object. A GPU node pool is a group of nodes that share labels and a taint, so GPU Pods select those nodes and other Pods stay off them. Cloud providers and the Karpenter autoscaler name and manage such groups (derived from [9, 10, 29, 42]).
- Dynamic Resource Allocation (DRA) lets a Pod claim devices by attributes such as GPU model and memory. Its core graduated in v1.34, and DRA support for extended resources graduated in v1.37, so Pods that request a GPU count can be served by DRA [3, 4, 5].
- The Kubernetes scheduler does not preempt DRA resources. A higher-priority Pod that needs a DRA device waits until that device is free [3, 18].
- NVIDIA's DRA driver allocates full GPUs and Multi-Instance GPU (MIG) slices, and builds ComputeDomains, multi-node NVLink fabrics that last only as long as the workload. For driver v0.5.1, NVIDIA lists full GPU and existing MIG allocation and ComputeDomains as generally available, and its sharing feature gates (ConsumableShares, MPSSupport and TimeSlicingSettings) as off by default [18, 19].
- A GPU driver upgrade unloads and reloads kernel modules, so the GPU's clients stop first. The GPU Operator's upgrade controller is on by default and, without a policy, upgrades one node at a time with at most 25% of nodes unavailable [17].
- Allocation is not use. In a 2019 study of one multi-tenant GPU cluster that ran on Apache YARN, GPUs in use averaged about 52% utilization over two months, and in a 2022 study of another cluster with over 6,000 GPUs, the median instance used 0.042 GPUs' worth of compute against a median request of 0.5 GPUs [33, 34].

Background

From a GPU to a container

Kubernetes reaches hardware that needs vendor-specific setup, such as GPUs, network cards and FPGAs, through device plugins, which vendors write without changing Kubernetes code and which you deploy by hand or as a DaemonSet [1]. Its guide to scheduling GPUs gives the order of work: install the GPU vendor's driver on the nodes, then run the vendor's device plugin, after which the cluster exposes a schedulable resource such as `amd.com/gpu` or `nvidia.com/gpu` [2]. The guide links plugins from AMD, Intel and NVIDIA, and lists support for managing AMD and NVIDIA GPUs through device plugins as stable since v1.26 [2].

A container toolkit sits between the driver and the container. NVIDIA describes its toolkit as libraries and utilities for building and running GPU-accelerated containers, among them a container runtime, the `nvidia-ctk` command-line tool, hooks for the Container Device Interface (CDI) and the `libnvidia-container` library [39]. CDI is an open specification that defines what access to a device means and standardizes it across container runtimes. Since GPU Operator v25.10.0, CDI is the default way GPUs reach containers on Kubernetes, through CDI support in containerd and CRI-O [23].

The scheduler filters out the nodes that cannot run a Pod, scores the feasible ones and picks the node with the top score, breaking ties at random. If no node fits, the Pod stays unscheduled until the scheduler can place it [11]. It works from each node's allocatable resources, which the kubelet keeps up to date [7].

Why GPUs are scheduled differently

A container can request a fraction of a CPU core [7]. Through a device plugin, a GPU request is a whole number: extended resources are integers, they cannot be overcommitted, and a device cannot be shared between containers [1]. Research on large GPU clusters has measured how much of each allocated GPU is used, and how many GPUs fragmentation leaves unallocated [33, 34, 35].

The CNCF conformance checklist for Kubernetes v1.37 requires (MUST) that accelerator access be isolated and mediated by Kubernetes resource management, through a device plugin or DRA, and that a cluster autoscaler, where the platform has one, can scale node groups of a given accelerator type up and down based on Pods waiting for that accelerator [32]. Kubernetes v1.37 was released on 26 August 2026 with 67 enhancements, 16 of them graduating to stable [6].

This report is for platform engineers, machine learning engineers and buyers of GPU clusters who need to know how a Pod gets a GPU on Kubernetes and which parts decide where it runs.

Method

What was examined. How a GPU reaches a container on Kubernetes: device plugins, the NVIDIA GPU Operator, node labels, taints and tolerations, quotas and queues, scheduler scoring and NUMA alignment, Dynamic Resource Allocation and NVIDIA's DRA driver, GPU sharing in brief, driver upgrades and the security of the node software.

Sources. Kubernetes documentation, blog posts and changelogs, NVIDIA documentation and repositories, the DRA driver project's documentation, two National Vulnerability Database records, Google Cloud, Microsoft, Karpenter, Kueue and KAI Scheduler documentation, the CNCF conformance checklist for v1.37 [32] and three USENIX papers [33, 34, 35].

How. Every page was opened on 8 November 2026 ([31] and [37] again and [45] on 13 November 2026), and each claim comes from the source's own text. Maturity is given as each project states it: the version in which a feature graduated, or whether it is on or off by default. Where the DRA driver project and NVIDIA's documentation describe the driver in different terms, this report gives each statement as its publisher's own. The decision table in Finding 14 and every statement marked "derived" combine several sources. No cluster was built and no benchmark was run for this report.

Findings

1. The device plugin turns GPUs into a count

A device plugin registers with the kubelet over a gRPC service on a Unix socket and names its resource as vendor-domain/resourcetype, such as `nvidia.com/gpu`. The kubelet then advertises the count to the API server in its node status updates [1]. When a container is assigned devices, the plugin's Allocate call returns the container runtime settings that give it access to them [1].

GPUs go in the `limits` section of a container spec. A container can set limits alone, which Kubernetes then uses as the request, or limits and requests at the same value, never requests alone [2]. Device plugins can report the NUMA node of each device, so that the kubelet's Topology Manager can align devices with CPUs, an integration stable since v1.27 [1].

NVIDIA's device plugin runs as a DaemonSet that exposes the number of GPUs on each node, keeps track of their health and runs GPU-enabled containers. Its README notes that it lacks comprehensive GPU health checking and GPU cleanup features [20]. When a plugin marks a device unhealthy, the kubelet lowers the node's allocatable count for that resource while the capacity count stays the same, and Pods already on the failed device typically start to fail [1].

2. What a device plugin request cannot say

Kubernetes' DRA page lists the limits of device plugins: they need per-container device requests, do not support device sharing and do not support filtering devices by expression [3]. The DRA driver project describes the same model in its own terms. Hardware becomes opaque countable integers, a workload gets a whole unit or nothing, and there is no way to express sharing, per-workload configuration, capability constraints or topology requirements [19]. That project presents DRA as a replacement for the device plugin architecture, while the Kubernetes documentation sets out DRA's benefits next to device plugins [3, 19].

3. The GPU Operator installs the node software

NVIDIA writes that configuring GPU nodes means configuring several components, such as drivers, container runtimes and libraries, which is difficult and prone to errors [14]. Its GPU Operator uses the Kubernetes operator framework to manage them: the driver, the device plugin, the container toolkit, automatic node labels from GPU Feature Discovery (GFD) and monitoring based on DCGM, among others [14].

Release 26.7.0 shipped driver 595.91.07, Container Toolkit v1.20.0, device plugin v0.20.0 and DRA driver v0.5.0 [15]. Release 26.7.2, published on 14 October 2026 [44], moved the DRA driver to v0.5.1 [15]. NVIDIA's platform table lists Kubernetes 1.33 to 1.37 for release 26.7 [38].

The Operator deploys Node Feature Discovery (NFD) by default and finds GPU worker nodes by the label `feature.node.kubernetes.io/pci-10de.present=true`, 0x10de being NVIDIA's PCI vendor ID [16]. Labels such as `nvidia.com/gpu.deploy.operands=false` keep its components off a node [16]. A ClusterPolicy runs the device plugin stack, and a GPUCluster resource, added in 26.7.0, runs the DRA driver stack [15, 18]. A cluster can have one or the other, never both [15, 18].

GPU Operator vs device plugin. The device plugin is one component the Operator installs and configures, next to the driver, the toolkit, GFD, DCGM monitoring, MIG Manager and a validator (derived from [14, 15]). The plugin also runs alone, as a DaemonSet or a Helm chart, on nodes that already have the NVIDIA driver, the container toolkit and the NVIDIA runtime configured [20]. Managed clouds split the same pieces their own way. On GKE, driver installation is an option of the GPU node pool, set to `default`, `latest` or `disabled` [27]. On AKS, Microsoft recommends GPU node pools in which AKS installs and maintains the NVIDIA driver, the device plugin and the DCGM metrics exporter [28].

4. Labels tell GPU types apart

Kubernetes' GPU guide says that when nodes hold different GPU types, node labels and node selectors send Pods to the right nodes, and that its `accelerator` label key is only an example [2]. NFD detects each node's hardware features and typically advertises them as node labels, and a vendor plugin for NFD adds GPU labels [2].

GFD, kept in NVIDIA's device plugin repository, generates labels such as `nvidia.com/gpu.product` for the model, `nvidia.com/gpu.memory` in MiB, `nvidia.com/gpu.count` and labels for the driver and CUDA versions [43]. When a sharing strategy is on and `renameByDefault` is false, the product label gains a `-SHARED` suffix, so a node selector can attract Pods to nodes with shared GPUs [20, 43].

Node affinity adds required and preferred rules to the basic nodeSelector, and rules marked `IgnoredDuringExecution` leave a running Pod in place if the node's labels change later [10]. For isolation, the documentation advises label keys that the kubelet cannot modify, since a compromised node could otherwise set them on itself [10].

5. A node pool is labels plus taints

Core Kubernetes has no node pool object. Its documentation places work with labels, affinity and taints, and the pool is a name that cloud providers and autoscalers give a group of nodes (derived from [9, 10, 29, 42]). Google defines a GKE node pool as a group of nodes within a cluster that all have the same configuration, and labels each node with `cloud.google.com/gke-nodepool` and the pool's name [42]. Karpenter's NodePool sets constraints on the nodes Karpenter can create and on the Pods that can run on them, taints included [29].

Taints and tolerations do the fencing. A NoSchedule taint keeps new Pods without a matching toleration off a node, PreferNoSchedule is a soft version, and NoExecute also evicts running Pods that do not tolerate it [9]. For nodes with special hardware, the documentation's recommended form represents the hardware as an extended resource, taints the nodes with the extended resource name and runs the ExtendedResourceToleration admission controller, which adds the toleration to any Pod that requests the resource [9]. That admission controller is disabled by default [41]. A toleration lets a Pod onto a tainted node and does not keep it there, so dedicated nodes also carry a label that matches the taint, which Pods require with node affinity [9].

Cloud providers apply the pattern in their own ways. GKE adds the taint `nvidia.com/gpu=present:NoSchedule` when a GPU node pool joins a cluster that already has a non-GPU pool, and runs ExtendedResourceToleration so only Pods requesting GPUs are scheduled on GPU nodes, which it says lets GPU nodes scale down quickly [27]. Karpenter's example for isolating expensive hardware sets an `nvidia.com/gpu` taint with NoSchedule, and a NodePool can cap `nvidia.com/gpu`, though Karpenter warns that its limit checks are eventually consistent and can overrun during rapid scale-outs [29].

6. Quotas and queues divide GPUs between teams

A namespace quota caps GPU requests. Kubernetes' example limits the GPUs requested in a namespace to 4 with `requests.nvidia.com/gpu: 4`, and only `requests.` entries are allowed for extended resources, because they cannot be overcommitted [8]. DRA claims count against per-DeviceClass quotas, and when DRA serves extended resources, the extended resource quota also counts the devices provided by the device plugin [8].

Kueue maps quota to node types with ResourceFlavors, which tie resource variants to nodes through labels, taints and tolerations. When Kueue admits a workload, it adds the flavor's node labels to the Pod's nodeSelector and the flavor's tolerations to the Pod template [30]. Its documentation warns that with two GPU types under one `nvidia.com/gpu` resource, Pods may land on either type unless flavors carry node labels that tell them apart [30].

The default scheduler places Pods one at a time, so part of a job can start while the rest waits, and the v1.37 release announcement notes that such partial scheduling can lead to deadlocks and inefficient use of cluster resources [6]. The same announcement describes how v1.37 extends gang scheduling through the Workload API and PodGroup [6]. The CNCF checklist for v1.37 requires (MUST) that a platform can install and run at least one gang scheduling solution that schedules all or nothing, naming Kueue and Volcano as examples [32].

GPU-focused secondary schedulers also exist. KAI Scheduler, under the Apache 2.0 license, offers all-or-nothing batch scheduling, bin packing, hierarchical queues with quotas, DRA support and GPU sharing, and it can run alongside other schedulers on the cluster [31]. How Kubernetes queues and gang placement compare with Slurm is the subject of Slurm or Kubernetes for GPU clusters: how to choose (https://rillor.com/insights/slurm-or-kubernetes-for-gpu-clusters).

7. Placement: packing, fragmentation and NUMA

The scheduler's NodeResourcesFit plugin has two scoring strategies that pack Pods together. MostAllocated favors nodes with higher allocation and takes a weight per resource, and the documentation's example weights two extended resources at 3 against 1 for CPU and memory [12]. RequestedToCapacityRatio lets operators bin pack extended resources to improve the use of scarce resources in large clusters [12]. Packing helps prepare the least used nodes for scale-down [12].

Three studies show how far use can fall behind allocation. One cluster ran on Apache YARN, one on the Fuxi scheduling system, and the third paper's production clusters run on a customized Kubernetes system, with its scheduler tested on an emulated cluster [33, 34, 35]:

- In a two-month trace of one company's multi-tenant GPU cluster, published in 2019, GPUs in use ran at only around 52% utilization on average, and about 30% of jobs were killed or finished unsuccessfully [33]. That cluster ran on Apache YARN, not Kubernetes [33].
- In a two-month trace of a second company's machine learning cluster with 6,742 GPUs, published in 2022, the median instance used 0.042 GPUs' worth of streaming multiprocessors against a median request of 0.5 GPUs [34].
- A 2023 paper on production traces from that second company, whose tasks run on a customized Kubernetes system, found that allocating partial GPUs can leave hundreds of GPUs unable to be allocated in large clusters [35]. Its Fragmentation Gradient Descent scheduler, built as a new scheduler in Kubernetes and tested on an emulated cluster of more than 6,200 GPUs, cut unallocated GPUs by up to 49% [35].

NUMA alignment happens on the node, in the kubelet. Before the Topology Manager, the CPU and device managers made allocation decisions independently, which could place CPUs and devices on different NUMA nodes and add latency [13]. The Topology Manager, stable since v1.27, aligns them under the `best-effort`, `restricted` or `single-numa-node` policy [13]. Under the last two, a Pod that cannot be aligned is rejected at admission, the scheduler does not try to place it again, and the documentation recommends a Deployment or ReplicaSet, or an external control loop, to replace it [13]. The scheduler does not see NUMA alignment when it picks a node (derived from [13]).

8. DRA claims GPUs by attribute

DRA lets Pods request and share resources. Device drivers and administrators define device classes that workloads can claim, and Kubernetes allocates matching devices to each claim and places the Pods on nodes that can reach the allocated devices [3].

Drivers publish ResourceSlices that describe their devices, configure the devices and attach them to containers through CDI. Administrators install drivers and may create DeviceClasses, and workload operators create ResourceClaims or ResourceClaimTemplates [3]. A ResourceClaimTemplate gives every Pod its own claim, and a shared ResourceClaim lets several Pods use the same devices. The task guide's example

selects devices whose type attribute is "gpu" and whose memory capacity is 64Gi [40]. The DRA page also lists filtering on device attributes with CEL expressions, vendor configuration per claim in place of per node, and device classes by use case, such as a cost-optimized class and a high-performance class [3].

The core of DRA graduated to general availability in v1.34, on by default, with the stable v1 API as the default version [4]. The v1.35 changelog records that the feature gate for the core functionality is locked to enabled and can no longer be disabled [36], and the documentation lists DRA as stable since v1.35 [3]. One limit matters for GPU fleets: the scheduler does not support preemption for DRA resources, so a higher-priority Pod that needs DRA devices waits until a device frees up [3, 18].

9. What graduated in v1.37

The DRA features that graduated in v1.37 include these [5, 6]:

- **Extended resources through DRA.** A DRA driver can satisfy requests for an extended resource such as `example.com/gpu` without a separate device plugin. A DeviceClass can carry an extended resource name, and workloads written against extended resources keep working unmodified while the allocation moves to DRA [5].
- **Device taints and tolerations.** Drivers can taint devices so new Pods skip them, administrators can apply a DeviceTaintRule across the cluster, and Pods on a tainted device can be evicted unless their claim tolerates the taint, so an operator can take a single device offline for maintenance [5].
- **Device data in ResourceClaim status,** such as a network interface name, a MAC address and IP addresses [5, 6].

A standard NUMA attribute, `resource.kubernetes.io/numaNode`, was added in v1.37 with no feature gate and lets devices from different drivers be compared by NUMA node [5, 6].

The newest 1.37 patch on 13 November 2026 was 1.37.2, released on 28 October 2026 [45], and the Kubernetes releases page gives 28 October 2027 as the end of life for 1.37 [37].

10. NVIDIA's DRA driver and ComputeDomains

NVIDIA's DRA driver manages two kinds of resource. GPUs (`gpu.nvidia.com`) cover full GPUs, MIG slices and VFIO passthrough devices. ComputeDomains (`compute-domain.nvidia.com`) set up multi-node NVLink fabrics using IMEX, so that Pods on different nodes can share GPU memory at full NVLink bandwidth [19]. The fabric is torn down automatically when the workload finishes, and the driver needs Kubernetes 1.32 or later [19].

NVIDIA's maturity table for driver v0.5.1 lists ComputeDomains, full GPU and existing MIG device allocation, and three ComputeDomain-related feature gates as generally available and enabled [18]. ConsumableShares, DeviceMetadata, DynamicMIG, MPSSupport, NVMLDeviceHealthCheck, PassthroughSupport and TimeSlicingSettings are off by default [18]. Under DRA, a workload selects a GPU with CEL over attributes

such as model, architecture, memory, compute capability and PCI topology. The device plugin offers a resource name and a count, with node labels for any structured selection [18]. The GPUCluster workflow in GPU Operator 26.7.0 needs Kubernetes v1.34.2 or later, NVIDIA driver 580 or later and a CDI-compatible container runtime [15].

Two details matter when moving from the device plugin. The Operator's managed `gpu.nvidia.com` DeviceClass does not set an extended resource name of `nvidia.com/gpu` [18], so existing `nvidia.com/gpu` requests do not move to DRA by themselves under that class (derived from [5, 18], and not tested for this report). Health reporting differs as well. NVML device health checks are off by default in v0.5.1, while the device plugin reports unhealthy devices through the Device Plugin API [18].

11. Sharing in brief

Time-slicing defines replicas of a GPU and hands each to a Pod. NVIDIA's documentation says there is no memory or fault isolation between replicas, unlike MIG, and that a request for more than one replica does not guarantee a proportional share of compute [22]. MIG partitions a GPU in hardware. The single strategy exposes each MIG device as `nvidia.com/gpu` and needs every GPU on the node in MIG mode with one profile, and the mixed strategy exposes each profile under its own name, such as `nvidia.com/mig-1g.10gb` [21].

Under DRA, sharing across independent claims is off by default in driver v0.5.1 [18]. The CNCF checklist asks (SHOULD) for at least one GPU sharing strategy and for hardware partitions exposed as schedulable resources [32]. The methods, their measurements and how to size a slice are covered in MIG, MPS or time-slicing: sharing GPUs on Kubernetes (https://rillor.com/insights/mig-mps-or-time-slicing).

12. Driver upgrades on a live cluster

A GPU driver's kernel modules must be unloaded and loaded again on each driver container restart. NVIDIA's sequence is to disable the driver's clients, unload the modules, start the new driver Pod, load the new modules and re-enable the clients [17]. The GPU Operator manages only containerized drivers, and leaves drivers preinstalled on the host alone [17].

The upgrade controller is on by default and recommended by NVIDIA. An upgrade is a change to the driver version field of the NVIDIADriver or ClusterPolicy resource, and each node's `nvidia.com/gpu-driver-upgrade-state` label moves to `upgrade-done` when the node is finished [17]. Without a policy, upgrades are automatic with `maxParallelUpgrades: 1` and `maxUnavailable: 25%` [17]. Drain is off by default and serves as a fallback when deleting GPU Pods does not remove the workloads [17]. A policy per NVIDIADriver resource can set these per node pool [17].

Since 26.7.0, a patch upgrade with an unchanged driver configuration cordons the node and restarts the driver Pod in place, without evicting workloads or draining the node [15]. Under DRA, Pods with an allocated `gpu.nvidia.com` claim count as GPU workloads during an upgrade [17]. On the Kubernetes side, the Device

Plugins page asks plugin developers to support several plugin API versions and to upgrade plugins before nodes [1].

13. Security: privileged components, CDI and the toolkit

The node software runs with high privilege. Where Pod Security Admission restricts Pods, the Operator's namespace is labelled `pod-security.kubernetes.io/enforce=privileged` [16]. The driver and toolkit containers need `privileged: true`, `hostPID: true` and `hostIPC: true` to reach host devices, restart services such as containerd and load kernel modules. NVIDIA writes that only the cluster administrator needs access to the Operator's namespace [24].

The container toolkit has carried flaws that NVIDIA scored as critical. CVE-2024-0132, published 26 September 2024, is a time-of-check time-of-use flaw in toolkit 1.16.1 and earlier, in the default configuration, through which a crafted container image may gain access to the host file system [25]. Its record says it does not affect use cases where CDI is used, and lists scores of 9.0 from NVIDIA and 8.3 from NIST [25]. CVE-2025-23266, published 17 July 2025, sits in hooks used to initialize the container and lets an attacker execute arbitrary code with elevated permissions, scored 9.0 by NVIDIA [26].

NVIDIA's security page for the Operator lists seven toolkit CVEs, from CVE-2024-0132 to CVE-2025-23359, each with the toolkit and Operator release that fixed it [24]. CVE-2024-0132 was fixed in toolkit 1.16.2 and Operator 24.6.2 [24]. CDI has been the default injection path since Operator v25.10.0 [23], and it is the path DRA drivers use to attach devices to containers [3]. The history of these container escapes is set out in Security risks in self-hosted LLM deployments (https://rillor.com/insights/self-hosted-llm-security-risks).

14. A decision table

Situation	What the sources point to	Sources
One GPU model, whole GPUs per Pod	Device plugin with `nvidia.com/gpu` limits, GPU nodes tainted, tolerations added by ExtendedResourceToleration or written in the Pod	[1, 2, 9]
Several GPU models	Node labels from NFD or GFD with nodeSelector or node affinity, or DRA DeviceClasses with CEL on model and memory	[2, 3, 18, 43]
Many teams on one cluster	Per-namespace `requests.nvidia.com/gpu` quotas, or Kueue flavors and queues	[8, 30]
Pods that need less than a GPU	Time-slicing replicas without memory or fault isolation, or MIG profiles	[21, 22]

Situation	What the sources point to	Sources
Multi-node NVLink jobs	DRA ComputeDomains, with the DRA driver, Kubernetes v1.34.2 or later, driver 580 or later and a CDI runtime	[15, 18, 19]
Gang-scheduled training	The Workload API and PodGroup, or a gang scheduler or queue such as Kueue, Volcano or a GPU-focused scheduler	[6, 31, 32]
Higher-priority GPU jobs that must preempt others	Device plugin resources, since the scheduler does not preempt DRA resources	[3, 18]
Packing for scale-down	MostAllocated or RequestedToCapacityRatio with a weight on the GPU resource	[12]

Table 1. Derived from the sources in each row. No row was tested for this report.

Implications

For platform teams. Choose between the device plugin stack and the DRA stack early, since the GPU Operator runs one or the other in a cluster [18]. Fence GPU nodes with a taint and a matching label, because a toleration alone lets Pods in without keeping them there [9], and use label keys the kubelet cannot change for any label that isolates workloads [10]. Where high-priority jobs must displace others, account for the scheduler's lack of preemption for DRA resources [3].

For teams that share a cluster. A quota on `requests.nvidia.com/gpu` caps a namespace [8], and Kueue's flavors keep GPU types apart in its queues [30]. Jobs that need all their Pods at once need all-or-nothing placement, which v1.37 extends through the Workload API and PodGroup, or a gang scheduler such as Kueue or Volcano [6, 32].

For those who run the node software. NVIDIA's controller upgrades one node at a time by default, with drain off unless set [17]. Keep the container toolkit current, since each of the seven toolkit CVEs in NVIDIA's table was fixed in a named toolkit and Operator release [24].

For buyers and capacity planners. Allocation figures overstated use in both cluster traces reviewed [33, 34]. Packing strategies and sharing methods are the levers the sources describe [12, 21, 22, 35].

Platform work. Rillor designs and builds Kubernetes platforms for superintelligence (SI) work on GPUs, on servers the client controls. Nodes carry role labels, GPU workloads are placed on the nodes that hold GPUs, and GPU driver and kernel versions are held and updated by plan with a node restart.

Limits of this analysis

- This is an explainer of documentation, changelogs and papers. No cluster was built and no benchmark was run for it.
- No source reviewed counts how many clusters use the device plugin, the GPU Operator or DRA, so the report gives no adoption figures.
- No source reviewed measures scheduling latency or throughput for DRA against the device plugin.
- The utilization and fragmentation figures come from traces of two companies' clusters: one ran on Apache YARN [33], and the other company's clusters ran on its Fuxi scheduling system in the 2022 paper [34] and on a customized Kubernetes system in the 2023 paper [35]. The 49% result comes from an emulated cluster [35].
- NVIDIA's documentation is cited at GPU Operator 26.7, device plugin v0.20.1 and DRA driver v0.5.1 [15, 18, 20]. Its maturity table and the DRA driver project's own pages describe the driver in different terms [18, 19, 20].

Sources

1. Device Plugins. Kubernetes documentation, last modified 27 July 2026, viewed 8 November 2026. https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/

2. Schedule GPUs. Kubernetes documentation, last modified 20 September 2024, viewed 8 November 2026. https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/

3. Dynamic Resource Allocation. Kubernetes documentation, last modified 27 July 2026, viewed 8 November 2026. https://kubernetes.io/docs/concepts/resource-management/dynamic-resource-allocation/

4. Kubernetes v1.34: DRA has graduated to GA. The DRA team, Kubernetes Blog, 1 September 2025, viewed 8 November 2026. https://kubernetes.io/blog/2025/09/01/kubernetes-v1-34-dra-updates/

5. Kubernetes v1.37: DRA Updates. Kashish Verma, Kubernetes Blog, 3 September 2026, viewed 8 November 2026. https://kubernetes.io/blog/2026/09/03/kubernetes-v1-37-dra-updates/

6. Kubernetes v1.37: Garhwal. Kubernetes v1.37 Release Team, Kubernetes Blog, 26 August 2026, viewed 8 November 2026. https://kubernetes.io/blog/2026/08/26/kubernetes-v1-37-release/

7. Resource Management for Pods and Containers. Kubernetes documentation, last modified 27 July 2026, viewed 8 November 2026. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

8. Resource Quotas. Kubernetes documentation, last modified 28 September 2026, viewed 8 November 2026. https://kubernetes.io/docs/concepts/policy/resource-quotas/

9. Taints and Tolerations. Kubernetes documentation, last modified 27 July 2026, viewed 8 November 2026. https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/

10. Assigning Pods to Nodes. Kubernetes documentation, last modified 10 February 2026, viewed 8 November 2026. https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/

11. Kubernetes Scheduler. Kubernetes documentation, last modified 16 February 2024, viewed 8 November 2026. https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

12. Resource Bin Packing. Kubernetes documentation, last modified 22 May 2026, viewed 8 November 2026. https://kubernetes.io/docs/concepts/scheduling-eviction/resource-bin-packing/

13. Control Topology Management Policies on a node. Kubernetes documentation, last modified 29 July 2026, viewed 8 November 2026. https://kubernetes.io/docs/tasks/administer-cluster/topology-manager/

14. About the NVIDIA GPU Operator. NVIDIA GPU Operator documentation, version 26.7, last updated 3 November 2026, viewed 8 November 2026. https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/overview.html

15. Release Notes. NVIDIA GPU Operator documentation, version 26.7, last updated 3 November 2026, viewed 8 November 2026. https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/release-notes.html

16. Installing the NVIDIA GPU Operator. NVIDIA GPU Operator documentation, version 26.7, last updated 3 November 2026, viewed 8 November 2026. https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/getting-started.html

17. GPU Driver Upgrades. NVIDIA GPU Operator documentation, version 26.7, last updated 3 November 2026, viewed 8 November 2026. https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/gpu-driver-upgrades.html

18. DRA Driver for NVIDIA GPUs. NVIDIA GPU Operator documentation, version 26.7, last updated 3 November 2026, viewed 8 November 2026. https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/dra-intro-install.html

19. DRA Driver for NVIDIA GPUs Overview. Kubernetes SIGs, dra-driver-nvidia-gpu documentation, last modified 23 July 2026, viewed 8 November 2026. https://dra-driver-nvidia-gpu.sigs.k8s.io/docs/

20. NVIDIA device plugin for Kubernetes (README). NVIDIA, GitHub repository NVIDIA/k8s-device-plugin, main branch at release v0.20.1 of 22 September 2026, viewed 8 November 2026. https://github.com/NVIDIA/k8s-device-plugin

21. GPU Operator with MIG. NVIDIA GPU Operator documentation, version 26.7, last updated 3 November 2026, viewed 8 November 2026. https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/gpu-operator-mig.html

22. Time-Slicing GPUs in Kubernetes. NVIDIA GPU Operator documentation, version 26.7, last updated 3 November 2026, viewed 8 November 2026. https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/gpu-sharing.html

23. Container Device Interface (CDI) and Node Resource Interface (NRI) Plugin Support. NVIDIA GPU Operator documentation, version 26.7, last updated 3 November 2026, viewed 8 November 2026. https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/cdi.html

24. Security Considerations. NVIDIA GPU Operator documentation, version 26.7, last updated 3 November 2026, viewed 8 November 2026. https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/security.html

25. CVE-2024-0132. National Vulnerability Database, NIST, published 26 September 2024, last modified 17 June 2026, viewed 8 November 2026. https://nvd.nist.gov/vuln/detail/CVE-2024-0132

26. CVE-2025-23266. National Vulnerability Database, NIST, published 17 July 2025, last modified 17 June 2026, viewed 8 November 2026. https://nvd.nist.gov/vuln/detail/CVE-2025-23266

27. Run GPUs in GKE Standard node pools. Google Cloud documentation, last updated 7 November 2026, viewed 8 November 2026. https://docs.cloud.google.com/kubernetes-engine/docs/how-to/gpus

28. Use GPUs on Azure Kubernetes Service (AKS). Microsoft Learn, 2 July 2026, viewed 8 November 2026. https://learn.microsoft.com/en-us/azure/aks/use-nvidia-gpu

29. NodePools, Karpenter documentation v1.15. Karpenter project, last modified 5 November 2026, viewed 8 November 2026. https://karpenter.sh/docs/concepts/nodepools/

30. Resource Flavor, Kueue documentation. Kubernetes SIGs, last modified 6 November 2026, viewed 8 November 2026. https://kueue.sigs.k8s.io/docs/concepts/resource_flavor/

31. KAI Scheduler (README). kai-scheduler/KAI-Scheduler on GitHub, release v0.19.0 of 11 November 2026, viewed 13 November 2026. https://github.com/kai-scheduler/KAI-Scheduler

32. Kubernetes AI Conformance Checklist, v1.37 (AIConformance-1.37.yaml). Cloud Native Computing Foundation, cncf/k8s-ai-conformance on GitHub, 26 August 2026, viewed 8 November 2026. https://github.com/cncf/k8s-ai-conformance/blob/main/docs/AIConformance-1.37.yaml

33. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao and Fan Yang, USENIX ATC 2019, USENIX Association, July 2019, viewed 8 November 2026. https://www.usenix.org/conference/atc19/presentation/jeon

34. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang and others, USENIX NSDI 2022, USENIX Association, April 2022, viewed 8 November 2026. https://www.usenix.org/conference/nsdi22/presentation/weng

35. Beware of Fragmentation: Scheduling GPU-Sharing Workloads with Fragmentation Gradient Descent. Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang and Liping Zhang, USENIX ATC 2023, USENIX Association, July 2023, viewed 8 November 2026. https://www.usenix.org/conference/atc23/presentation/weng

36. CHANGELOG-1.35. Kubernetes project, GitHub repository kubernetes/kubernetes, viewed 8 November 2026. https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG/CHANGELOG-1.35.md

37. Releases. Kubernetes, viewed 13 November 2026. https://kubernetes.io/releases/

38. Platform Support. NVIDIA GPU Operator documentation, version 26.7, last updated 3 November 2026, viewed 8 November 2026. https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/platform-support.html

39. Overview. NVIDIA Container Toolkit documentation, last updated 18 September 2026, viewed 8 November 2026. https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/index.html

40. Allocate Devices to Workloads with DRA. Kubernetes documentation, last modified 27 July 2026, viewed 8 November 2026. https://kubernetes.io/docs/tasks/configure-pod-container/assign-resources/allocate-devices-dra/

41. Admission Control in Kubernetes. Kubernetes documentation, last modified 28 September 2026, viewed 8 November 2026. https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/

42. About node pools. Google Cloud documentation, last updated 7 November 2026, viewed 8 November 2026. https://docs.cloud.google.com/kubernetes-engine/docs/concepts/node-pools

43. GPU Feature Discovery (README). NVIDIA, GitHub repository NVIDIA/k8s-device-plugin, docs/gpu-feature-discovery, viewed 8 November 2026. https://github.com/NVIDIA/k8s-device-plugin/blob/main/docs/gpu-feature-discovery/README.md

44. Release v26.7.2. NVIDIA, GitHub repository NVIDIA/gpu-operator, 14 October 2026, viewed 8 November 2026. https://github.com/NVIDIA/gpu-operator/releases/tag/v26.7.2

45. Release v1.37.2. Kubernetes project, GitHub repository kubernetes/kubernetes, 28 October 2026, viewed 13 November 2026. https://github.com/kubernetes/kubernetes/releases/tag/v1.37.2

About this report

Published by Rillor on 14 November 2026. Superintelligence (SI) tools assist research and drafting, and every figure cites its source. Rillor builds and secures SI systems, supplies the data they learn from and delivers the infrastructure they run on. We apply SI to financial and compute markets.

Cite as: Rillor. GPU scheduling on Kubernetes: operators, sharing and node pools. 14 November 2026. https://rillor.com/insights/gpu-scheduling-on-kubernetes

Notices

Rillor is not a registered investment adviser or commodity trading advisor and does not provide investment or trading advice. Research, forecasts, data and software described on this site are for research and engineering use. Nothing here is an offer or recommendation to buy or sell any security, commodity interest or digital asset. Past or simulated results do not indicate future results.

NVIDIA, AMD, Ceph, Kubernetes and related product names are trademarks of their owners and are used only to identify products. No affiliation or endorsement is implied.

Work with Rillor

Send the workloads, the GPUs and servers they run on, the teams that deploy to the cluster and who runs it afterward. Contact us at rillor.com/contact?type=kubernetes or contact@rillor.com. More on Kubernetes platforms: rillor.com/compute/kubernetes