HAMI: An Open-Source Solution for Fine-Grained GPU Scheduling and Heterogeneous Resource Management in Kubernetes

1. Introduction: The Need for Advanced GPU Scheduling in Kubernetes

The proliferation of Artificial Intelligence (AI) and Machine Learning (ML) workloads has positioned Graphics Processing Units (GPUs) as indispensable resources within modern computing infrastructure. Kubernetes has emerged as the de facto standard for orchestrating containerized applications, including these GPU-accelerated workloads.[1] However, traditional Kubernetes scheduling mechanisms, primarily designed for CPU and memory, often fall short in efficiently managing the unique characteristics of GPUs.[3] Standard schedulers typically treat GPUs as opaque, indivisible resources, leading to underutilization, increased costs, and challenges in supporting diverse AI/ML tasks that may not require an entire GPU.[3]

This limitation has spurred the development of specialized GPU scheduling solutions. Among these is **HAMi (Heterogeneous AI Computing Virtualization Middleware)**, formerly known as k8s-vgpu-scheduler. HAMi is an open-source project, accepted as a Cloud Native Computing Foundation (CNCF) Sandbox project on August 21, 2024.[5] It aims to provide fine-grained GPU sharing, resource isolation, and management of diverse heterogeneous computing devices within Kubernetes clusters.[5]

This report provides a comprehensive analysis of HAMI, delving into its architecture, core functionalities, installation, configuration, and its position within the broader ecosystem of Kubernetes GPU scheduling solutions. It explores how HAMI addresses the challenges of GPU sharing, supports various hardware accelerators, and integrates with other scheduling frameworks. The objective is to offer a deep understanding of HAMI's capabilities, its practical implications for developers and MLOps engineers, and its potential to enhance GPU resource management in Kubernetes environments.

2. Understanding HAMI: Core Concepts and Architecture

HAMI is designed as a middleware solution to bridge the gap between Kubernetes and the underlying heterogeneous hardware, primarily focusing on GPUs but extending to other AI accelerators. Its core design revolves around enabling device sharing and virtualization at a software level, aiming to improve resource utilization and flexibility without requiring modifications to user applications.[5]

2.1. HAMI's Architectural Components

HAMI's architecture comprises several key components that work in concert to deliver its GPU sharing and scheduling capabilities [5]:

- **Unified Mutating Webhook:** This component intercepts pod creation and modification requests. If a pod requests HAMI-managed resources (e.g., fractional GPUs), the webhook can modify the pod specification, for instance, by injecting necessary configurations or directing the pod to HAMI's custom scheduler.[5]
- **Unified Scheduler Extender (or Custom Scheduler Logic):** HAMI implements custom scheduling logic, often as a Kubernetes scheduler extender. This logic is responsible for making fine-grained decisions about where to place pods based on the availability of fractional GPU resources and defined scheduling policies (e.g., binpack, spread).[5]
- **Device Plugins:** HAMI utilizes Kubernetes device plugins for various supported hardware types. These plugins advertise virtualized device resources (e.g., vGPUs with specific memory/core allocations) to the Kubelet, making them discoverable and requestable by pods.[5]
- **HAMi-core (In-Container Control Component):** This is arguably the most critical component for HAMI's virtualization capabilities. HAMi-core is a user-space library (often libvgpu.so for NVIDIA GPUs) that operates within the container. It intercepts CUDA API calls (or similar APIs for other hardware) between the application and the native device driver.[13] This interception allows HAMi-core to enforce resource limits (memory, compute) and manage the virtualized view of the GPU presented to the application.

The interplay between these components allows HAMI to present a virtualized GPU environment to workloads. The webhook and scheduler ensure pods are matched with appropriate virtual resources, while the device plugin and HAMi-core enforce these allocations at the node and container level. This modular design, separating scheduling logic from device-level control, provides a flexible framework for managing heterogeneous resources.

Table 2.1: HAMi Core Architectural Components

Component	Primary Function	Key Interactions	Relevant Snippets
Mutating Webhook	Intercepts and modifies pod specifications for HAMI-managed resources.	Kubernetes API Server, HAMI Scheduler	[5]

Scheduler Extender/Logic	Makes fine-grained scheduling decisions based on virtual resource availability and policies.	Kubernetes Scheduler, HAMI Device Plugins, Pod Specifications	[5]
Device Plugins	Advertise virtualized device resources to Kubelet; manage device allocation on the node.	Kubelet, HAMi-core, HAMI Scheduler	[5]
HAMi-core (e.g., libvgpu.so)	User-space library in container; intercepts device API calls to enforce resource limits and virtualization.	Application, Native Device Drivers	[13]

2.2. GPU Sharing Mechanism: CUDA API Interception

The cornerstone of HAMI's GPU sharing for NVIDIA devices is its HAMi-core component, which employs a technique of CUDA API interception.[13] This user-space approach differentiates HAMI from hardware-based partitioning methods like NVIDIA MIG or kernel-level virtualization.

The mechanism typically involves the following steps:

1. **LD_PRELOAD Hijacking:** The HAMi-core library (e.g., libvgpu.so) is preloaded into the application's environment within the container. This allows it to intercept calls made by the application to the CUDA runtime library (libcudart.so) before they reach the actual CUDA driver (libcuda.so).[14]
2. **Memory Limiting:** When an application attempts to allocate GPU memory (e.g., via cuMemAlloc* calls), HAMi-core intercepts this call. It checks the requested allocation against the virtual GPU memory quota assigned to the container (tracked, for example, in shared memory). If the request exceeds the quota, the allocation is denied. Furthermore, API calls that query GPU memory information (e.g., cuMemGetInfo_v2) are "faked" to return values reflecting the virtual quota, not the total physical GPU memory.[14] This creates the illusion for the application that it has exclusive access to a smaller, dedicated GPU.
3. **Compute Limiting (Core Utilization):** HAMI implements a form of compute limiting, often described as a "soft" limit. For NVIDIA GPUs, this might involve a background thread that periodically polls actual GPU utilization (e.g., via NVML)

and adjusts a token-based system representing "virtual CUDA cores." When an application launches a CUDA kernel, it consumes tokens; if insufficient tokens are available, the launch might be delayed or throttled.[15] This mechanism aims to ensure that, over time, the application's compute usage adheres to its assigned percentage.

4. **Device Visibility:** HAMI, in conjunction with the device plugin, controls which physical GPU(s) or parts of a GPU are visible to the container, often by setting environment variables like NVIDIA_VISIBLE_DEVICES appropriately for the virtualized context.

This API interception method allows HAMI to offer fine-grained control over GPU memory and compute resources without requiring custom drivers or kernel modifications, aiming for "zero code changes" for existing applications.[9] However, being a user-space software solution, the robustness of isolation and potential performance overhead are important considerations compared to hardware-enforced mechanisms.[18]

2.3. Resource Isolation Capabilities

HAMI aims to provide resource isolation for shared devices, a critical feature for multi-tenancy and predictable performance.[5]

- **Memory Isolation:** HAMI asserts "hard isolation" of memory resources for NVIDIA GPUs and some other supported devices.[8] As described above, this is achieved by intercepting memory allocation APIs and ensuring a container cannot allocate more GPU memory than it has been assigned. For example, if a pod requests nvidia.com/gpumem: 3000, it will only see 3GB of GPU memory available.[9]
- **Compute Isolation:** For NVIDIA GPUs and Hygon DCUs, HAMI supports compute isolation, typically through the percentage-based core utilization control managed by HAMi-core.[7] This is generally a "soft" isolation, meaning it controls average utilization over time rather than providing strict, instantaneous compute partitioning like MIG. For Cambricon MLUs, compute isolation was listed as not supported or in development in some documents.[7]
- **Fault Isolation:** The documentation does not extensively detail fault isolation. User-space API interception methods generally offer less fault isolation than hardware partitioning (like MIG) or full hypervisor-based virtualization. A crash in one process sharing a GPU via HAMI could potentially affect other processes on the same physical GPU if the issue occurs at the driver or hardware level below HAMI's interception layer. This is a common characteristic of software-based sharing solutions like NVIDIA MPS as well.[14]

The effectiveness of these isolation mechanisms is a key differentiator for HAMI. While "hard memory isolation" through API call denial is a strong claim, the overall isolation

boundary is at the software level within the container, which differs from the hardware-level boundaries provided by technologies like NVIDIA MIG.

2.4. Heterogeneous Device Support

A significant aspect of HAMI's design is its goal to manage a diverse range of heterogeneous computing devices beyond just NVIDIA GPUs, providing a unified interface for users.[5]

Table 2.2: Supported Heterogeneous Devices in HAMI (with Isolation Features, as of early-mid 2025)

Product	Manufacturer	Memory Isolation	Compute Isolation	Multi-GPU Support (within a single pod for one task)	Key Snippet(s)
GPU	NVIDIA	Yes (Hard limit via API interception)	Yes (Soft limit, percentage-based core control)	Yes	[7]
MLU	Cambricon	Yes	No / In Development	No / In Development	[7]
DCU	Hygon	Yes	Yes	No / In Development	[7]
Ascend NPU	Huawei	Yes (with templates/granularity) / In Development	Yes (with templates/granularity) / In Development	No / In Development	[7]
Iluvatar GPU	Iluvatar CoreX	In Development	In Development	No / In Development	[7]
Mthreads	Mthreads	Supported	Supported	Supported	[5]

GPU		(details vary)	(details vary)	(details vary)	
Metax GPU	MetaX	Supported (details vary)	Supported (details vary)	Supported (details vary)	[12]
Intel GPU	Intel	Planned	Planned	Planned	[7]
AMD GPU	AMD	Planned	Planned	Planned	[7]

Note: "In Development" or "Planned" statuses are based on information available up to early-mid 2025 and may have changed. Multi-GPU support refers to a single task using multiple virtual devices that might span physical devices.

This multi-vendor support is achieved by having specific device plugin implementations and potentially different in-container control components tailored for each hardware type, all while aiming to expose them through a consistent set of resource requests and scheduling policies at the Kubernetes level.[5] This ambition to provide a common management layer across diverse AI accelerators is a key differentiator for HAMI in the open-source ecosystem. However, the maturity and feature parity (e.g., compute isolation, dynamic sharing capabilities) can vary between different supported hardware types.

3. Deploying and Utilizing HAMI

Deploying and configuring HAMI involves several steps, from preparing the Kubernetes nodes to defining pod specifications that request shared GPU resources. Understanding these aspects is crucial for effectively leveraging HAMI's capabilities.

3.1. Prerequisites for HAMI Installation

Before installing HAMI, certain prerequisites must be met on the Kubernetes cluster and its nodes, particularly those equipped with GPUs [5]:

Table 3.1: HAMI Installation Prerequisites

Category	Requirement	Notes	Key Snippet(s)
Kubernetes	Version ≥1.16 (some	Ensure compatibility with HAMI	[5]

	docs mention ≥1.18)	components.	
NVIDIA GPUs	NVIDIA drivers ≥440	Essential for GPU operation.	[5]
	nvidia-docker version > 2.0 (or nvidia-container-toolkit)	Enables containers to access NVIDIA GPUs.	[5]
	Default container runtime (Docker, containerd, CRI-O) configured to use nvidia as the default runtime.	Ensures GPU-enabled containers are correctly launched.	[5]
System Libs	glibc ≥2.17 & glibc < 2.30 (or < 2.3 for some docs)	Runtime library dependency.	[5]
Kernel	Version ≥3.10	Minimum kernel version.	[5]
Tooling	Helm > 3.0	HAMI is typically installed via a Helm chart.	[5]
Node Labeling	GPU nodes must be labeled (e.g., gpu=on) for HAMI's scheduler to manage them.	kubectl label nodes {nodeid} gpu=on	[5]

Meeting these prerequisites, especially the NVIDIA driver and runtime configurations on each GPU node, is fundamental. The node labeling step is also critical for HAMI's scheduler to identify and manage GPU-equipped nodes. The dependency on specific glibc versions suggests potential compatibility considerations on different Linux distributions.

3.2. Installation Process via Helm

HAMI is typically installed using Helm, which simplifies the deployment of its various components [5]:

1. **Add HAMI Helm Repository:**
   Bash
   ```
   helm repo add hami-charts https://project-hami.github.io/HAMi/
   helm repo update
   ```

   This command adds the official HAMI chart repository to the local Helm configuration.[5]
2. **Determine Kubernetes Server Version:**
   Bash
   ```
   kubectl version
   ```

   The Kubernetes server version is needed because some HAMI components, particularly the scheduler, might have images tagged according to the Kubernetes version they are compatible with.[9]
3. **Install HAMI Chart:**
   Bash
   ```
   helm install hami hami-charts/hami \
     --set scheduler.kubeScheduler.imageTag=vX.Y.Z \ # Replace vX.Y.Z with your K8s server version, e.g., v1.25.0
     -n kube-system # Or another namespace like hami-system
   ```

   This command installs HAMI. The scheduler.kubeScheduler.imageTag should be set according to the cluster's Kubernetes version. Various other configuration parameters can be set using --set flags or by providing a custom values file.[7]
4. Verify Installation:
   After installation, check the status of HAMI pods (typically hami-device-plugin or vgpu-device-plugin, and hami-scheduler or vgpu-scheduler):
   Bash
   ```
   kubectl get pods -n kube-system # Or the namespace used for installation
   ```

   The pods should be in the Running state.[9]

The Helm-based installation streamlines deployment, but careful attention to prerequisites and version compatibility (especially for the scheduler image) is necessary. For production environments, reviewing and customizing the Helm chart's values.yaml or using --set for specific configurations (like default scheduling policies, resource names, or device-specific parameters) is recommended.[11]

3.3. Core Configuration Parameters

HAMI's behavior is controlled through a combination of a central ConfigMap (hami-scheduler-device), Helm chart values, and pod annotations. These configurations dictate GPU sharing parameters, scheduling policies, and device-specific settings.[11]

Table 3.2: Key HAMI Configuration Parameters (via ConfigMap & Helm)

Parameter Name	Scope	Type	Default Value	Description	Key Snippet(s)
nvidia.deviceMemoryScaling	ConfigMap	Float	1.0	Ratio for NVIDIA GPU memory scaling. >1 enables virtual device memory (experimental). S * M total vGPU memory for physical M memory if set to S.	[11]
nvidia.deviceSplitCount	ConfigMap	Integer	10	Maximum number of tasks (vGPUs) that can be assigned to a single physical NVIDIA GPU.	[11]
nvidia.migStrategy	ConfigMap	String	"none"	Strategy for NVIDIA MIG. "none" ignores MIG; "mixed" allows allocating MIG devices	[11]

				as separate resources.	
nvidia.disableCoreLimit	ConfigMap	String	"false"	If "true", disables GPU core utilization limiting for NVIDIA GPUs.	[11]
nvidia.defaultMem	ConfigMap	Integer	0 (MB)	Default GPU memory (in MB) for a task if not specified. 0 means use 100% of available device memory for the vGPU.	[11]
nvidia.defaultCores	ConfigMap	Integer	0 (%)	Default GPU core percentage for a task if not specified. 0 allows fitting into any GPU with enough memory; 100 implies exclusive use of a physical GPU's cores.	[11]
nvidia.defaultGPUNum	ConfigMap	Integer	1	Default number of vGPUs if nvidia.com/gpu is not set	[11]

				but fractional resources are. 0 means ineffective.	
nvidia.resourceCountName	ConfigMap	String	"nvidia.com/gpu"	Resource name for requesting vGPU count.	[11]
nvidia.resourceMemoryName	ConfigMap	String	"nvidia.com/gpumem"	Resource name for requesting vGPU memory in MiB.	[11]
nvidia.resourceMemoryPercentageName	ConfigMap	String	"nvidia.com/gpumem-percentage"	Resource name for requesting vGPU memory as a percentage.	[11]
nvidia.resourceCoreName	ConfigMap	String	"nvidia.com/gpucores" (or "nvidia.com/cores")	Resource name for requesting vGPU core percentage.	[11]
devicePlugin.service.schedulerPort	Helm	Integer	31998	NodePort for the scheduler webhook service.	[11]
scheduler.defaultSchedulerPolicy.nodeSchedulerPolicy	ConfigMap/Helm	String	"binpack"	Default node-level scheduling policy: "binpack" (consolidate	[11]

				on fewer nodes) or "spread" (distribute across nodes).	
scheduler.defaultSchedulerPolicy.gpuSchedulerPolicy	ConfigMap/Helm	String	"spread"	Default GPU-level scheduling policy within a node: "binpack" (consolidate on fewer GPUs) or "spread" (distribute across GPUs).	[11]
scheduler.patch.enabled	Helm	Boolean	true	If true, Helm uses kube-webhook-certgen to generate TLS certificates for the webhook.	[11]

These parameters provide administrators with significant control over how GPU resources are virtualized and scheduled. The deviceMemoryScaling option, for instance, allows for memory oversubscription, a powerful but potentially risky feature if not managed carefully. The distinction between node-level and GPU-level scheduling policies (nodeSchedulerPolicy vs. gpuSchedulerPolicy) offers nuanced control over workload placement, catering to different optimization goals like maximizing utilization (binpack) or improving fault tolerance (spread). The ability to customize resource names offers compatibility with existing conventions or user preferences.

3.4. Requesting Shared GPU Resources in Pods

Users request HAMI-virtualized GPU resources through standard Kubernetes pod specifications, using specific resource names and annotations to define their requirements and influence scheduling.

Table 3.3: HAMI Pod Annotations and Resource Requests

Annotation/ Resource Key	Type	Purpose	Example Value(s)	Notes/Impact	Key Snippet(s)
nvidia.com/gpu	Resource Limit	Number of virtual GPUs (vGPUs) to allocate.	"1", "2"	For HAMI, this typically refers to vGPU count. The total physical memory/cores are further divided based on other requests.	[5]
nvidia.com/gpumem	Resource Limit	Amount of GPU memory per vGPU in MiB.	"3000" (for 3000MiB)	Enforces memory limit for the vGPU.	[9]
nvidia.com/gpumem-percentage	Resource Limit	Amount of GPU memory per vGPU as a percentage of physical GPU memory.	"50" (for 50%)	Alternative to absolute memory request. Cannot be used with nvidia.com/gpumem.	[11]
nvidia.com/gpucores	Resource Limit	Percentage of GPU compute cores per vGPU.	"30" (for 30%)	Controls compute share. 100 can mean exclusive core usage.	[11]
hami.io/node	Pod	Overrides	"binpack",	Allows	[7]

-scheduler-policy	Annotation	default node-level scheduling policy for this pod.	"spread"	per-pod customization of node selection strategy.	
hami.io/gpu-scheduler-policy	Pod Annotation	Overrides default GPU-level (within-node) scheduling policy for this pod.	"binpack", "spread"	Allows per-pod customization of GPU selection strategy on a node.	[7]
nvidia.com/use-gputype	Pod Annotation	Specifies a comma-separated list of allowed NVIDIA GPU models.	"Tesla V100-PCIE-32GB,NVIDIA A10"	Pod will only be scheduled on nodes with one of these GPU types.	[7]
nvidia.com/nouse-gputype	Pod Annotation	Specifies a comma-separated list of disallowed NVIDIA GPU models.	"Tesla K80"	Pod will not be scheduled on nodes with these GPU types.	[7]
nvidia.com/use-gpuuuid	Pod Annotation	Specifies a comma-separated list of allowed NVIDIA GPU UUIDs.	"GPU-AAA,GPU-BBB"	Pod must use one of the specified physical GPUs.	[7]
nvidia.com/nouse-gpuuuid	Pod Annotation	Specifies a comma-separated list of disallowed NVIDIA GPU UUIDs.	"GPU-CCC"	Pod will not use any of the specified physical GPUs.	[7]

nvidia.com/vgpu-mode	Pod Annotation	Instructs HAMI on the virtualization mode to use.	"mig", "hami-core"	Used for features like dynamic MIG, directing HAMI to provision a MIG instance or a standard HAMi-core vGPU.	11
cambricon.com/mlunum	Resource Limit	Number of Cambricon MLUs.	"1"	For requesting Cambricon devices.	62
cambricon.com/mlumem	Resource Limit	Cambricon MLU memory in MiB.	"10240"	For requesting specific memory on Cambricon MLUs.	62
hygon.com/dcunum	Resource Limit	Number of Hygon DCUs.	"1"	For requesting Hygon DCU devices.	64
hygon.com/dcumem	Resource Limit	Hygon DCU memory in MiB.	"2000"	For requesting specific memory on Hygon DCUs.	64
hygon.com/dcucores	Resource Limit	Hygon DCU cores.	"15"	For requesting specific core count on Hygon DCUs.	64

huawei.com/Ascend910A-memory (example)	Resource Limit	Huawei Ascend NPU memory (resource name may vary by model/template).	"2184"	Example for Ascend NPU, actual names depend on HAMI's Ascend plugin configuration and templates.	[66]

Example Pod Specifications:

- **NVIDIA GPU Sharing (HAMi-core):**

```yaml
YAML
apiVersion: v1
kind: Pod
metadata:
  name: hami-nvidia-sharing-example
  annotations:
    hami.io/gpu-scheduler-policy: "binpack" # Optional: override default GPU policy
spec:
  schedulerName: hami-scheduler # Or ensure webhook directs to HAMI logic
  containers:
  - name: cuda-app
    image: nvidia/cuda:11.8.0-base-ubuntu22.04 # Or any CUDA-enabled image
    resources:
      limits:
        nvidia.com/gpu: "1" # Number of vGPUs
        nvidia.com/gpumem: "3000" # Request 3000MiB GPU memory
        nvidia.com/gpucores: "30" # Request 30% of GPU core compute
    env:
    - name: GPU_CORE_UTILIZATION_POLICY
      value: "force" # or "default", "disable" [11]
```

- **NVIDIA Dynamic MIG:** [26]

```yaml
YAML
apiVersion: v1
kind: Pod
metadata:
  name: hami-dynamic-mig-example
  annotations:
```

```
    nvidia.com/vgpu-mode: "mig" # Instructs HAMI to use a MIG instance
spec:
  schedulerName: hami-scheduler
  containers:
  - name: mig-container
    image: nvidia/cuda:11.8.0-base-ubuntu22.04
    resources:
      limits:
        nvidia.com/gpu: "1" # Number of MIG instances
        nvidia.com/gpumem: "10000" # Request 10GB, HAMI will find suitable MIG profile
```

- **Cambricon MLU:** [5]

```
YAML
apiVersion: v1
kind: Pod
metadata:
  name: hami-cambricon-example
spec:
  schedulerName: hami-scheduler
  containers:
  - name: mlu-container
    image: cambricon-image # Replace with actual Cambricon image
    resources:
      limits:
        cambricon.com/mlunum: "1"
        cambricon.com/mlumem: "10240" # Request 10GB MLU memory
        # cambricon.com/mlu.smlu.vcore: "50" # If core sharing supported by specific HAMI version
for MLU
```

- **Hygon DCU:** [5]

```
YAML
apiVersion: v1
kind: Pod
metadata:
  name: hami-hygon-example
spec:
  schedulerName: hami-scheduler
  containers:
  - name: dcu-container
    image: image.sourcefind.cn:5000/dcu/admin/base/pytorch:2.1.0-centos7.6-dtk24.04-py310 #
Example image
    resources:
```

```
    limits:
      hygon.com/dcunum: "1"
      hygon.com/dcumem: "2000" # Request 2000MiB DCU memory
      hygon.com/dcucores: "15" # Request 15 DCU cores
```

- **Huawei Ascend NPU:** [5]

```
YAML
apiVersion: v1
kind: Pod
metadata:
  name: hami-ascend-example
  annotations:
    hami.io/use-Ascend910B-uuid: "device-uuid-1" # Optional: specify NPU UUID
spec:
  schedulerName: hami-scheduler
  containers:
  - name: ascend-container
    image: ascendhub.huawei.com/public-ascendhub/ascend-mindspore:23.0.RC3 # Example image
    resources:
      limits:
        # Actual resource names depend on HAMI's Ascend plugin configuration and NPU model templates.
        # Example based on 'vir02' template for Ascend910A from [66]:
        huawei.com/Ascend910A-memory: "2184"
        # huawei.com/Ascend910A-aicore: "2"
```

Important Notes from Documentation:
It is consistently advised to avoid using privileged: true in pod security contexts and to avoid setting spec.nodeName directly. Instead, nodeSelector or node affinity/anti-affinity rules should be used for targeting specific nodes.5 Using privileged: true can bypass HAMI's controls and expose all GPUs, while nodeName bypasses the scheduler.
The flexibility in requesting resources (absolute memory, percentage memory, core percentage) alongside the ability to influence scheduling via annotations provides a powerful toolkit for users. However, this also necessitates clear understanding and documentation to prevent misconfigurations. For instance, the dual meaning of nvidia.com/gpu (vGPU count for HAMI vs. physical GPU count in other contexts) could be a source of confusion if not properly clarified for users.[5] The introduction of distinct resource names for different hardware vendors (e.g., cambricon.com/mlunum, hygon.com/dcunum) is logical but requires users to be aware of the correct names for the specific hardware they are targeting.

3.5. Monitoring and Observability with HAMI

Effective monitoring is essential for understanding resource utilization and the behavior of shared GPU workloads. HAMI provides mechanisms for exposing metrics and integrating with common observability tools.[5]

- **Metrics Exposure:** HAMI automatically enables metrics exposure after installation. Cluster-wide scheduler metrics are available via an HTTP endpoint, typically http://{scheduler_ip}:{monitorPort}/metrics. The default monitorPort is 31993 and can be customized during Helm installation using devicePlugin.service.httpPort.[5] It's noted that vGPU status on a node is typically collected only after a vGPU has been actively used on that node.[17]
- **Prometheus and Grafana Integration:** HAMI's metrics are designed to be scraped by Prometheus. The official documentation and community resources often point to using NVIDIA DCGM Exporter in conjunction with HAMI's own metrics to provide a comprehensive view of both physical GPU health/performance and vGPU allocation.[22] A publicly available Grafana dashboard (ID: 22043 on grafana.com, titled "hami-vgpu-metrics-dashboard") is specifically designed for visualizing these combined metrics.[22]
- **HAMi-WebUI:** For a more direct visualization and management interface, the Project-HAMi organization provides HAMi-WebUI. This open-source tool offers an intuitive web interface to see GPU resource allocation and usage across nodes, with detailed views for tasks and individual GPUs.[24]
- **Specific Metrics:** While a complete, exhaustive list of all HAMI-specific metrics is not fully detailed in one place across the snippets, various pieces of information point to the types of metrics available:
    - **MIG Instance Metrics:** When using dynamic MIG, nodeGPUMigInstance (a gauge) provides details about active MIG instances on a node.[26]
    - **Volcano Integration Metrics:** When HAMI is integrated with the Volcano scheduler, Volcano exposes metrics like volcano_vgpu_device_allocated_cores and volcano_vgpu_device_allocated_memory via its own metrics endpoint ({volcano_scheduler_ip}:8080/metrics).[27] The volcano-vgpu-device-plugin itself also exposes metrics related to GPU utilization, memory usage, and pod-specific limits/usage on port 9394 of the device plugin pod.[28]
    - **General vGPU Allocation Metrics:** The HAMI scheduler itself tracks vGPU allocations, which form the basis of its metrics.
    - **Combined with DCGM:** For NVIDIA GPUs, DCGM provides a rich set of hardware-level metrics (utilization, memory, temperature, power) [1], which are essential for a complete picture and are leveraged by the recommended Grafana dashboard.
- **Potential Monitoring Challenges:** An open issue on GitHub mentioned problems with vGPUmonitor being unable to obtain metrics.[10] This highlights that, as with any

complex system, ensuring reliable and comprehensive metrics collection can sometimes present challenges that need to be addressed by the project.

The monitoring strategy of HAMI, relying on its own metrics for vGPU allocation and encouraging the use of standard tools like Prometheus/Grafana alongside vendor-specific exporters like DCGM, is a practical approach. It allows users to leverage familiar observability stacks. The HAMi-WebUI offers a more tailored experience for users focused specifically on HAMI's state. The key to effective monitoring is to combine HAMI's vGPU allocation data with actual physical GPU utilization and health metrics to get a true sense of efficiency and performance.

Table 3.4: Key HAMI Monitoring Metrics (Illustrative)

Metric Name (Conceptual or Actual)	Source Example	Description	Typical Use Case	Key Snippet(s)
nodeGPUMigInstance	HAMI Scheduler (for MIG)	Gauge indicating active MIG instances, their profiles, and associated physical device.	Monitoring dynamic MIG configurations and availability.	26
volcano_vgpu_device_allocated_cores	Volcano Scheduler (with HAMI integration)	Percentage of GPU compute cores allocated to vGPUs on a specific physical GPU.	Tracking vGPU core allocation by Volcano.	27
volcano_vgpu_device_allocated_memory	Volcano Scheduler (with HAMI integration)	Amount of GPU memory (e.g., in MiB) allocated to vGPUs on a specific physical GPU.	Tracking vGPU memory allocation by Volcano.	27
Pod GPU Memory	Volcano Device Plugin (with HAMI	Actual GPU memory used by a pod vs. its	Identifying memory overruns or	28

Usage/Limit	integration)	allocated vGPU memory limit.	underutilization at the pod level.	
Pod GPU Core Utilization	Volcano Device Plugin / DCGM via HAMI Dashboard	Actual GPU core utilization by a pod.	Assessing if a pod is effectively using its allocated vGPU compute resources.	[1]
Physical GPU Utilization	DCGM via HAMI Dashboard	Overall utilization of the physical GPU (compute, memory bandwidth, encoder/decoder).	Understanding the load on the physical hardware supporting the vGPUs.	[1]
Physical GPU Memory Used/Total	DCGM via HAMI Dashboard	Total memory used and available on the physical GPU.	Assessing overall memory pressure on the physical GPU.	[1]
HAMI Scheduler Metrics	HAMI Scheduler (/metrics endpoint)	General metrics about scheduler operations, vGPU allocations, node status from HAMI's perspective.	Monitoring the health and activity of the HAMI scheduling components.	[5]

4. HAMI within the Open-Source GPU Scheduling Ecosystem

HAMI does not operate in a vacuum; it is part of a larger ecosystem of tools and schedulers aimed at optimizing GPU usage in Kubernetes. Its ability to integrate with other schedulers and its comparative strengths and weaknesses define its role.

4.1. Integration with Kubernetes Schedulers

HAMI's architecture allows it to work alongside or in conjunction with other Kubernetes scheduling components, offering flexibility in how its GPU sharing capabilities are leveraged.

- **Koordinator:** Koordinator, a QoS-based scheduler for Kubernetes, explicitly integrates with HAMI to provide an end-to-end GPU sharing solution. Koordinator utilizes HAMi-core for its GPU isolation capabilities at the node level.[13] Pods signal their intent to use HAMi-managed shared GPUs by including the label koordinator.sh/gpu-isolation-provider: HAMi-core. Koordinator then handles the scheduling of these pods, taking into account the fractional GPU resources defined (e.g., koordinator.sh/gpu-core, koordinator.sh/gpu-memory-ratio), while HAMi-core enforces these limits within the container.[13] The Koordinator v1.6 release further enhanced this by improving device topology awareness with HAMI, supporting more GPU models, and enabling joint allocation of GPU and RDMA resources.[31] This collaboration allows users to benefit from Koordinator's advanced scheduling features (like load-aware scheduling and co-location) while using HAMI for the underlying GPU virtualization.
- **Volcano Scheduler:** Volcano, a batch scheduling system designed for high-performance computing and AI/ML workloads, also integrates with HAMI. Project-HAMi provides a dedicated volcano-vgpu-device-plugin that leverages HAMi-core to enable hard resource isolation and device sharing for NVIDIA GPUs scheduled by Volcano.[27] This integration requires Volcano version 1.9 or newer. To enable this, the Volcano scheduler's ConfigMap must be updated to include and enable the deviceshare plugin with deviceshare.VGPUEnable: true.[28] Pods then request vGPU resources using Volcano-specific annotations such as volcano.sh/vgpu-number, volcano.sh/vgpu-memory, and volcano.sh/vgpu-cores.[28] This setup allows organizations to use Volcano's gang scheduling and other batch-oriented features in conjunction with HAMI's fine-grained GPU sharing. Some adopters explicitly use HAMI with Volcano for managing automatic training pipelines.[34] Community discussions also reflect interest in using Volcano's scheduling capabilities for non-NVIDIA devices managed by HAMI, aiming to consolidate on a single advanced scheduler.[35]
- **NVIDIA KAI Scheduler:** As of early-mid 2025, there is no official, documented direct integration between HAMI and the NVIDIA KAI Scheduler. KAI is a comprehensive Kubernetes scheduler open-sourced by NVIDIA (formerly from Run:ai), designed for AI/ML workloads. It features its own mechanisms for GPU sharing (often described as time-slicing like, using a reservation pod system, and currently lacking hard isolation), along with advanced features like hierarchical queues, fairness policies, and batch scheduling.[36] While KAI aims to improve GPU utilization, its current sharing model does not provide the same level of resource isolation that HAMI targets with HAMi-core.[38] Community discussions indicate an interest in combining KAI's sophisticated scheduling and resource governance with HAMI's hard isolation capabilities, recognizing that HAMI already provides this layer for projects like Volcano and Koordinator.[39] HAMI's roadmap does include plans for

NVIDIA GPU Operator integration [9], which is a prerequisite for installing KAI Scheduler.[37] This future integration might open pathways for closer collaboration or interoperability, but currently, they represent distinct approaches to GPU sharing and scheduling.

- **Default Kubernetes Scheduler:** HAMI can also function with the default Kubernetes scheduler (kube-scheduler). Its architecture, including a scheduler extender and mutating webhook, allows it to influence scheduling decisions for pods requesting HAMI-managed vGPU resources.[5] The webhook can identify pods requiring vGPUs and ensure they are handled by HAMI's scheduling logic, which then filters and scores nodes based on vGPU availability and defined policies.

This ability for HAMI to provide its core GPU virtualization (HAMi-core) as a foundational layer for various established schedlers is a notable strategic aspect. It allows HAMI to focus on the complexities of device-level sharing and isolation while users can leverage the advanced scheduling algorithms and features of systems like Koordinator or Volcano. This modularity enhances HAMI's adoption potential within diverse Kubernetes ecosystems. However, the variation in resource request syntax (e.g., koordinator.sh/*, volcano.sh/*, nvidia.com/*) when using HAMI through different schedulers can introduce a layer of complexity for users operating in environments with multiple scheduling solutions.

4.2. Comparative Analysis

HAMI's approach to GPU sharing and scheduling can be better understood when compared to native NVIDIA solutions and other open-source alternatives.

Table 4.1: Comparison of GPU Sharing Techniques: HAMI vs. Native NVIDIA

Feature	HAMI (via HAMi-core)	NVIDIA MIG (Multi-Instance GPU)	NVIDIA Time-Slicing	NVIDIA MPS (Multi-Process Service)
Sharing Granularity	Fine-grained (Memory in MB/%, Cores in %)	Fixed, predefined hardware partitions (e.g., 1g.5gb)	GPU time shared among multiple containers	Multiple processes share a single GPU context
Memory Isolation	Aims for hard isolation (via API	Strong (Hardware-level)	None (shared memory space)	None (shared memory space)

	interception)			
Compute Isolation	Soft isolation (percentage-based core control)	Strong (Hardware-level)	None (compete for compute cycles)	Limited (kernels can interleave, shared compute)
Fault Isolation	Limited (software-level; driver/HW faults can impact)	Strong (Hardware-level)	None (a crashing pod can affect others)	Poor (MPS daemon failure affects all clients)
Hardware Support	Broad (NVIDIA, other vendors planned/supported)	Newer NVIDIA GPUs (A100, H100, etc.) [43]	Most NVIDIA GPUs supporting device plugin extensions	NVIDIA GPUs (Volta and later) [19]
Performance Overhead	Potential from API interception, context switching [18]	Minimal (hardware partitioning)	Context switching overhead [19]	Lower context switching than time-slicing for some loads [19]
Ease of Use/Config	"Zero code changes"; Helm install; ConfigMap/annotations	Can be complex to configure/manage MIG profiles [43]	Simpler ConfigMap configuration via device plugin	Simpler setup, managed by device plugin
Key Snippet(s)	9	69	4	59

Table 4.2: HAMI vs. Selected Open-Source GPU Schedulers/Sharing Solutions

Feature/Aspect	**HAMI**	**NVIDIA KAI Scheduler**	**TensorFusion**	**Aliyun gpushare-scheduler**
Primary Goal	Fine-grained GPU sharing, isolation,	Advanced AI/ML workload scheduling,	Fractional GPU, GPU pooling, remote GPU,	GPU memory sharing for

	heterogeneous device mgmt.	fairness, GPU sharing (soft isolation).	VRAM expansion, advanced autoscaling.	NVIDIA GPUs.
Sharing Mechanism	User-space API interception (HAMi-core), vGPU concept.	Fractional requests via annotations, reservation pod, time-slicing-like.	GPU virtualization & remoting, TFlops-based fractional GPU.	Scheduler extender, device plugin for memory sharing.
Isolation Strength	Hard memory (claimed), soft compute.	Soft/None for memory & compute in its sharing mechanism.	Claims isolation (details of mechanism vary).	Primarily memory accounting, limited isolation.
Heterogeneous Support	Yes (NVIDIA, Cambricon, Hygon, Ascend, etc. planned/supported).	Primarily NVIDIA (supports DRA for other vendors if drivers exist).	NVIDIA (AMD WIP).	NVIDIA only.
Scheduler Integration	Own scheduler logic (extender), integrates with Koordinator, Volcano.	Full standalone scheduler, runs alongside default.	Webhook-only (no device/scheduler plugin needed).	Scheduler extender for default K8s scheduler.
Key Differentiators	HAMi-core virtualization, broad vendor support, CNCF Sandbox.	Hierarchical queues, DRF fairness, batch scheduling, Run:ai heritage.	Rich enterprise features (VRAM expansion, remote GPU), simpler deployment arch.	Early open-source GPU sharing solution.
Community/Maturity	CNCF Sandbox, active development, growing	Recently open-sourced by NVIDIA, backed by	Commercial offering with open-source components,	Mature, but perhaps less active development

	adoption (esp. in Asia).	NVIDIA/Run:ai.	newer.	than newer solutions.
Key Snippet(s)	5	36	71	72

This comparative analysis reveals that the Kubernetes GPU sharing and scheduling landscape is diverse. HAMI carves out a distinct position by focusing on software-based, fine-grained virtualization with an emphasis on strong memory isolation and broad heterogeneous hardware support. While NVIDIA's native solutions like MIG offer robust hardware isolation, they can be less flexible. Other open-source projects like KAI Scheduler prioritize advanced scheduling policies and fairness for AI workloads, with a simpler sharing model that currently lacks HAMI's isolation strengths. TensorFusion presents itself as a more feature-rich platform with a different architectural approach to virtualization.

The choice of solution is therefore not straightforward and depends heavily on specific requirements:

- For strong, hardware-enforced isolation on compatible NVIDIA hardware, **MIG** is the standard.
- For basic sharing on NVIDIA GPUs where isolation is less critical, **Time-Slicing** or **MPS** might suffice.
- For sophisticated AI/ML scheduling with queueing and fairness on NVIDIA hardware, **KAI Scheduler** is a strong contender, though its current GPU sharing lacks hard isolation.
- For fine-grained, software-enforced memory isolation and broad support for heterogeneous hardware (including non-NVIDIA), **HAMI** offers a compelling open-source option.
- For a platform with features like remote GPU sharing or VRAM expansion, **TensorFusion** is an alternative.

HAMI's user-space API interception is a key technological choice. While it enables flexibility and broad hardware compatibility without kernel modifications, it inherently differs from hardware-level partitioning (like MIG) in terms of the "hardness" of isolation and potential performance characteristics. For environments demanding the utmost security or predictable, low-latency performance for co-located tenants, this distinction is vital.

5. Gaining Insights: How HAMI Enhances GPU Resource Management

HAMI's features and architecture translate into tangible benefits for managing GPU resources in Kubernetes, particularly in scenarios demanding high utilization, multi-tenancy, and support for diverse hardware.

5.1. Achieving Fine-Grained Resource Control and Improved Utilization

A primary driver for adopting solutions like HAMI is the need to move beyond whole-GPU allocation. HAMI allows users to request GPU resources with a high degree of granularity:

- **GPU Memory:** Can be specified in absolute terms (e.g., megabytes) or as a percentage of the physical GPU's memory.[5]
- **GPU Compute Cores:** Can be requested as a percentage of the GPU's total compute capability.[9]

This fine-grained control means that multiple workloads, each potentially requiring only a fraction of a GPU's total capacity, can be co-located on a single physical GPU. This directly addresses the common problem of underutilization where, for example, an inference task might only use 10-20% of a powerful GPU if allocated the entire device.[8] By enabling such sharing, HAMI helps organizations maximize the return on their expensive GPU investments. Real-world use cases have reported significant improvements, with GPU utilization increasing from below 20% to over 60% in some instances.[14] This efficient packing of workloads is crucial for cost optimization, especially as AI/ML computational demands continue to grow.

5.2. Enabling Multi-Tenancy and Diverse Workload Co-existence

HAMI's resource isolation capabilities, particularly its claim of hard memory isolation for NVIDIA GPUs [8], are designed to support multi-tenant environments. Different users, teams, or applications can share physical GPUs with greater confidence that one workload will not unduly interfere with another's memory space. This is vital for:

- **R&D Platforms:** Where multiple researchers or developers need simultaneous access to limited GPU resources for experimentation (e.g., Jupyter notebooks).[14]
- **Educational Settings:** Providing fractional GPU access to many students from a smaller pool of physical GPUs.[9]
- **Cloud Providers:** Offering more affordable, fractional GPU instances to customers, thereby increasing the monetization potential of each physical GPU.[14]
- **Mixed Workload Environments:** Co-locating different types of AI/ML tasks, such as model training and inference serving, on the same hardware. For example, SNOW (Korea) successfully used HAMI to run training and inference workloads concurrently on NVIDIA A100 GPUs, a scenario where MIG's fixed partitions were too rigid and MPS lacked sufficient isolation.[14]

The ability to assign different GPU tiers or resource profiles to different user groups, coupled with features like idle GPU reclamation based on container-level metrics (as reported in some use cases [14]), further enhances HAMI's utility in shared environments.

5.3. Facilitating Management of Diverse Hardware Accelerators

One of HAMI's strategic aims is to provide a unified management layer for heterogeneous AI accelerators from various vendors.[5] As organizations increasingly deploy a mix of hardware (e.g., NVIDIA GPUs alongside specialized NPUs from Cambricon, Huawei Ascend, or Hygon DCUs), the complexity of managing these diverse resources grows. HAMI seeks to abstract some of this complexity by offering:

- Support for multiple device types through a common architectural framework (device plugins, scheduler logic).
- The goal of a consistent way to request these resources in pod specifications, even if the underlying resource names differ per vendor.
- Unified scheduling policies that can apply across different types of accelerators.

A reported use case involved a major bank using HAMI to manage a combination of domestic (non-NVIDIA) GPUs and NVIDIA GPUs under a single scheduling system.[14] This capability simplifies operations and allows for more flexible workload placement in environments with a diverse hardware portfolio. However, it is important to note that the level of feature support (e.g., compute isolation, dynamic sharing) may vary across different supported device types, as indicated in HAMI's documentation.[7]

5.4. Real-World Use Cases and Reported Benefits

The practical impact of HAMI is best illustrated by its adoption in various real-world scenarios:

- **Banking Sector (Dynamic Inference):** A major bank running numerous lightweight inference tasks with cyclical peak/off-peak demand patterns experienced low GPU utilization (<20%) when dedicating full GPUs per task. By implementing HAMI and enabling features like memory oversubscription and priority-based preemption, they reportedly increased average GPU utilization to over 60%, while still meeting service level agreements (SLAs). HAMI also facilitated the management of a mixed fleet of domestic and NVIDIA GPUs.[14] This demonstrates HAMI's potential for significant cost savings and operational efficiency in environments with bursty, low-footprint GPU workloads.
- **R&D (Securities & Autonomous Driving):** In research environments characterized by many users and relatively few GPUs (e.g., internal Kubeflow platforms where Jupyter Notebooks would occupy full GPUs even if idle), HAMI's virtual GPU support, card-type-based scheduling, and container-level monitoring enabled more effective sharing. Idle GPUs could be automatically reclaimed based

on real-time usage metrics, leading to improved overall resource utilization.[14] This highlights HAMI's value in democratizing access to scarce GPU resources.

- **GPU Cloud Provider (Monetizing GPU Slices):** A cloud vendor leveraged HAMI to transition from offering whole-card GPU instances (e.g., an H800 at $2/hr) to providing more granular, fractional GPU offerings (e.g., 3GB slices at $0.26/hr). This strategy drastically improved affordability for end-users and reportedly tripled the revenue per physical card for the provider, by allowing up to 26 concurrent users on a single H800 GPU.[14] This case underscores the economic benefits HAMI can unlock by enabling new service models.
- **SNOW (Korea - AI Workload Migration to Kubernetes):** When migrating AI workloads to Kubernetes, SNOW faced the challenge of co-locating training and inference tasks on the same NVIDIA A100 GPUs. They found NVIDIA MIG too inflexible due to its fixed partitions, NVIDIA MPS lacking necessary memory isolation, and full Kubeflow too heavyweight for their needs. HAMI provided a solution that allowed them to share full GPUs safely among different tasks without requiring application code changes, facilitating a smoother infrastructure migration for hundreds of A100 GPUs.[14] This illustrates HAMI's role as a practical enabler for complex workload consolidation.

These use cases collectively suggest that HAMI's value lies in its ability to improve GPU utilization, enable cost-effective multi-tenancy, and simplify the management of diverse workloads on shared GPU infrastructure. The "zero code changes" aspect is frequently highlighted as a key enabler for adoption, as it minimizes the friction of migrating existing AI/ML applications.[5]

5.5. Performance Considerations

While HAMI offers significant benefits in terms of resource sharing and utilization, it is essential to consider the potential performance implications of its user-space API interception mechanism.

- **Overhead of API Interception:** HAMi-core operates by hijacking CUDA API calls in user space.[14] This software layer, while enabling flexibility, can introduce latency or computational overhead compared to direct hardware access or kernel-level virtualization techniques.[8] The magnitude of this overhead is workload-dependent and influenced by the frequency and type of intercepted API calls. For compute-intensive tasks with infrequent API interactions, the overhead might be negligible. However, for workloads with many fine-grained CUDA calls, the cumulative impact could be more noticeable.
- **Context Switching Costs:** GPU context switching is inherently more expensive than CPU context switching due to the larger state that needs to be saved and restored.[19] Sharing mechanisms that lead to frequent context switches between

virtualized GPU instances can impact overall performance. HAMI's compute limiting, which may involve regulating kernel launches, could incur such costs, similar to time-slicing approaches.
- **Benchmarking Data:** Specific, comprehensive, and independent benchmarks comparing HAMI's performance against other sharing solutions (MIG, MPS, time-slicing) or whole-GPU allocation are not extensively detailed within the provided research snippets. While general GPU sharing benchmarks exist [44], and academic reviews discuss the complexities of GPU virtualization performance analysis [45], dedicated HAMI performance studies are less visible in this material. The RiseUnion blog on HAMI mentions fine-grained allocation and isolation [8] but does not quantify the performance overhead. This lack of readily available, detailed performance benchmarks for HAMI across various workloads and hardware types represents an area where more public data would be beneficial for potential adopters.

The "soft" nature of HAMI's compute limiting (based on polling and token accounting [15]) might also lead to performance variability for highly sensitive applications, as it aims to control average utilization rather than providing instantaneous, hard caps on compute cycles. This is a common characteristic of software-based sharing mechanisms. Therefore, users considering HAMI for performance-critical applications should conduct their own thorough testing and benchmarking to validate its suitability for their specific workloads and performance SLAs.

6. The Future Trajectory: HAMI's Roadmap and GPU Scheduling Trends

The landscape of GPU resource management in Kubernetes is continuously evolving. HAMI's development roadmap and the broader trends in the ecosystem indicate a move towards more sophisticated, efficient, and standardized solutions.

6.1. HAMI's Official and Discussed Development Plans

HAMI, as a CNCF Sandbox project, has a publicly visible direction, with several key areas targeted for future development, reflecting its ambition to become a more comprehensive and versatile solution:

- **Broader Hardware Support:** A core tenet of HAMI is its heterogeneous nature. Explicit plans include adding support for **Intel GPU devices** and **AMD GPU devices**.[7] This expansion is critical for users with mixed-vendor environments and aligns with HAMI's goal of providing a unified management interface.
- **Dynamic Resource Allocation (DRA) Support:** HAMI plans to integrate with Kubernetes' **Dynamic Resource Allocation (DRA)** framework.[7] DRA is a significant Kubernetes enhancement designed to provide a more flexible and

extensible way for workloads to request and consume specialized hardware resources beyond the traditional device plugin model.[2] Adopting DRA would align HAMI with the future direction of Kubernetes resource management.

- **NUMA Affinity Scheduling:** The roadmap includes support for more flexible scheduling policies, specifically mentioning **NUMA (Non-Uniform Memory Access) affinity**.[7] This is crucial for optimizing performance on multi-socket servers with multiple GPUs, as ensuring that pods are scheduled on NUMA nodes with local access to their assigned GPUs can significantly reduce memory access latency.[48] This was also highlighted as a topic in a KubeCon presentation involving HAMI contributors.[50]
- **NVIDIA GPU Operator Integration:** Integration with the **NVIDIA GPU Operator** is planned.[7] The NVIDIA GPU Operator automates the deployment and lifecycle management of NVIDIA drivers and related software components.[1] This integration could simplify the setup of NVIDIA prerequisites for HAMI users.
- **Richer Observability Capabilities:** Continuous improvements in monitoring and observability are a stated goal [7], likely involving more detailed metrics and easier integration with monitoring stacks.
- **Video Encoding/Decoding Processing Support:** Targeting support for specific workload types like video encoding/decoding indicates an effort to cater to a broader range of GPU-accelerated applications.[7]
- **NVLink Topology-Aware Scheduling:** A maintainer comment on Reddit mentioned **NVLink topology-aware scheduling** as a work in progress.[14] This would allow HAMI to make more intelligent scheduling decisions based on the high-bandwidth interconnects between NVIDIA GPUs.
- **General Roadmap Maintenance and Standardization:** The project acknowledges the ongoing need for regular roadmap updates and adherence to versioning standards like SemVer v2 [52], which is important for project maturity and user trust.

These roadmap items suggest HAMI is aiming to enhance its core virtualization capabilities, broaden its hardware compatibility, align with Kubernetes advancements like DRA, and improve operational aspects like installation and observability. The successful execution of this roadmap will depend on sustained community involvement and development effort.

6.2. Emerging Trends in Kubernetes GPU Management

HAMI's development occurs within a dynamic field. Several broader trends are shaping how GPUs and other specialized hardware are managed in Kubernetes:

- **Dominance of Dynamic Resource Allocation (DRA):** DRA is emerging as the next-generation framework in Kubernetes for handling specialized hardware. Driven by Google and the wider community, DRA provides a more extensible and

standardized API for resource discovery, advertisement, and allocation compared to the original device plugin framework.[2] This allows for more sophisticated resource claims and lifecycle management, crucial for complex devices like GPUs. HAMI's plan to support DRA is a strategic alignment with this pivotal trend.

- **Rise of AI-Specific Schedulers:** The unique demands of AI/ML workloads (e.g., gang scheduling for distributed training, batch processing, awareness of data locality and accelerator topology) are driving the development of specialized schedulers.[2] Solutions like NVIDIA's KAI Scheduler [36] and Volcano [56] exemplify this, offering features beyond what the default Kubernetes scheduler provides. This trend underscores the need for schedulers that are not just resource brokers but also understand the semantics of AI/ML jobs.
- **Push for Standardization and Vendor Neutrality:** While hardware vendors like NVIDIA provide powerful, integrated toolchains (e.g., NVIDIA GPU Operator, CUDA, various device plugins) [1], there is a persistent community desire for more standardized and vendor-neutral approaches to resource management. DRA contributes to this by offering a common API framework. Projects like HAMI, with their explicit goal of supporting multiple hardware vendors, also reflect this trend.
- **Intensified Focus on Efficiency and Cost Optimization:** The high cost of GPUs and the massive computational requirements of modern AI models (especially LLMs) make resource efficiency and cost optimization paramount.[60] This fuels the demand for solutions that enable GPU sharing (like HAMI, KAI's sharing features, MIG), improve utilization, and provide better cost visibility and control.
- **Increasing Importance of NUMA and Topology Awareness:** For performance-critical applications, especially distributed training across multiple GPUs or nodes, the physical topology of the system (NUMA nodes, CPU-GPU affinity, inter-GPU connectivity like NVLink) significantly impacts performance.[48] Schedulers and resource managers are increasingly expected to be topology-aware to make optimal placement decisions.

HAMI's roadmap aligns well with several of these trends, particularly DRA support, NUMA awareness, and broader hardware support. Its integration with the NVIDIA GPU Operator, while potentially simplifying setup for NVIDIA users, will need careful implementation to maintain its vendor-neutral appeal. The evolution of Kubernetes itself, with DRA becoming more mainstream, might also influence HAMI's long-term architecture. It's conceivable that as Kubernetes and specialized schedulers become more adept at handling complex resource requests and scheduling paradigms, HAMI's unique value will increasingly center on its HAMi-core virtualization engine, serving as a pluggable isolation and sharing layer for various DRA-compliant resource drivers or advanced schedulers.

7. Conclusion and Strategic Recommendations

HAMI (Heterogeneous AI Computing Virtualization Middleware) emerges as a significant open-source contribution to the Kubernetes ecosystem, addressing the critical need for more efficient and flexible management of GPUs and other AI accelerators. Its core value proposition lies in enabling fine-grained resource sharing through a user-space virtualization technology, HAMi-core, which aims to provide strong memory isolation and configurable compute sharing for a variety of hardware.

Key Strengths of HAMI:

- **Fine-Grained Resource Allocation:** Allows requests for GPU memory by specific amounts (MB) or ratios (%), and GPU compute cores by percentage, facilitating much higher utilization than whole-GPU allocation.
- **Resource Isolation:** Claims hard memory isolation (via API interception for NVIDIA GPUs) and provides mechanisms for compute limiting, crucial for multi-tenant environments.
- **Heterogeneous Device Support:** Actively supports and plans to expand support for a wide range of AI accelerators beyond NVIDIA, including devices from Cambricon, Hygon, Huawei Ascend, and with future plans for Intel and AMD GPUs. This is a key differentiator.
- **Flexibility:** Offers various scheduling policies (binpack, spread at node and GPU levels), dynamic MIG support for NVIDIA GPUs, and integration capabilities with other Kubernetes schedulers like Koordinator and Volcano.
- **Open Source and Community:** As a CNCF Sandbox project, HAMI benefits from open governance and community contributions, fostering transparency and broader adoption.
- **Improved GPU Utilization and Cost Savings:** Real-world use cases demonstrate HAMI's potential to significantly increase GPU utilization (e.g., from <20% to >60%) and enable new, cost-effective service models like fractional GPU offerings.

Current Limitations and Considerations:

- **Performance Overhead:** The user-space API interception mechanism (HAMi-core) may introduce performance overhead compared to hardware-level virtualization or direct device access. Thorough benchmarking for specific workloads is advisable.
- **Isolation Robustness:** While HAMI aims for strong isolation, software-based methods are inherently different from hardware-enforced isolation (like NVIDIA MIG) and may have different security and fault isolation characteristics.
- **Driver and API Compatibility:** Maintaining HAMi-core's compatibility with rapidly evolving vendor drivers and APIs (especially CUDA) is an ongoing challenge for any software-based virtualization layer.
- **Feature Parity Across Devices:** The maturity and feature completeness (e.g., compute isolation, dynamic sharing) can vary for different supported non-NVIDIA

devices.
- **Documentation and Global Engagement:** As acknowledged by maintainers, continuous improvement in documentation and broader international community engagement are areas for development.
- **Configuration Complexity:** The extensive range of configuration options and potentially varying resource naming conventions across different integrations can present a learning curve.

When HAMI is a Suitable Solution:
HAMI is particularly well-suited for:

- Environments with a **diverse portfolio of AI accelerators** from multiple vendors, where a unified management approach is desired.
- Use cases requiring **more granular GPU sharing** than what NVIDIA MIG offers, or where MIG is unavailable (e.g., on older or non-MIG-capable cards) or its fixed partitions are too restrictive.
- **Development, testing, and educational environments** where maximizing the utilization of a limited pool of GPUs is a primary concern.
- **Production inference workloads** that are numerous, individually lightweight, and can benefit from sharing a physical GPU among multiple instances, provided the isolation and performance characteristics meet requirements.
- Organizations that are **comfortable adopting and potentially contributing to open-source solutions** and are prepared to manage the operational aspects of such a system.

HAMI's Contribution to "Vast Insight" into GPU Scheduling:
HAMI offers valuable insights into the GPU scheduling landscape in several ways:

1. It demonstrates a **practical, alternative approach (user-space API interception) to GPU virtualization and sharing**. This provides a deeper understanding of the complexities, trade-offs (flexibility vs. overhead, software vs. hardware isolation), and possibilities beyond native hardware partitioning or basic time-slicing.
2. Its existence and adoption underscore the **significant real-world demand for more flexible and efficient GPU resource management** than what is often provided by standard Kubernetes or default vendor tools. The documented use cases highlight tangible economic and operational benefits.
3. By actively pursuing **heterogeneous device support**, HAMI provides a unique lens into the challenges and potential solutions for unifying the management of diverse AI accelerators under the Kubernetes umbrella, moving beyond a single-vendor ecosystem.
4. Its **integrations with advanced schedulers like Koordinator and Volcano** showcase how specialized device virtualization can complement and enhance sophisticated scheduling systems, creating powerful combined solutions.

Final Thoughts:
The field of GPU management in Kubernetes is dynamic and rapidly evolving. Trends point towards increased standardization (e.g., Dynamic Resource Allocation - DRA), greater scheduler intelligence for AI/ML workloads, and a relentless pursuit of efficiency and broader hardware compatibility. HAMI is a noteworthy and impactful project within this landscape, pushing the boundaries of software-based GPU virtualization and sharing. Its commitment to open source, heterogeneous support, and fine-grained control makes it a valuable tool for many organizations.
Future success for HAMI will likely depend on continued robust development, active community engagement, clear and comprehensive documentation, and transparent demonstration of its performance and isolation characteristics across an expanding range of hardware and demanding workloads. As Kubernetes and its ecosystem mature, HAMI's core virtualization technology (HAMi-core) may become an even more critical building block, potentially integrating with DRA-compliant resource drivers and next-generation schedulers to deliver the flexible, efficient, and multi-vendor GPU management that modern AI infrastructure demands.

Works cited

1. Accelerate AI Workloads on Kubernetes | Portworx, accessed May 30, 2025, https://portworx.com/knowledge-hub/kubernetes-ai/
2. Transforming Kubernetes and GKE into the leading platform for AI/ML, accessed May 30, 2025, https://opensource.googleblog.com/2025/05/transforming-kubernetes-and-gke-into-the-leading-platform-for-aiml.html
3. Dynamic GPU-Aware Scheduling for Distributed Data Science Workloads in Kubernetes - EA Journals, accessed May 30, 2025, https://eajournals.org/ejcsit/wp-content/uploads/sites/21/2025/05/Dynamic-GPU.pdf
4. Troubleshooting GPU Scheduling Issues in Amazon EKS, accessed May 30, 2025, https://www.cloudkeeper.com/insights/blog/troubleshooting-gpu-scheduling-issues-amazon-eks
5. Project-HAMi/HAMi: Heterogeneous AI Computing Virtualization Middleware - GitHub, accessed May 30, 2025, https://github.com/Project-HAMi/HAMi
6. hami | CNCF, accessed May 30, 2025, https://www.cncf.io/projects/hami/
7. HAMi - DaoCloud Enterprise, accessed May 30, 2025, https://docs.daocloud.io/en/community/hami/
8. How HAMi(GPU Virtualization Technology) Can Save You Money! - RiseUnion, accessed May 30, 2025, https://www.theriseunion.com/blog/HAMi-vgpu-intro.html
9. HAMi - DaoCloud Enterprise, accessed May 30, 2025, https://docs.daocloud.io/en/community/hami
10. HAMi/README.md at master · Project-HAMi/HAMi · GitHub, accessed May 30, 2025, https://github.com/Project-HAMi/HAMi/blob/master/README.md
11. HAMi/docs/config.md at master · Project-HAMi/HAMi · GitHub, accessed May 30,

2025, https://github.com/Project-HAMi/HAMi/blob/master/docs/config.md
12. HAMi Source Code Analysis: Device Management and Scheduling - RiseUnion, accessed May 30, 2025, https://www.theriseunion.com/blog/HAMi-code-analyst-1.html
13. Device Scheduling - GPU Share With HAMi | Koordinator, accessed May 30, 2025, https://koordinator.sh/docs/next/user-manuals/device-scheduling-gpu-share-with-hami/
14. [Seeking Advice] CNCF Sandbox project HAMi – Why aren't more global users adopting our open-source fine-grained GPU sharing solution? : r/kubernetes - Reddit, accessed May 30, 2025, https://www.reddit.com/r/kubernetes/comments/1kvy06i/seeking_advice_cncf_sandbox_project_hami_why/
15. [Seeking Advice] CNCF Sandbox project HAMi – Why aren't more global users adopting our open-source fine-grained GPU sharing solution? : r/mlops - Reddit, accessed May 30, 2025, https://www.reddit.com/r/mlops/comments/1kvy17n/seeking_advice_cncf_sandbox_project_hami_why/
16. HAMi-core design | HAMi, accessed May 30, 2025, https://project-hami.io/docs/developers/hami-core-design/
17. HAMi: Open Source GPU Virtualization for AI Computing - RiseUnion, accessed May 30, 2025, https://www.theriseunion.com/blog/Project-HAMi.html
18. How does GPU virtualization impact performance in HPC workloads? - Massed Compute, accessed May 30, 2025, https://massedcompute.com/faq-answers/?question=How%20does%20GPU%20virtualization%20impact%20performance%20in%20HPC%20workloads?
19. How to Efficiently Share GPU Resources? - ZStack, accessed May 30, 2025, https://www.zstack-cloud.com/blog/how-to-efficiently-share-gpu-resources/
20. chaunceyjiang/k8s-vgpu-scheduler: OpenAIOS vGPU scheduler for Kubernetes is originated from the OpenAIOS project to virtualize GPU device memory. - GitHub, accessed May 30, 2025, https://github.com/chaunceyjiang/k8s-vgpu-scheduler
21. HAMi Configuration Guide: GPU Resource Pool Management - RiseUnion, accessed May 30, 2025, https://www.theriseunion.com/blog/HAMi-code-analyst-how-to-start.html
22. HAMi/docs/dashboard.md at master - GitHub, accessed May 30, 2025, https://github.com/Project-HAMi/HAMi/blob/master/docs/dashboard.md
23. hami-vgpu-metrics-dashboard | Grafana Labs, accessed May 30, 2025, https://grafana.com/grafana/dashboards/22043-hami-vgpu-metrics-dashboard/
24. Project-HAMi/HAMi-WebUI - GitHub, accessed May 30, 2025, https://github.com/Project-HAMi/HAMi-WebUI
25. Project-HAMi - GitHub, accessed May 30, 2025, https://github.com/Project-HAMi
26. HAMi/docs/dynamic-mig-support.md at master · Project-HAMi/HAMi ..., accessed May 30, 2025, https://github.com/Project-HAMi/HAMi/blob/master/docs/dynamic-mig-support.md
27. Monitor volcano-vgpu | HAMi, accessed May 30, 2025, https://project-hami.io/docs/v1.3.0/userguide/volcano-vgpu/nvidia%20gpu/monitor/

28. README.md - Project-HAMi/volcano-vgpu-device-plugin - GitHub, accessed May 30, 2025, https://github.com/Project-HAMi/volcano-vgpu-device-plugin/blob/main/README.md
29. The real-world guide to the NVIDIA GPU Operator for K8s AI - Spectro Cloud, accessed May 30, 2025, https://www.spectrocloud.com/blog/the-real-world-guide-to-the-nvidia-gpu-operator-for-kubernetes-ai
30. Support for scheduling GPU workloads (e.g., for Ascend devices) based on actual memory usage on the device · Issue #1081 · Project-HAMi/HAMi - GitHub, accessed May 30, 2025, https://github.com/Project-HAMi/HAMi/issues/1081
31. Koordinator v1.6: Supports Heterogeneous Resource Scheduling in AI/ML Scenarios, accessed May 30, 2025, https://www.alibabacloud.com/blog/koordinator-v1-6-supports-heterogeneous-resource-scheduling-in-aiml-scenarios_602051
32. Blog | Koordinator, accessed May 30, 2025, https://koordinator.sh/blog/
33. How to use volcano vgpu | HAMi, accessed May 30, 2025, https://project-hami.io/docs/userguide/volcano-vgpu/NVIDIA%20GPU/how-to-use-volcano-vgpu
34. HAMi Adopters, accessed May 30, 2025, https://project-hami.io/adopters/
35. volcano schedule · Issue #453 · Project-HAMi/HAMi - GitHub, accessed May 30, 2025, https://github.com/Project-HAMi/HAMi/issues/453
36. KAI Scheduler: NVIDIA open-sources Kubernetes GPU scheduler - Developer Tech News, accessed May 30, 2025, https://www.developer-tech.com/news/kai-scheduler-nvidia-open-sources-kubernetes-gpu-scheduler/
37. NVIDIA/KAI-Scheduler: KAI Scheduler is an open source Kubernetes Native scheduler for AI workloads at large scale - GitHub, accessed May 30, 2025, https://github.com/NVIDIA/KAI-Scheduler
38. Exploring GPU Sharing in Kubernetes with NVIDIA KAI Scheduler and Exostellar SDG, accessed May 30, 2025, https://exostellar.io/2025/04/08/gpu-sharing-in-kubernetes-nvidia-kai-vs-exostellar-sdg/
39. Deep Dive: How KAI-Scheduler Enables GPU Sharing on Kubernetes (Reservation Pod Mechanism & Soft Isolation) - Reddit, accessed May 30, 2025, https://www.reddit.com/r/kubernetes/comments/1jsx0n3/deep_dive_how_kaischeduler_enables_gpu_sharing_on/
40. NVIDIA KAI Scheduler: Optimize GPU Usage in ZenML Pipelines, accessed May 30, 2025, https://www.zenml.io/blog/nvidia-kai-scheduler-optimize-gpu-usage-in-zenml-pipelines
41. Trending Go repositories on GitHub today, accessed May 30, 2025, https://github.com/trending/go
42. GPU Sharing in Kubernetes: Nvidia Kai vs. Exostellar SDG - Hacker News, accessed May 30, 2025, https://news.ycombinator.com/item?id=43622647
43. Boost GPU efficiency in Kubernetes with NVIDIA Multi-Instance GPU ..., accessed

May 30, 2025, https://developers.redhat.com/articles/2025/05/27/boost-gpu-efficiency-kubernetes-nvidia-mig
44. Benchmarking GPU sharing strategies in Kubernetes - Weblog - Ronan Quigley, accessed May 30, 2025, https://ronanquigley.com/blog/benchmarking-gpu-sharing-strategies-in-kubernetes/
45. Analyzing GPU Performance in Virtualized Environments: A Case Study - MDPI, accessed May 30, 2025, https://www.mdpi.com/1999-5903/16/3/72
46. Kubernetes v1.33: An Insider Perspective - Komodor, accessed May 30, 2025, https://komodor.com/blog/kubernetes-v133-an-insiders-perspective/
47. Dynamic Resource Allocation for better device usage efficiency, accessed May 30, 2025, https://srujanpakanati.com/dynamic-resource-allocation-for-better-device-usage-efficiency
48. Container Service for Kubernetes:Enable NUMA topology-aware scheduling - Alibaba Cloud, accessed May 30, 2025, http://www.alibabacloud.com/help/en/ack/ack-managed-and-ack-dedicated/user-guide/enable-numa-topology-aware-scheduling
49. Enable Metax GPU topology-aware scheduling - HAMi, accessed May 30, 2025, https://project-hami.io/docs/userguide/metax-device/enable-metax-gpu-schedule/
50. Unlocking Heterogeneous AI Infrastructure K8s Cluster: Leveraging the...- Xiao Zhang & Mengxuan Li - YouTube, accessed May 30, 2025, https://www.youtube.com/watch?v=kcGXnp_QShs
51. Introduction — NVIDIA AI Enterprise: OpenShift on VMware vSphere ..., accessed May 30, 2025, https://docs.nvidia.com/ai-enterprise/deployment/openshift-on-vmware/latest/introduction.html
52. HAMi SANDBOX PROJECT ONBOARDING task list · Issue #476, accessed May 30, 2025, https://github.com/Project-HAMi/HAMi/issues/476
53. Incubating Preperations · Issue #1057 · Project-HAMi/HAMi - GitHub, accessed May 30, 2025, https://github.com/Project-HAMi/HAMi/issues/1057
54. NVIDIA Open Sources Run:ai Scheduler to Foster Community Collaboration, accessed May 30, 2025, https://developer.nvidia.com/blog/nvidia-open-sources-runai-scheduler-to-foster-community-collaboration/
55. Batch Scheduling on Kubernetes: Comparing Apache YuniKorn, Volcano.sh, and Kueue, accessed May 30, 2025, https://www.infracloud.io/blogs/batch-scheduling-on-kubernetes/
56. Practical Tips for Preventing GPU Fragmentation for Volcano Scheduler - NVIDIA Developer, accessed May 30, 2025, https://developer.nvidia.com/blog/practical-tips-for-preventing-gpu-fragmentation-for-volcano-scheduler/
57. Batch Scheduling on Kubernetes: Comparing Apache YuniKorn, Volcano.sh, and Kueue, accessed May 30, 2025, https://dev.to/infracloud/batch-scheduling-on-kubernetes-comparing-apache-yunik

orn-volcanosh-and-kueue-5b1g
58. Practical Tips for Preventing GPU Fragmentation for Volcano Scheduler - Technical Blog, accessed May 30, 2025, https://forums.developer.nvidia.com/t/practical-tips-for-preventing-gpu-fragmentation-for-volcano-scheduler/328845
59. NVIDIA/k8s-device-plugin: NVIDIA device plugin for ... - GitHub, accessed May 30, 2025, https://github.com/NVIDIA/k8s-device-plugin
60. Optimizing GPU Utilization for AI Workloads on AWS EKS - | World Journal of Advanced Research and Reviews, accessed May 30, 2025, https://journalwjarr.com/sites/default/files/fulltext_pdf/WJARR-2025-1233.pdf
61. 2025 Kubernetes Cost Benchmark Report, accessed May 30, 2025, https://430224.fs1.hubspotusercontent-na1.net/hubfs/430224/Cast%20AI%202025%20Kubernetes%20Cost%20Benchmark%20Report.pdf
62. k8s-vgpu-scheduler/docs/cambricon-mlu-support.md at master - GitHub, accessed May 30, 2025, https://github.com/chaunceyjiang/k8s-vgpu-scheduler/blob/master/docs/cambricon-mlu-support.md
63. k8s-vgpu-scheduler/docs/cambricon-mlu-support.md at master - GitHub, accessed May 30, 2025, https://github.com/warmchang/k8s-vgpu-scheduler/blob/master/docs/cambricon-mlu-support.md
64. Allocate device core and memory resource - HAMi, accessed May 30, 2025, https://project-hami.io/docs/next/userguide/hygon-device/examples/allocate-core-and-memory/
65. Allocate exclusive device - HAMi, accessed May 30, 2025, https://project-hami.io/docs/userguide/hygon-device/examples/allocate-exclusive/
66. Ascend NPU Virtualization Guide: 910 Series & 310P Best Practices - RiseUnion, accessed May 30, 2025, https://www.theriseunion.com/blog/HAMi-ascend-910b-support.html
67. What is HAMi?, accessed May 30, 2025, https://project-hami.io/docs/
68. HAMi/CHANGELOG.md at master - GitHub, accessed May 30, 2025, https://github.com/Project-HAMi/HAMi/blob/master/CHANGELOG.md
69. Time-Slicing GPUs in Kubernetes — NVIDIA GPU Operator, accessed May 30, 2025, https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/gpu-sharing.html
70. Optimizing AI Workloads with NVIDIA GPUs, Time Slicing, and Karpenter (Part 2), accessed May 30, 2025, https://blogs.cisco.com/developer/optimizing-ai-workloads-with-nvidia-gpus-time-slicing-and-karpenter-part-2
71. Compare with HAMi | Maximize your GPU usage, powered by GPU ..., accessed May 30, 2025, https://tensor-fusion.ai/guide/comparison/compare-with-hami
72. AliyunContainerService/gpushare-scheduler-extender ... - GitHub, accessed May 30, 2025, https://github.com/AliyunContainerService/gpushare-scheduler-extender
73. GPU Sharing Scheduler Extender Now Supports Fine-Grained Kubernetes Clusters, accessed May 30, 2025,

https://www.alibabacloud.com/blog/gpu-sharing-scheduler-extender-now-supports-fine-grained-kubernetes-clusters_594926
74. Implementing Fractional GPUs in Kubernetes with Aliyun Scheduler - Hugging Face, accessed May 30, 2025, https://huggingface.co/blog/NileshInfer/implementing-fractional-gpus-in-kubernetes