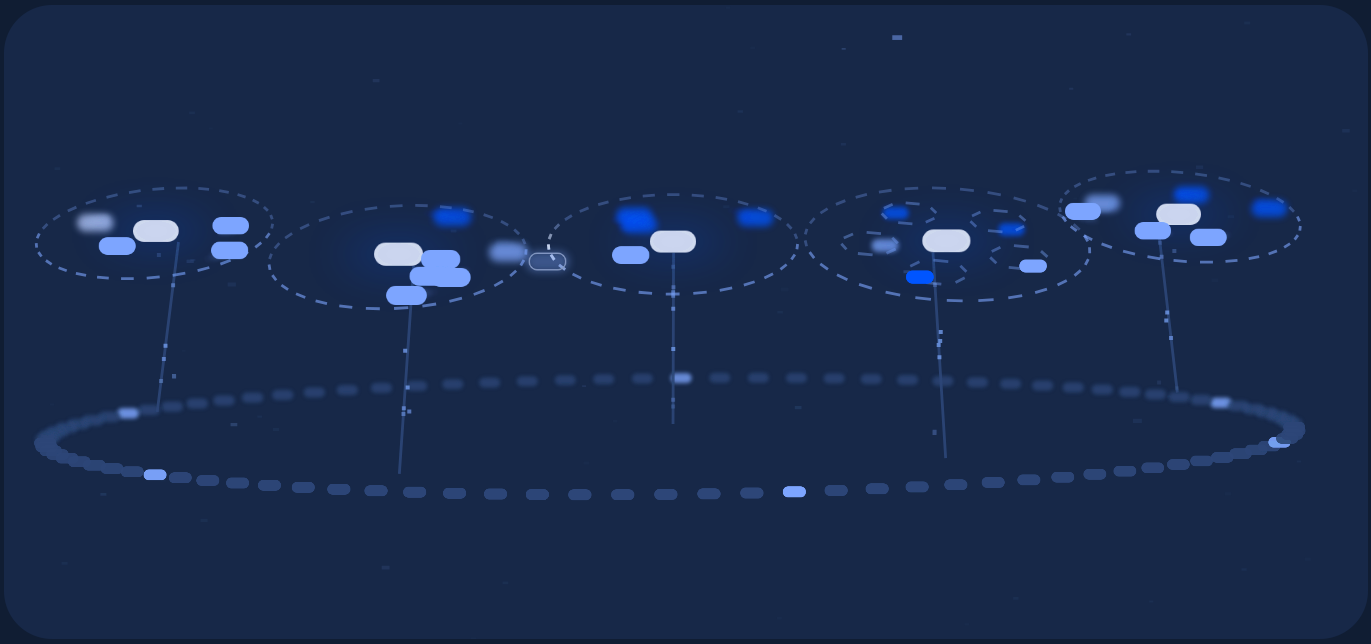
Rillor

EXPLAINER

MIG, MPS or time-slicing: sharing GPUs on Kubernetes

MIG vs time-slicing vs MPS on Kubernetes: what each GPU sharing method isolates, what measurements show, how the device plugin exposes it and how to size a slice.



Contents

01 Summary

02 Background

03 Method

04 Findings

05 Implications

06 Limits of this analysis

07 Sources

Summary

Kubernetes gives each container whole GPUs, and three NVIDIA methods let several workloads share one: time-slicing, the Multi-Process Service (MPS) and Multi-Instance GPU (MIG) [2].

- By default a Kubernetes device plugin hands out whole devices, which cannot be shared between containers [1]. In a 2020 trace of one production cluster with over 6,000 GPUs, the median instance used 0.042 GPUs' worth of streaming multiprocessors [3].
- Time-slicing advertises each GPU as several replicas that take turns on it. NVIDIA's GPU Operator documentation says there is no memory or fault isolation between replicas, and that a request for more than one replica does not guarantee a proportional share of compute [5].
- MPS, NVIDIA's Multi-Process Service, runs kernels from several processes on one GPU at the same time, and each client process has its own GPU address space. NVIDIA calls its error containment limited: a fatal fault is reported to every client on the GPUs where the fault is contained [13].
- MIG, Multi-Instance GPU, splits a GPU of the Ampere generation or later into up to seven instances, each with its own paths through the memory system, chosen from a fixed menu of profiles for each GPU model [17, 19, 20]. On an H100 80GB the menu runs from seven 1g.10gb instances to one 7g.80gb [20].
- In a 2026 study on an A30, MPS improved performance by up to 30% in favorable pairings and worsened it by around 30% under memory contention, while MIG gave more consistent gains at a higher overhead [25].
- MIG isolates more than the other two methods, and still not everything. Studies report timeouts in a two-process inference test under MIG, contention for shared PCIe bandwidth and, in an October 2026 preprint on one platform, covert channels across MIG instances [26, 28, 29].
- In Kubernetes, the NVIDIA device plugin applies time-slicing or MPS to every GPU on a node, never to one GPU alone, and exposes MIG either as plain GPUs with one profile per node (the single strategy) or as one resource per profile (the mixed strategy) [6, 22]. Dynamic Resource Allocation (DRA), which the Kubernetes documentation lists as stable since v1.35, lets Pods share a device by referencing the same resource claim [30].
- Trust shapes the choice across tenants. Google's GKE documentation limits GPU time-sharing, its name for time-slicing, to workloads in the same trust boundary [10], and NVIDIA's MPS multiuser mode drops isolation between users [13].

Background

Why Kubernetes hands out whole GPUs

Kubernetes offers GPUs to Pods through device plugins. In its list of how device resources differ from others, the Kubernetes documentation says extended resources are only supported as integers and cannot be overcommitted, and that devices cannot be shared between containers [1]. Google's GKE documentation spells out the result. A workload can ask for a fraction of a CPU, but a Pod manifest must request GPUs in integers, so an entire physical GPU goes to one container even when it needs only a fraction [2]. HAMi, a CNCF project for GPU sharing, gives the example of a model that needs 2 GiB of a 24 GiB card and still receives the whole card [4].

How much of a GPU a workload uses

The NSDI 2022 paper "MLaaS in the Wild" studied a two-month trace, from July and August 2020, of a production machine learning cluster with over 6,000 GPUs [3]. The median instance used 0.042 GPUs' worth of streaming multiprocessors (SMs), the GPU's compute units [3]. Comparing the measured system with a simulated replay of the trace without GPU sharing, the authors found that only 50% of the GPUs were needed with sharing on average, and that sharing saved up to 73% in the peak hour of around 10 a.m. [3]. The trace covers one operator's workloads on older GPU classes, the P100, T4 and V100 among them [3].

What the GPU does between processes

NVIDIA's MPS documentation describes the default. The GPU's time-sliced scheduler runs work from different CUDA contexts, and work from different contexts cannot run on the compute engine at the same time, which can leave compute idle when one context's work does not fill the GPU [11]. GKE's documentation adds that NVIDIA GPUs since the Pascal architecture support instruction-level preemption, which gives every process a fair timeslice [2].

	Time-slicing	MPS	MIG
How a GPU is shared	Replicas take turns on the whole GPU	Kernels from several processes run at once	The GPU is split into hardware instances
Memory	No isolation between replicas	Isolated address spaces, optional limits	Dedicated memory per instance
Faults	One fault domain	Limited containment	Fault isolation per instance
GPUs	Includes older GPUs without MIG	Full GPUs only, in the device plugin	Ampere generation and later

	Time-slicing	MPS	MIG
Sources	[5, 6]	[6, 13]	[5, 17, 19]

Table 1. How each method shares a GPU, as NVIDIA's documentation describes it.

This report is for platform engineers who run GPU nodes for inference services, notebooks and batch jobs, and for the people who decide how teams share those nodes.

Method

What was examined. Time-slicing, MPS and MIG as NVIDIA documents them, how the NVIDIA device plugin and GPU Operator expose each one on Kubernetes, what papers and preprints measured when workloads share a GPU, and how Dynamic Resource Allocation in Kubernetes itself lets Pods share a device.

Sources. NVIDIA's documentation, blog and repositories [5, 6, 7, 8, 11, 12, 13, 15, 16, 17, 18, 19, 20, 21, 22], Kubernetes documentation [1, 30], GKE, EKS and AKS documentation [2, 9, 10, 14, 23, 24], HAMi's documentation [4], the CNCF's conformance checklist for Kubernetes v1.37 [31], one conference paper [3] and five arXiv papers [25, 26, 27, 28, 29].

How. Each claim comes from the source's own text, opened on 8 November 2026. Where a cloud provider's documentation differs from NVIDIA's, this report gives each statement as its publisher's own. Features are described by what they do and the version that has them. Numbers computed for this report are labelled. Models in the studies are described by architecture. No benchmarks were run for this report.

Findings

1. Time-slicing: replicas that take turns

NVIDIA's GPU Operator enables oversubscription of GPUs through extended options of the NVIDIA Kubernetes device plugin [5]. An administrator sets a number of replicas for a GPU, and each replica can be handed to a Pod on its own [5]. With `replicas: 4`, a node advertises four GPU resources for each GPU, so a machine with two GPUs advertises eight [5].

A replica gives shared access, where a normal request gives exclusive access [5]. The GPU gives an equal share of time to all GPU processes across all the Pods, and a request for more than one time-sliced GPU does not guarantee a proportional amount of compute [5]. With the `failRequestsGreaterThanOne` setting on, such a request fails with an UnexpectedAdmissionError [5]. The device plugin's README recommends turning it on and keeps it off by default for backward compatibility [6].

Node labels tell the scheduler which GPUs are shared. GPU Feature Discovery adds a replicas label and appends `-SHARED` to the product label, for example `nvidia.com/gpu.product = A100-SXM4-40GB-SHARED`, so a node selector can ask for shared or exclusive GPUs [5]. Since CUDA 11.1, `nvidia-smi compute-policy --set-timeslice` sets the length of a slice to DEFAULT, SHORT, MEDIUM or LONG [8]. Time-slicing also serves older GPUs that do not support MIG [5], and Amazon's EKS documentation notes that it uses software only and needs no special hardware [9].

2. What time-slicing leaves unisolated

NVIDIA's documentation is plain: unlike MIG, there is no memory or fault isolation between replicas [5]. The device plugin's README adds that each workload can reach the GPU memory and runs in the same fault domain as the others, so if one workload crashes, they all do [6]. NVIDIA's 2022 blog lists the tradeoffs as increased latency, jitter and potential out-of-memory conditions when many applications share a GPU this way [8].

Amazon's documentation says one Pod can exhaust the memory other Pods rely on, the GPU scheduler shares compute across slots without guarantees, and the context switching reduces training efficiency [9]. Microsoft's AKS documentation says time-slicing gives no hardware-level isolation, so workloads contend for the same memory and compute [23]. Google's GKE guide, which calls the method GPU time-sharing, states that GKE enforces address space, performance and error isolation between containers on a time-shared GPU, and that it does not enforce memory limits [10]. That statement describes GKE's implementation, and NVIDIA's describes the device plugin's [5, 6].

Monitoring changes too. NVIDIA's DCGM-Exporter cannot associate metrics with containers when time-slicing is on [5], and GKE does not collect container-level accelerator metrics on GPUs shared this way [2]. GKE allows up to 48 containers to time-share one physical GPU [10].

3. MPS: kernels from several processes at once

NVIDIA describes MPS as a lightweight runtime service for cooperative multi-process CUDA work, with a control daemon, a server that owns the GPU scheduling resources and a client runtime built into `libcuda.so` [12]. MPS lets kernels from different processes execute in parallel on the GPU [8]. Clients submit work directly to the GPU without passing through the server, and each owns its own GPU address space [11]. NVIDIA lists better GPU utilization and fewer context switches as the benefits, with memory and SM partitioning, priorities and dynamic resource adjustment [12].

MPS helps, in NVIDIA's guidance, when each process does not generate enough work to saturate the GPU, because the leftover capacity can run kernels from other processes [13]. GKE's documentation recommends MPS for batch processing of small jobs and describes it as suited to cooperative processes acting as one application, such as MPI jobs [2].

Since CUDA 11.4, two settings limit a client: `CUDA_MPS_PINNED_DEVICE_MEM_LIMIT` for memory and `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` for threads [8]. A client held to a device memory limit gets an out-of-memory error when it allocates beyond it [13]. The thread limit works differently. NVIDIA's documentation says it reserves no dedicated resources for any client and only limits what a client can use, and kernels from different clients may run on the same SM [13]. GKE allows up to 48 containers to share one GPU through MPS, and 16 before the Volta generation [14].

4. Where MPS isolation ends

NVIDIA calls MPS error containment limited [13]. A fatal GPU fault is reported to all the clients running on the GPUs where the fault is contained, without saying which client caused it, and the server waits for every affected client to exit before it recreates the GPU contexts [13]. Ending a client without synchronizing its outstanding GPU work can leave the server and other clients in an undefined state, with hangs, failures or corruption [13].

Users matter. Only one user on a system may have an active MPS server, and requests from other users queue, which serializes access to the GPU between users [13]. A multiuser mode accepts clients with different user IDs while dropping isolation between users [13]. NVIDIA's architecture page warns that in this mode a fatal fault from one client may bring down another user's client that shares any GPU with it [11].

On GKE, MPS limits each container's pinned device memory and active threads. Memory bandwidth, encoders and decoders fall outside those limits, so containers can slow each other [14]. MPS on GKE also needs `hostIPC: true` on Pods, which GKE's documentation says gives the container access to a host resource and introduces security risks [14].

In the NVIDIA device plugin, MPS gives each client an equal fraction of device memory and caps its compute [6]. With `replicas: 10` on a node with 8 GPUs, the node advertises 80 `nvidia.com/gpu` resources, each with a tenth of a GPU's memory and compute [6]. MPS there works only with full GPUs, not with MIG devices, and time-slicing and MPS are mutually exclusive [6]. Amazon's documentation notes that time-slicing sets the

GPU compute mode to DEFAULT while MPS requires EXCLUSIVE_PROCESS, so the two never run on the same physical GPU at the same time [9]. The plugin added MPS in v0.15.0, released on 17 April 2024. Its release notes list known issues: the plugin and the MPS control daemon are not synchronized under restarts or configuration changes, so workloads may crash if they lose access to the shared resources [7].

5. MPS v3: named servers, namespaces and memory partitioning

NVIDIA's documentation covers MPS v3, an opt-in control daemon interface with named servers, namespaces and file-based configuration [12]. Legacy MPS, version 2, remains the default, and version 3 is selected with `-p 3` or `CUDA_MPS_PROTOCOL_VERSION=3` [15]. Where v2 allows one active server per user ID, v3 runs several named servers at once, and a namespace subdivides a server's resources for a group of clients, optionally tied to a cgroup v2 subtree [15].

MPS v3 memory partitioning divides a GPU's device memory across cgroups and containers, so that one tenant cannot monopolize the GPU and schedulers such as Kubernetes can pack workloads by memory [16]. A hard limit fails allocations beyond it, and a soft limit is a target that makes a client a candidate for eviction once crossed [16]. It needs Linux with cgroup v2, CUDA 13.4 or newer and a device without MIG [16].

6. MIG: hardware partitions from a fixed menu

MIG lets GPUs from the NVIDIA Ampere architecture on be partitioned into up to seven separate GPU instances [17]. Each instance's processors have separate, isolated paths through the memory system. The on-chip crossbar ports, L2 cache banks, memory controllers and DRAM address buses are each assigned to one instance [17]. NVIDIA says this gives predictable throughput and latency, with the same L2 cache allocation and DRAM bandwidth, even when other tasks thrash their own caches or saturate their DRAM interfaces [17]. MIG also partitions SMs and engines such as copy engines and decoders, with fault isolation between clients [17].

The partitions come in fixed sizes. A memory slice is roughly one eighth of the GPU's memory capacity and bandwidth, and an SM slice roughly one seventh of its SMs [18]. On the A100-40GB that means eight 5 GB memory slices and seven SM slices, so profile `1g.5gb` holds one of each and `4g.20gb` four [18]. The number of slices an instance can have is not arbitrary [18].

MIG needs compute capability 8.0 or later [19]. Seven instances is the maximum on the A100, H100, H200, B200, B300 and GB200, four on the A30 and RTX PRO 6000 Blackwell, and two on the RTX PRO 5000 and RTX PRO 4500 Blackwell [19].

GPU	One-slice profile (count)	Other profiles (count)	Full-GPU profile
H100 80GB	1g.10gb (7)	1g.20gb (4), 2g.20gb (3), 3g.40gb (2), 4g.40gb (1)	7g.80gb

GPU	One-slice profile (count)	Other profiles (count)	Full-GPU profile
H200 141GB	1g.18gb (7)	1g.35gb (4), 2g.35gb (3), 3g.71gb (2), 4g.71gb (1)	7g.141gb
B200 180GB	1g.23gb (7)	1g.45gb (4), 2g.45gb (3), 3g.90gb (2), 4g.90gb (1)	7g.180gb
B300 288GB	1g.34gb (7)	1g.67gb (4), 2g.67gb (3), 3g.135gb (2), 4g.135gb (1)	7g.269gb
RTX PRO 6000 Blackwell 96GB	1g.24gb (4)	2g.48gb (2)	4g.96gb

Table 2. MIG profiles and the maximum count of each on one GPU, from NVIDIA's MIG User Guide [20].

Profile choice caps the count. On an H100 80GB, seven 1g.10gb instances add up to 70 GB by their profile names, while the 1g.20gb profile allows four instances (computed for this report from [20]).

7. MIG in operation: resets, strategies and reconfiguration

Turning MIG on is a privileged step. On the A100 and A30, setting MIG mode requires a GPU reset and so super-user privileges, after which instance management is dynamic [21]. Daemons that hold handles on the driver, DCGM among them, must stop first, and toggling MIG mode needs the CAP_SYS_ADMIN capability [21]. Amazon's EKS documentation says changing MIG mode or the layout requires a GPU reset and cannot be done in place [24]. Microsoft's AKS documentation says its MIG configurations are static at the node pool level, so a change requires reprovisioning nodes [23].

On Kubernetes, the GPU Operator deploys MIG Manager to manage MIG configuration on nodes [22]. MIG Manager enables MIG mode and creates the MIG devices when a configuration is applied, chosen by a node label such as `nvidia.com/mig.config=all-1g.10gb` [22]. It requires that no user workloads run on the GPUs being configured, and it terminates the GPU Pods before it enables MIG mode [22]. Some nodes need a reboot, so the documentation says a node might need to be cordoned first [22].

Two strategies expose the result to Pods.

- **Single.** Every MIG device appears as `nvidia.com/gpu`. If MIG is on for any GPU of the node, it must be on for every GPU, each GPU needs one or more MIG devices, and all devices use the same profile [22]. An H100 80GB with `all-1g.10gb` reports `nvidia.com/gpu.count: 7` [22].

- **Mixed.** Each MIG device appears under its profile, such as `nvidia.com/mig-1g.10gb`, and GPUs without MIG stay `nvidia.com/gpu` [22]. On an H100, `all-balanced` gives two 1g.10gb, one 2g.20gb and one 3g.40gb [22].

Methods can be stacked. NVIDIA's GPU Operator documentation describes time-slicing MIG instances [5], the device plugin lists mixed-strategy MIG resources among those it can time-slice [6], and GKE recommends combining MIG with time-sharing or with MPS [2]. Amazon's EKS documentation says time-slicing has no effect on a MIG instance and points to MPS instead [9], so the platforms' documentation differs on this point. NVIDIA's MIG guide says CUDA MPS is supported on top of MIG, with the maximum number of clients lowered in proportion to the compute instance size [21], while the device plugin does not support MPS sharing on MIG devices [6].

8. MIG vs time-slicing vs MPS in measurements

Three studies measured MPS and MIG side by side, and one of them also measured time-slicing.

Kernel pairs on an A30. Villarrubia, Costero, Igual and Olcoz ran pairs of workloads from the Altis benchmark suite and other sources, mainly on an A30 with 24 GiB, checking some results on a 40 GiB A100, with CUDA 13.0 and locked clocks [25]. In favorable scenarios, MPS improved performance by up to 30% and reduced energy by about 20%, using its provisioning option to keep one job from monopolizing resources [25]. Under memory contention it worsened performance by around 30% [25]. In the memory-intensive pairs with the greatest slowdown, those including the Sort test, MPS reached an average memory utilization of roughly 60%, against barely over 20% for time-slicing [25]. A dominant matrix-multiplication job monopolized the SMs under basic MPS, and a 70-30 SM split worked better than the other splits tested [25]. Many MIG pairs reduced overall slowdown by 20%, some by up to 30%, and memory-intensive pairs finished almost twice as fast as under MPS [25]. The authors found MPS results highly polarized, and MIG's rigid partitioning a cause of underutilization when a job's needs fit no instance well, more so on the A30 with its fewer options [25].

Inference isolation on an A100. Martín, Flich and Hernández compared MPS, MIG and Green Contexts on an A100 and a Jetson Orin with image classification models such as ResNet18 and ViT [26]. They found that MIG provided a high level of isolation, and in their conclusions more robust temporal isolation than the other mechanisms [26]. Still, with two processes on the A100, the first timeouts appeared despite MIG at higher frequencies, and for the smallest models tested MIG's overhead made it the least efficient option for isolation, though timeouts there stayed minimal [26]. MIG's partitions cannot be reallocated or resized once created [26]. MPS, they write, does not provide true isolation, in particular of memory bandwidth or SM scheduling, and remains a practical choice for general-purpose workloads [26].

Training on an A100. Robroek, Yousefzadeh-Asl-Miandoab and Tözün compared a MIG-enabled A100 with plain multi-process submission and with MPS for model training [27]. Collocating training runs could give up to four times the training throughput, despite longer epochs [27]. MIG helped when model sizes matched its partition options, and its rigid partitions could leave the GPU underused for mixed, changing workloads [27]. The authors recommend MPS for collocating the training jobs of a single user [27].

Study	GPUs	Workloads	MPS	MIG
Villarrubia et al., 2026 [25]	A30, some results checked on an A100	Benchmark pairs	Up to 30% better in favorable pairs, around 30% worse under memory contention	More consistent gains, higher overhead
Martín et al., 2026 [26]	A100, Jetson Orin	Image classification inference	No true isolation of memory bandwidth or SM scheduling	More temporal isolation, timeouts still seen
Robroek et al., 2022 to 2023 [27]	A100	Model training	Recommended for one user's training jobs	Helps when model sizes fit the partitions

Table 3. What three studies measured, in their own setups.

9. Where MIG isolation stops

Hardware partitions leave some resources shared. Ting, Chou, Chen and Chung report that contention persists under MIG through shared components such as PCIe bandwidth [28]. They also describe fragmentation, which follows from MIG's limited number of valid configurations and rigid profile placement, especially after jobs arrive and finish [28]. Their scheduler improved makespan by up to 35% [28]. Amazon's EKS documentation adds that compute is not strictly proportional to instance size, because memory bandwidth and cache behavior differ across profiles [24].

Security research has reached MIG too. In PAGE404, an arXiv preprint whose third version is dated 21 October 2026, Nazaraliyev, Zhang and Abu-Ghazaleh report two side channels in counter-driven page migration on the Grace Hopper GH200 that work across MIG partitions [29]. Their covert channels reached roughly 8,500 bits per second through eviction-based signaling and 8,100 bits per second through migration delays [29]. The authors write that the results call into question the isolation guarantees of current MIG deployments shared by several tenants [29].

10. Setting it up on Kubernetes

The device plugin. The NVIDIA device plugin offers two flavors of sharing, time-slicing and MPS, applies the same one to every GPU on a node and allows no per-GPU setting [6]. A node label, `nvidia.com/gpu.sharing-strategy`, reads `none`, `mps` or `time-slicing` [6]. MIG comes through the GPU Operator's MIG Manager and the single or mixed strategy [22]. On GKE, a container on time-shared or MPS GPUs can request only one GPU, and the number requested is not a measure of the compute it gets [2]. Installing the operator, its drivers and node labels belongs to the wider subject of GPU scheduling on Kubernetes.

Dynamic Resource Allocation. The Kubernetes documentation lists DRA as stable since v1.35 [30]. Pods can share a device by referencing the same resource claim [30]. Time-slicing through the NVIDIA DRA driver sits behind a `TimeSlicingSettings` feature gate that is off by default [9]. Allocation through DRA belongs to the wider subject of GPU scheduling on Kubernetes.

The CNCF checklist. The CNCF's conformance checklist for Kubernetes v1.37, added on 26 August 2026, lists GPU sharing at level SHOULD [31]. For accelerators that support static GPU sharing, a platform should provide well-defined mechanisms for at least one sharing strategy, expose hardware partitions as schedulable resources if the accelerator supports partitioning, and allow oversubscription if it supports software sharing such as time-slicing [31].

11. Across tenants, trust decides the method

The documentation ties each method to a level of trust. GKE's time-sharing guide says that, to avoid security issues, only workloads in the same trust boundary should share a GPU this way [10]. MPS serves one user per server by default, and its multiuser mode drops isolation between users and lets a fault from one user's client bring down another's [11, 13]. NVIDIA presents MIG for cloud providers with several tenants, saying it ensures one client cannot affect the work or scheduling of other clients [17]. The 2026 covert-channel results on one platform, in their authors' words, call such isolation guarantees into question [29]. With the NVIDIA DRA driver, Pods share a GPU only through a namespace-scoped claim, so sharing cannot cross namespaces [9].

Method	GKE recommends it for	AKS lists it for
Time-slicing	Bursty and interactive workloads with idle periods	Experimentation with variable GPU loads
MPS	Batch processing of small jobs, cooperative processes acting as one application	Low-latency, high-throughput workloads
MIG	Parallel workloads that need resiliency and quality of service	Production workloads

Table 4. What two cloud providers' documentation suggests each method for [2, 23].

12. Sizing a slice starts with memory

Memory is the limit each method sets first, or leaves unset. A MIG profile fixes the memory of an instance: the smallest instance holds 10 GB on an H100 80GB (1g.10gb) and 34 GB on a B300 (1g.34gb) [20]. The device plugin's MPS mode gives each replica an equal fraction of device memory [6], and MPS v3 can set hard and soft limits across cgroups and containers [16]. Time-slicing sets no memory limit at all, and GKE advises setting GPU memory limits in the workloads themselves to avoid running out of memory [10].

Because compute is not strictly proportional to instance size, Amazon advises benchmarking a workload on the intended profile before sizing partitions [24]. A worked case, computed for this report from [20]: a hypothetical workload that needs 15 GB of GPU memory does not fit a 1g.10gb instance on an H100 80GB. It fits a 1g.20gb or a 2g.20gb, of which one GPU holds up to four or three (computed from [20]).

Implications

For platform teams. Choose the method node by node, since the device plugin applies one sharing method to every GPU on a node [6]. Turn on `failRequestsGreaterThanOne` with time-slicing, so that no Pod asks for several replicas expecting more compute [5, 6]. Treat MIG changes as maintenance, because changing MIG mode or the layout requires a GPU reset [21, 24] and MIG Manager terminates the node's GPU Pods first [22]. Expect per-container GPU metrics to be unavailable under time-slicing [5], and on GKE under MPS as well [2].

For teams that share a cluster. Time-slicing suits workloads that tolerate latency compromises and sit in one trust boundary [2, 10]. MPS suits cooperative processes of one user [2, 13, 27]. Tenants kept apart call for MIG partitions, with the shared PCIe path and the 2026 side-channel results in mind [17, 28, 29].

For those who own the budget. In the one cluster trace reviewed, the cluster with sharing needed on average half the GPUs that a simulated replay without sharing needed [3]. The saving depends on the pairing: MPS gains in good pairings and loses under memory contention [25].

For anyone reading vendor documentation. NVIDIA, Amazon and Google describe time-slicing's isolation and its use on MIG instances differently [5, 9, 10]. Read each statement as its publisher's own, for its own platform.

Platform work. Rillor designs and builds Kubernetes platforms for superintelligence (SI) work on GPUs, on servers the client controls. GPUs are offered to the scheduler through the device plugin, and a workload gets only the GPUs it requests.

Limits of this analysis

- This is an explainer of documentation and papers. No benchmark was run for it, and it ranks no method.
- The measurements come from an A30, an A100 and a Jetson Orin with benchmark kernels, image classification and training [25, 26, 27]. No source reviewed measures LLM serving across all three methods on H100 or later GPUs.
- No source reviewed gives a general figure for MIG's overhead. The studies report it for their own workloads [25, 26].
- The utilization figures come from one cluster's 2020 trace on older GPU classes [3].
- Cloud providers' pages describe their own implementations and differ from NVIDIA's on some points [9, 10, 23, 24].
- Several sources are preprints, PAGE404 among them, and its result covers one platform [29].
- NVIDIA's documentation is cited at device plugin v0.20.1 and GPU Operator 26.7 [5, 6, 22]. The CNCF item is at level SHOULD [31].

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. About GPU sharing strategies in GKE. Google Cloud documentation, last updated 7 November 2026, viewed 8 November 2026. https://docs.cloud.google.com/kubernetes-engine/docs/concepts/timesharing-gpus

3. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang and others, 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), USENIX Association, April 2022, pp. 945-960, viewed 8 November 2026. https://www.usenix.org/conference/nsdi22/presentation/weng

4. What is HAMi? HAMi project documentation, version v2.11.0, undated, viewed 8 November 2026. https://project-hami.io/docs

5. 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

6. 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

7. NVIDIA device plugin release v0.15.0. NVIDIA, GitHub, 17 April 2024, viewed 8 November 2026. https://github.com/NVIDIA/k8s-device-plugin/releases/tag/v0.15.0

8. Improving GPU Utilization in Kubernetes. Kevin Klues, Kyrylo Perelygin and Pramod Ramarao, NVIDIA Technical Blog, 16 June 2022, viewed 8 November 2026. https://developer.nvidia.com/blog/improving-gpu-utilization-in-kubernetes/

9. Use time-slicing with NVIDIA GPUs on Amazon EKS. Amazon Web Services, Amazon EKS User Guide, undated, viewed 8 November 2026. https://docs.aws.amazon.com/eks/latest/userguide/device-management-nvidia-time-slicing.html

10. Share GPUs across workloads with GPU time-sharing. Google Cloud documentation, last updated 7 November 2026, viewed 8 November 2026. https://docs.cloud.google.com/kubernetes-engine/docs/how-to/timesharing-gpus

11. MPS Architecture. NVIDIA documentation, last updated 4 November 2026, viewed 8 November 2026. https://docs.nvidia.com/deploy/mps/latest/architecture.html

12. Multi-Process Service. NVIDIA documentation, last updated 4 November 2026, viewed 8 November 2026. https://docs.nvidia.com/deploy/mps/latest/index.html

13. When to Use MPS. NVIDIA documentation, last updated 4 November 2026, viewed 8 November 2026. https://docs.nvidia.com/deploy/mps/latest/when-to-use-mps.html

14. Share GPUs with multiple workloads using NVIDIA MPS. Google Cloud documentation, last updated 7 November 2026, viewed 8 November 2026. https://docs.cloud.google.com/kubernetes-engine/docs/how-to/nvidia-mps-gpus

15. MPS v3 Interface. NVIDIA documentation, last updated 4 November 2026, viewed 8 November 2026. https://docs.nvidia.com/deploy/mps/latest/mpsv3-interface.html

16. MPS v3 Memory Partitioning. NVIDIA documentation, last updated 4 November 2026, viewed 8 November 2026. https://docs.nvidia.com/deploy/mps/latest/mpsv3-memory-partitioning.html

17. MIG User Guide: Introduction. NVIDIA documentation, last updated 4 November 2026, viewed 8 November 2026. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/latest/introduction.html

18. MIG User Guide: Concepts. NVIDIA documentation, last updated 4 November 2026, viewed 8 November 2026. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/latest/concepts.html

19. MIG User Guide: Supported GPUs. NVIDIA documentation, last updated 4 November 2026, viewed 8 November 2026. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/latest/supported-gpus.html

20. MIG User Guide: Supported MIG Profiles. NVIDIA documentation, last updated 4 November 2026, viewed 8 November 2026. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/latest/supported-mig-profiles.html

21. MIG User Guide: Deployment Considerations. NVIDIA documentation, last updated 4 November 2026, viewed 8 November 2026. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/latest/deployment-considerations.html

22. 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

23. GPU node partitioning strategies in Azure Kubernetes Service (AKS). Microsoft Learn, updated 22 May 2026, viewed 8 November 2026. https://learn.microsoft.com/en-us/azure/aks/concepts-gpu-partitioning

24. Use multi-instance GPUs (MIG) with NVIDIA GPUs on Amazon EKS. Amazon Web Services, Amazon EKS User Guide, undated, viewed 8 November 2026. https://docs.aws.amazon.com/eks/latest/userguide/device-management-nvidia-mig.html

25. A comprehensive evaluation of spatial co-execution on GPUs using MPS and MIG technologies. Jorge Villarrubia, Luis Costero, Francisco D. Igual and Katzalin Olcoz, arXiv:2604.22430, version 2, 29 April 2026 (first version 24 April 2026), viewed 8 November 2026. https://arxiv.org/abs/2604.22430

26. Performance Isolation for Inference Processes in Edge GPU Systems. Juan José Martín, José Flich and Carles Hernández, arXiv:2601.07600, version 3, 13 July 2026 (first version 12 January 2026), viewed 8 November 2026. https://arxiv.org/abs/2601.07600

27. An Analysis of Collocation on GPUs for Deep Learning Training. Ties Robroek, Ehsan Yousefzadeh-Asl-Miandoab and Pınar Tözün, arXiv:2209.06018, version 3, 24 April 2023 (first version 13 September 2022), viewed 8 November 2026. https://arxiv.org/abs/2209.06018

28. An Online Fragmentation-Aware Scheduler for Managing GPU-Sharing Workloads on Multi-Instance GPUs. Hsu-Tzu Ting, Jerry Chou, Ming-Hung Chen and I-Hsin Chung, arXiv:2512.16099, 18 December 2025, viewed 8 November 2026. https://arxiv.org/abs/2512.16099

29. PAGE404: Breaking GPU Isolation Through Hardware-assisted Page Migration. Ravan Nazaraliyev, Yicheng Zhang and Nael Abu-Ghazaleh, arXiv:2610.20962, version 3, 21 October 2026 (first version 14 October 2026), viewed 8 November 2026. https://arxiv.org/abs/2610.20962

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

31. AIConformance-1.37.yaml, requirement gpu_sharing. CNCF, GitHub repository cncf/k8s-ai-conformance, added 26 August 2026 (commit 287b44f0), viewed 8 November 2026. https://github.com/cncf/k8s-ai-conformance/blob/main/docs/AIConformance-1.37.yaml

About this report

Published by Rillor on 12 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. MIG, MPS or time-slicing: sharing GPUs on Kubernetes. 12 November 2026. https://rillor.com/insights/mig-mps-or-time-slicing

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