KUBECON + CLOUDNATIVECON NORTH AMERICA 2026 · CONTRIBFEST

DYNAMIA.AI
Creators of HAMi

GPU Virtualization: First Setup and Understanding the Device Plugin / DRA

Reza Jelveh
Solution Architect, Dynamia AI - Makers of HAMi
github.com/fishman linkedin.com/in/rezajelveh

Mengxuan Li
CTO, Dynamia AI - HAMi maintainer



The 80 minutes

Tuesday, November 10, 4:40 to 6:00 pm · Salt Palace 255 D



Before the session

Everything runs on your laptop. The GPUs are fake.

```
git clone https://github.com/Project-HAMi/hami-workshop
cd hami-workshop/examples/containerdays-singapore
git clone https://github.com/fishman/hami-mock-monitor
docker pull kindest/node:v1.37.0 busybox:1.36 ghcr.io/llm-d/llm-d-inference-sim:v0.12.0
```

- Needs Docker (or OrbStack / Colima), 4 CPU and 8 GB free, plus `kind`, `kubectl`, `helm`, `jq`, `git`
- Go 1.27+ only to build HAMi or contribute code. A GitHub account for Lab 4
- Conference Wi-Fi is slow. Pull images at home

DYNAMIA.AI
Creators of HAMi

Why share GPUs

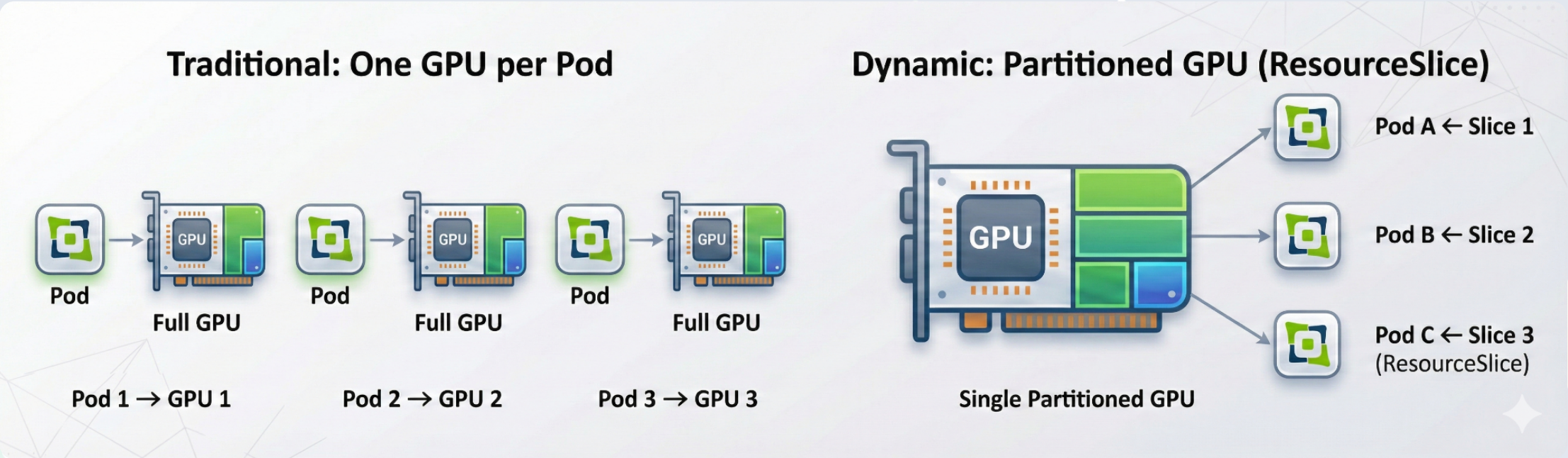


One GPU per task is the default

What it costs

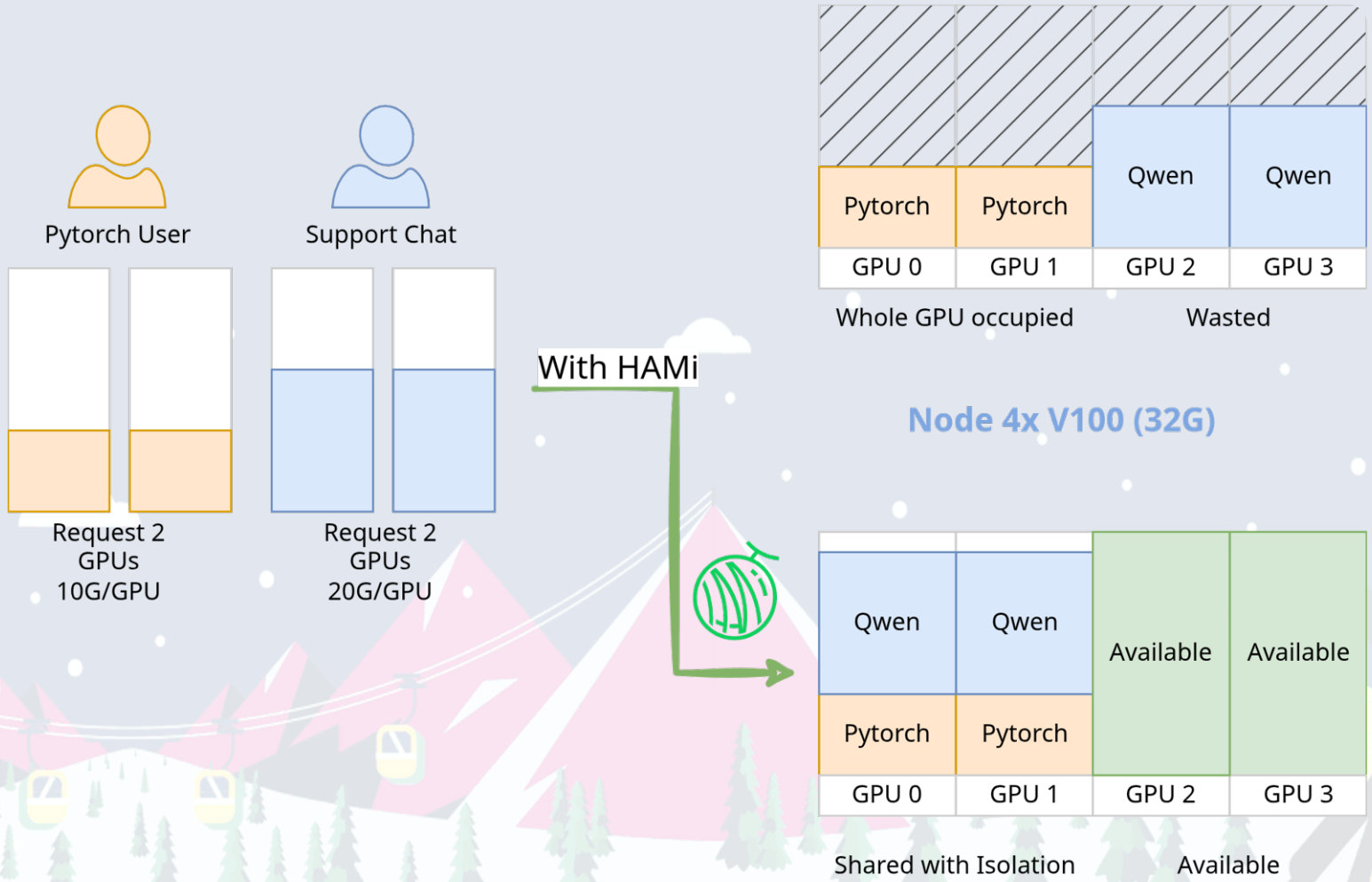
Kubernetes allocates GPUs atomically: one whole device, one task.

- A 1 GB task blocks an 80 GB device
- Most GPUs sit idle most of the time
- No advanced scheduling yet: no binpack, no spread, no topology



What is HAMi

One GPU, many tasks



HAMi Capabilities

Six things HAMi brings to GPU scheduling

Heterogeneous Management

Manage GPU, NPU, MLU, and other accelerators in one workflow.

Hard Isolation

Slice memory and compute with hard isolation at runtime.

Advanced Scheduling

Binpack, spread, and topology-aware placement policies.

Kubernetes Native

Kubernetes-native APIs, DRA, and CDI support.

Resource Isolation & QoS

Memory and core quotas for fair, stable sharing.

Unified Monitoring

Consistent metrics and visibility across vendors.

Asking for a slice

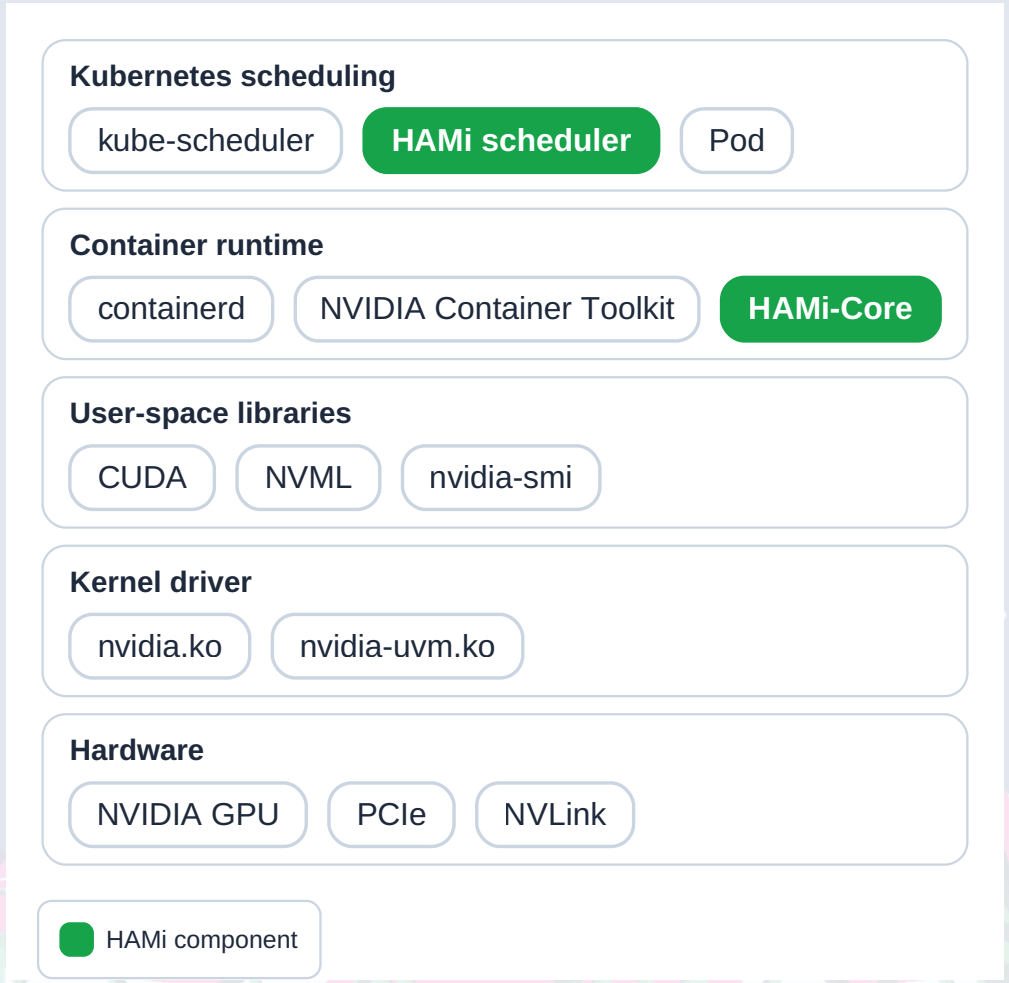
One API across vendors, no code changes

- **NVIDIA, Ascend, Cambricon, Hygon, Iluvatar** and more
- **Fine-grained:** memory as small as 1 MB, compute as small as 1% of a card
- **Transparent to tasks:** no code changes required
- **Hard resource isolation** inside containers

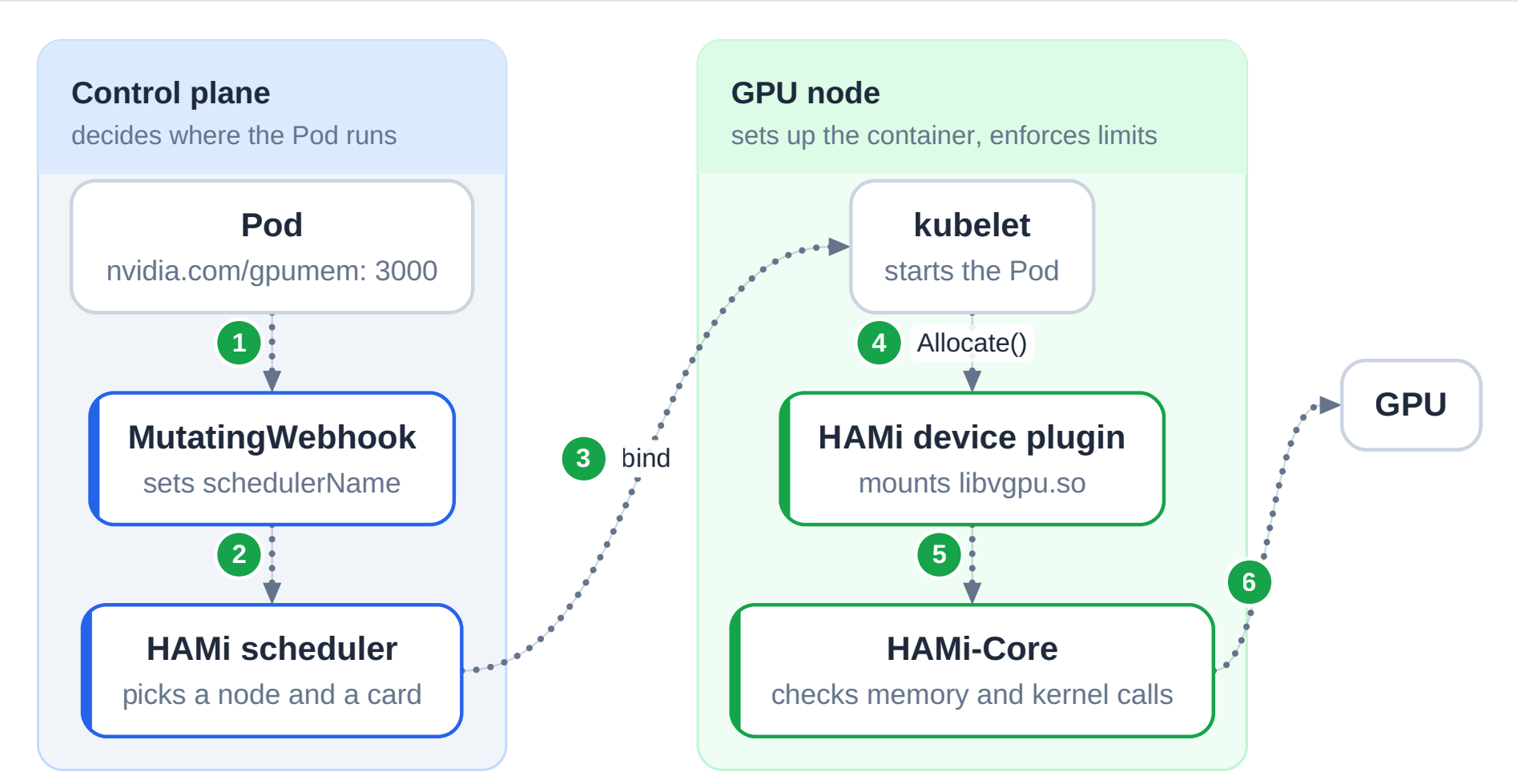
```yaml
# NVIDIA: 15 GB and 30% of one card
resources:
  limits:
    nvidia.com/gpu: 1
    nvidia.com/gpumem: 15000
    nvidia.com/gpucores: 30
```

```yaml
# Ascend 910C: 8GB + 20% compute
resources:
  limits:
    huawei.com/Ascend910C: "1"
    huawei.com/Ascend910C-core: "20"
    huawei.com/Ascend910C-memory: "8192"
```

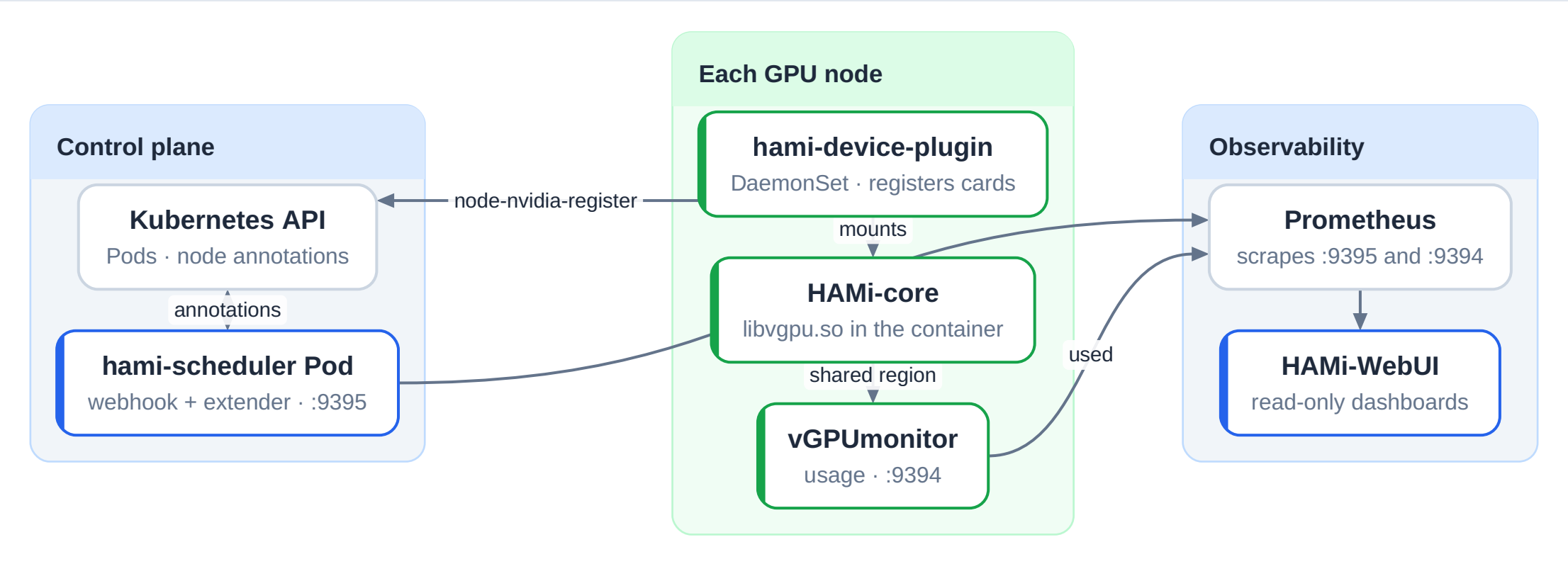
Where HAMi sits in the GPU stack



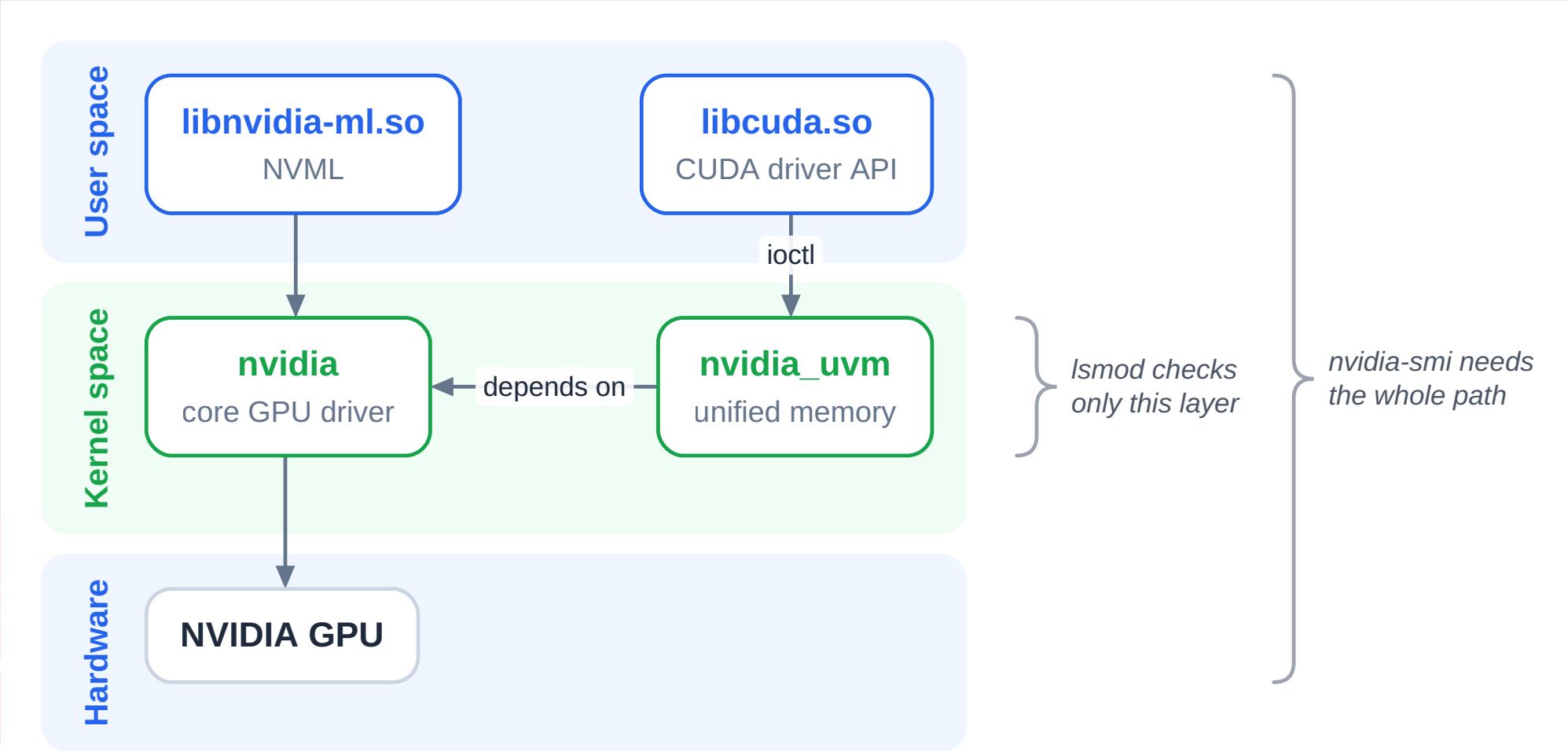
A Pod's path to the GPU



How the HAMi components work together

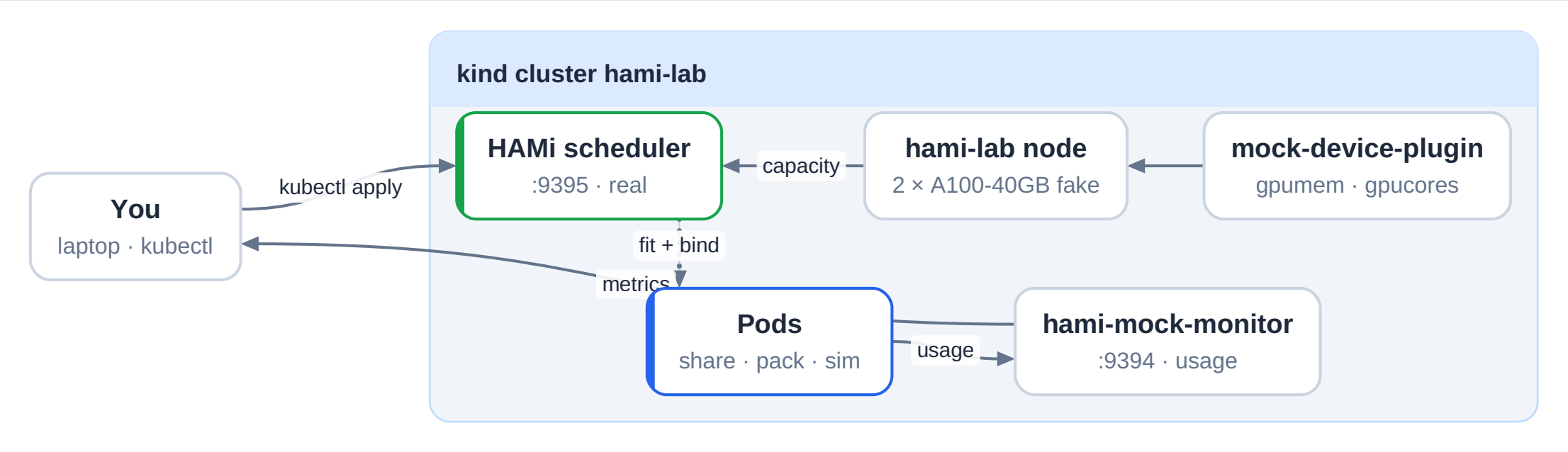


Why `nvidia-smi` can fail when `lsmod` looks fine



HAMi-Core intercepts at the CUDA and NVML layer, above the kernel driver.

Lab setup: real scheduler, fake GPUs



mock-device-plugin

Turns a node annotation into `nvidia.com/gpumem` and `nvidia.com/gpucores` capacity. The real device plugin writes that annotation from NVML.

hami-mock-monitor

Produces HAMi's usage metrics (normally from vGPUmonitor and HAMi-core) from a load source instead of a GPU.

DYNAMIA.AI
Creators of HAMi

Lab 1: Install HAMi



Lab 1: one command

```
./setup.sh
```

The script:

- Creates a kind cluster `hami-lab` and labels the node gpu=on
- Installs HAMi 2.10.0 with the real device plugin off and the mock plugin on
- Fakes two A100-40GB cards. Two pieces:the count resource `nvidia.com/gpu=20`: the mock plugin's health gate, 2 cards x 10 splits
- the node annotation `hami.io/node-nvidia-register` from `fake-gpus.json`, what the real device plugin writes after it reads NVML
- Builds and installs hami-mock-monitor with `completeBinds=true`

Expected at the end:

```
{ "nvidia.com/gpu": "20", "nvidia.com/gpucores": "200",
  "nvidia.com/gpumem": "81920", "nvidia.com/gpumem-percentage": "200" }
```

Lab 1: look around

```
kubectl -n kube-system get pods | grep hami
kubectl -n kube-system get cm hami-scheduler-device -o yaml | less
kubectl get node hami-lab-control-plane \
  -o jsonpath='{.metadata.annotations.hami\.io/node-nvidia-register}' | jq
```

```
[{"id":"GPU-mock-0","index":0,"count":10,"devmem":40960,"devcore":100,
  "type":"NVIDIA-A100-SXM4-40GB","health":true,"numa":0,"mode":"hami-core"}, ...]
```

- `count`: how many Pods may share the card. `devmem` in MB, `devcore` in percent
- The card list lives on the node, written by the device plugin. The scheduler reads it at `Filter`

DYNAMIA.AI
Creators of HAMi

Lab 2: Share a GPU



Lab 2: three Pods, two cards

```
kubectl apply -f share.yaml -f too-big.yaml
kubectl get pods -o wide
```

`share-1..3` each ask for 15000 MB and 30% of a card, and run. `too-big` asks for 45000 MB, more than any 40960 MB card, and stays `Pending`:

```
kubectl get events --field-selector involvedObject.name=too-big | tail -2
# 0/1 nodes are available: 1 2/2 CardInsufficientMemory.
```

45000 MB fits the node total, but no single 40960 MB card.

Lab 2: read what HAMi decided

```
for p in share-1 share-2 share-3; do
  echo -n "$p "; kubectl get pod $p -o jsonpath='{.metadata.annotations.hami\.io/vgpu-devices-allocated}{"\n"}'
done
# share-1 GPU-mock-1,NVIDIA,15000,30:;
# share-2 GPU-mock-0,NVIDIA,15000,30:;
# share-3 GPU-mock-1,NVIDIA,15000,30:;
```

The format is `uuid,type,memory_mb,cores:` per device, `;` per container. The device plugin reads it in `Allocate`.

The interception chain

Four steps, one node

- The admission **webhook** rewrites `schedulerName` to `hami-scheduler`
- The scheduler **extender** at `Filter` reads the node annotation `hami.io/node-nvidia-register` (the card list from Lab 1), checks memory and core headroom per card, and picks a physical card UUID
- At `Bind` it writes `hami.io/vgpu-devices-allocated` onto the Pod, sets `hami.io/bind-phase=allocating` and takes the node lock `hami.io/mutex.lock`
- The device plugin's `Allocate` mounts `/dev/nvidia*` and the vGPU files, reading the target from the Pod annotation, then marks `success` and releases the lock

> *The mock plugin skips step 4's completion, which is why hami-mock-monitor's* `completeBinds` *exists. Without it the next Pod waits up to 5 minutes for the lock.*

On a real node, step 4 also injects `libvgpu.so` via `ld.so.preload`, and HAMi-core enforces the memory and core limits from that moment on.

DYNAMIA.AI
Creators of HAMi

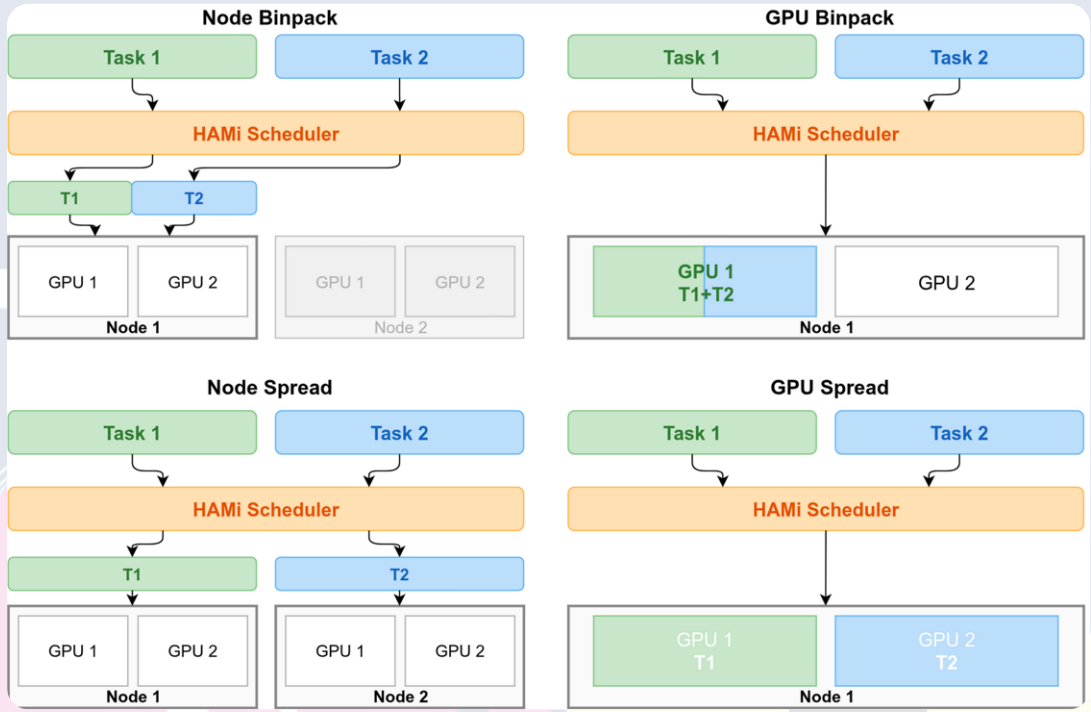
Lab 3: Scheduler policies



Scheduling policies

Binpack & spread, on nodes and on GPUs

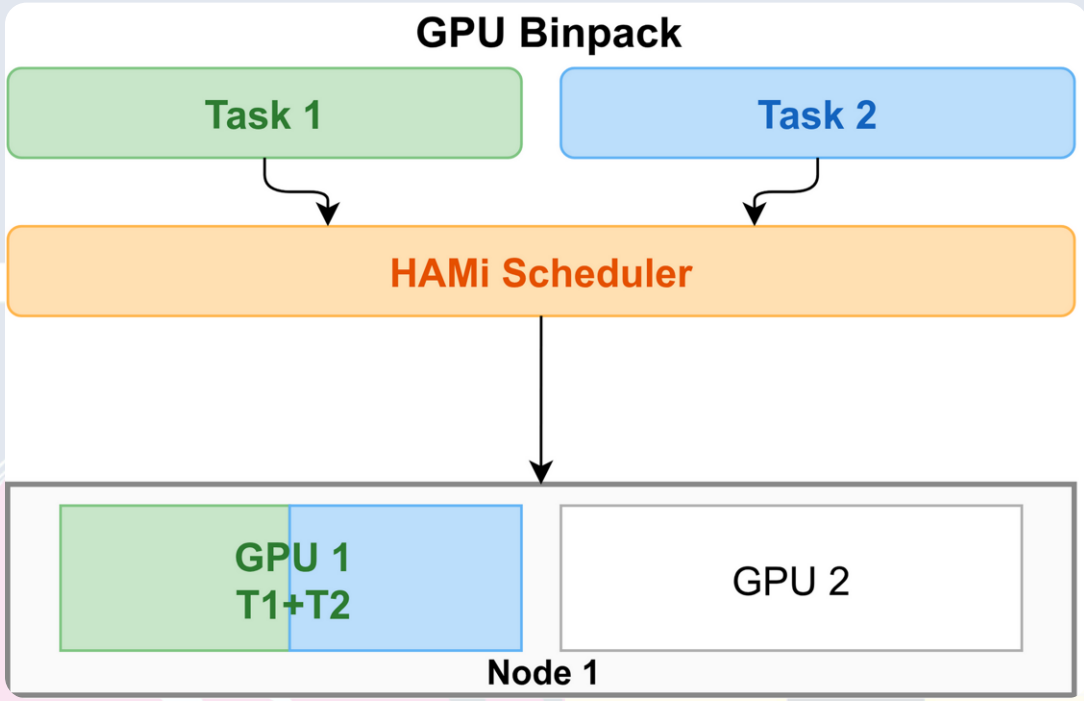
- **Node binpack** frees whole machines: reduces cost, helps cluster autoscaler
- **Node spread** isolates faults: HA across zones, blast radius control
- **GPU binpack** prevents fragmentation: frees entire GPUs for training
- **GPU spread** protects tail latency: reduces HBM and NVLink contention



GPU binpack

Fill one card, run many workloads

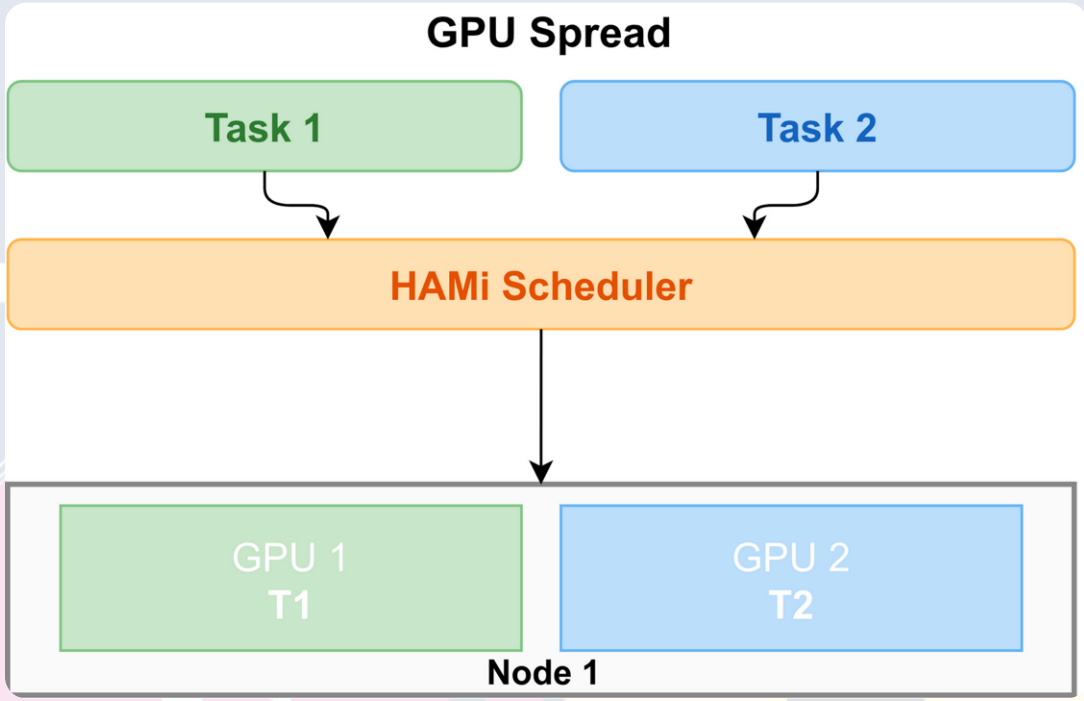
- Several workloads share one card, each with its own slice of memory and compute
- Prevents fragmentation: scattered small workloads no longer block whole cards
- Frees whole cards for the job that cannot share
- Best for: fleets of small workloads, batch jobs, dev and test



GPU spread

One workload per card, no shared bottlenecks

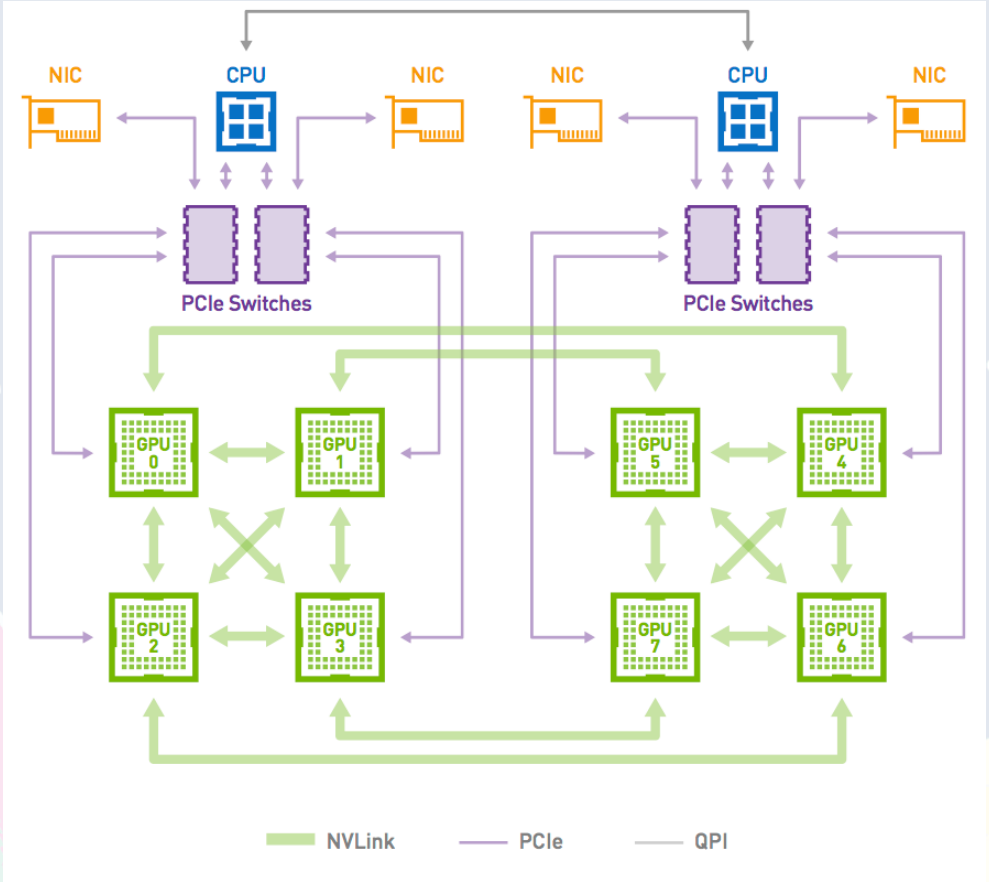
- One workload per card: compute and memory bandwidth are never shared
- Isolates noisy neighbours: a thrasher on one card does not slow the others
- Protects tail latency: no cross-tenant contention
- Best for: latency-sensitive serving with strict SLOs



Scheduling policies

Topology-aware

- **NVLink 3 (A100):** 600 GB/s, 12 links
- **NVLink 4 (H100/H200):** 900 GB/s bidirectional across 18 links
- **NVLink 5 (B200/B300):** 1.8 TB/s, 14x PCIe 5.0
- **NVLink 6 (Rubin):** ~3.6 TB/s target
- **PCIe 5.0 x16:** 128 GB/s. **PCIe 6.0:** 242 GB/s
- **HAMi topology policy:** prefers NVLink (NVIDIA), HCCS (Ascend), and other high-speed interconnects, avoids PCIe bridge pairs



Placement happens on two axes

Axis	Annotation	binpack	spread
Which node	`hami.io/node-scheduler-policy`	fewer nodes	across nodes
Which card on that node	`hami.io/gpu-scheduler-policy`	same card	across cards

- Both axes score cards by utilization ratios: binpack prefers the fuller card (consolidation), spread prefers the emptier one (dispersion)
- This lab has one node, so the node axis has a single candidate
- The card axis is fully observable on the two fake A100s

Lab 3: binpack

```
kubectl delete -f share.yaml -f too-big.yaml
kubectl apply -f binpack.yaml
for p in pack-1 pack-2; do
  echo -n "$p "; kubectl get pod $p -o jsonpath='{.metadata.annotations.hami\.io/vgpu-devices-allocated}{"\n"}'
done
```

Both Pods report the same `GPU-mock-X`: two workloads sharing one card. Frees whole cards for big jobs.

Lab 3: spread

```
kubectl delete -f binpack.yaml
kubectl apply -f spread.yaml
for p in spread-1 spread-2; do
  echo -n "$p "; kubectl get pod $p -o jsonpath='{.metadata.annotations.hami\.io/vgpu-devices-allocated}{"\n"}'
done
```

One Pod per card. Same cluster, same requests, different policy, different placement.

```
metadata:
  annotations:
    hami.io/gpu-scheduler-policy: spread   # or binpack, per Pod
```

Where topology information comes from

Every vendor: the node's device plugin reads it, an annotation carries it, the scheduler scores it



Topology, vendor by vendor

- **NVIDIA:** `ENABLE_TOPOLOGY_SCORE=true` scores every card pair from NVML into `hami.io/node-nvidia-score`. Policy `topology-aware`: one card gets the weakest-linked card, N cards get the best-linked set
- **AMD:** `xgmiPeers` and numa per GPU in `hami.io/node-amd-register`. Pod annotations `amd.com/numa-bind`, `amd.com/link-bind`, or policy `topology-aware`
- **MetaX:** the vendor plugin writes `metax-tech.com/gpu.topology.scores` and `.losses`. MetaXLink first, then PCIe switch
- **Kunlunxin, AWS Neuron:** no link data; the layout is fixed, so the scheduler reads it from card index (NUMA halves of 8, Neuron rings)

```yaml
metadata:
  annotations:
    hami.io/gpu-scheduler-policy: topology-aware
```

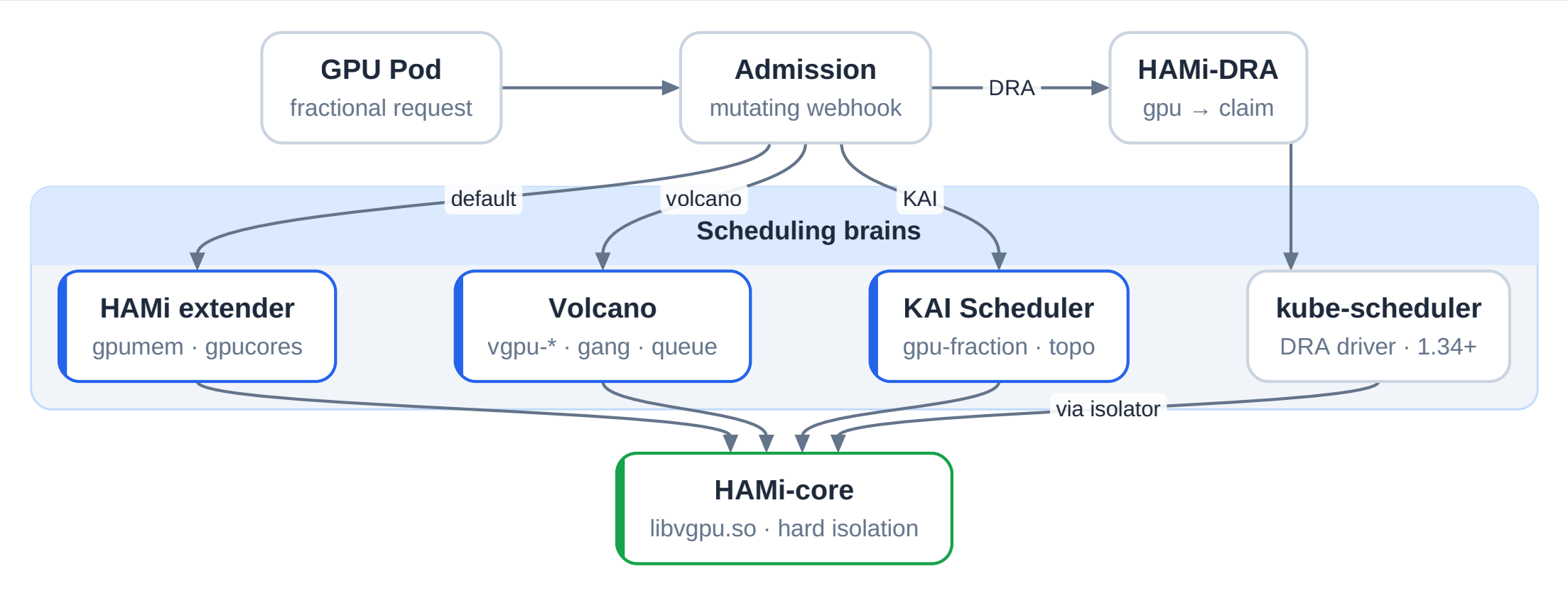
DYNAMIA.AI
Creators of HAMi

Lab 4: Beyond the default scheduler



HAMi is two things

One isolation layer, several scheduling brains



Three ways to use HAMi

DYNAMIA.AI
Creators of HAMi

Standard HAMi

HAMi scheduler extender + HAMi device plugin.

`nvidia.com/gpu`, `gpumem`, `gpucores`

Volcano vGPU

Volcano `deviceshare` + volcano-vgpu-device-plugin.

`volcano.sh/vgpu-number`, `vgpu-memory`, `vgpu-cores`

KAI Scheduler

KAI GPU sharing + kai-resource-isolator.

`gpu-memory` or `gpu-fraction` annotation

All three use **HAMi-core** in the container for isolation. One rule: one device-plugin path per GPU node.

KAI Scheduler: composes with HAMi

NVIDIA's scheduler for AI workloads (CNCF Sandbox): gang scheduling, hierarchical fair queues, fractional GPU, topology-aware placement.

```
helm install kai-scheduler oci://ghcr.io/nvidia/kai-scheduler --version v0.17.0 \
  --set global.gpuSharing=true --set binder.plugins.hamicore.enabled=true \
  -n kai-scheduler --create-namespace
```

- Since v0.16.4 the kai-resource-isolator DaemonSet plus a mutating webhook deliver the same `libvgpu.so` into Pods
- `CUDA_DEVICE_MEMORY_LIMIT` injected from `gpu-fraction` or `gpu-memory` annotations
- KAI makes the placement decisions, HAMi-core makes the memory cap hard

Volcano: gang jobs on shared GPUs

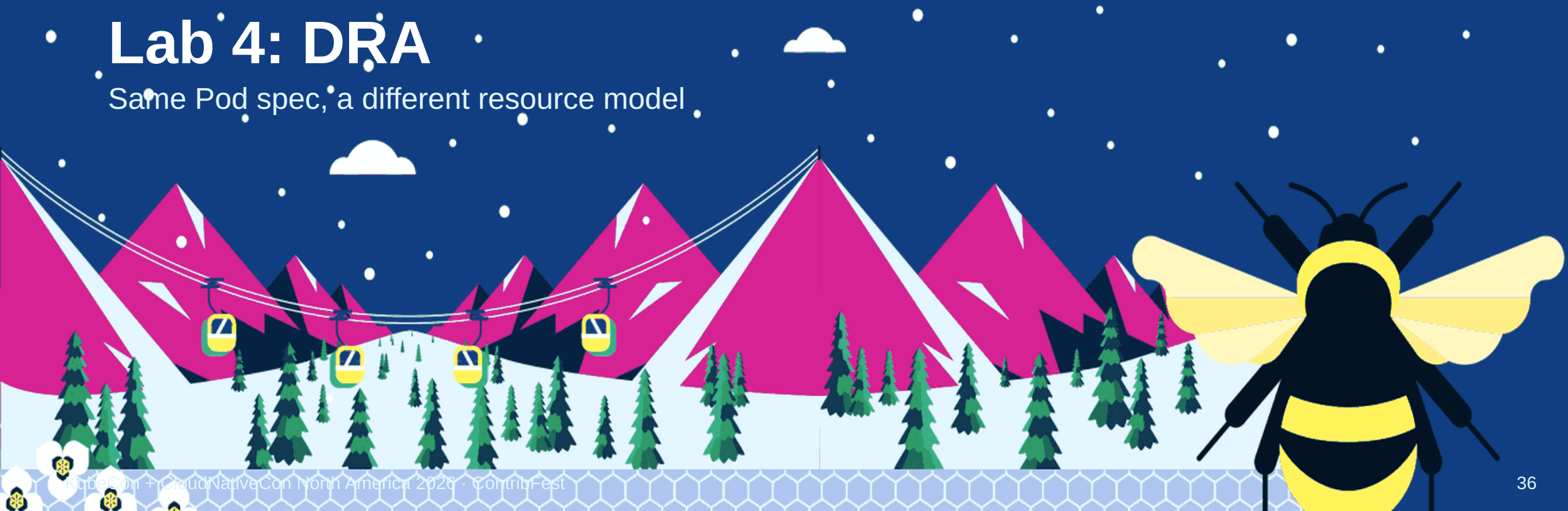
```yaml
kind: Job                    # batch.volcano.sh/v1alpha1
spec:
  schedulerName: volcano
  minAvailable: 2            # both workers or none
  queue: vgpu-queue
  tasks:
  - replicas: 2
    template:
      spec:
        containers:
        - resources:
            limits:
              volcano.sh/vgpu-number: 1
              volcano.sh/vgpu-memory: 2000
              volcano.sh/vgpu-cores: 30
```

- Volcano `deviceshare` plugin plus the volcano-vgpu-device-plugin, not the standard HAMi chart
- Queue `capability` caps vGPU memory per team

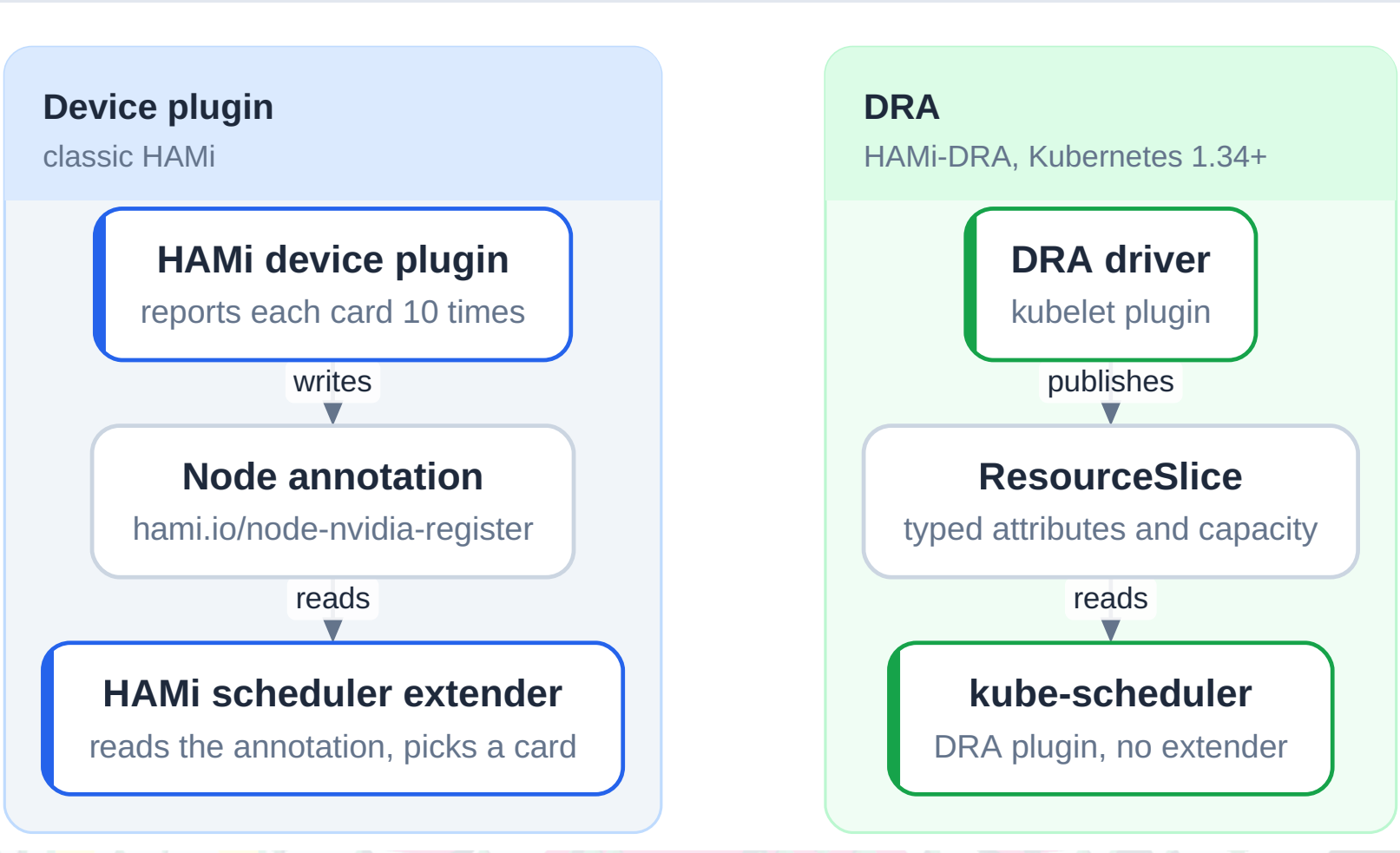
DYNAMIA.AI
Creators of HAMi

Lab 4: DRA

Same Pod spec, a different resource model



Device plugin vs DRA



The four DRA objects

ResourceSlice

Published by the DRA driver. Lists each device with its attributes and capacity.

DeviceClass

A named set of devices, selected with CEL. HAMi: `hami-core-gpu.project-hami.io`

ResourceClaim

A Pod's request for devices, like a PersistentVolumeClaim.

ResourceClaimTemplate

Kubernetes creates one ResourceClaim per Pod from it.

Consumable capacity: one card, many claims

DYNAMIA.AI
Creators of HAMi

```yaml
capacity:
  cores:
    value: "100"
    requestPolicy: { validRange: { min: "0", max: "100", step: "1" } }
  memory:
    value: 15Gi
    requestPolicy: { validRange: { min: 1Mi, max: 15Gi, step: 1Mi } }
```

- A Tesla T4 published as one device with two capacities, plus `allowMultipleAllocations: true`
- The scheduler does the accounting and enforces min, max and step. Extended resources accept any number
- Feature gate `DRAConsumableCapacity`: alpha in 1.34, on by default since 1.36

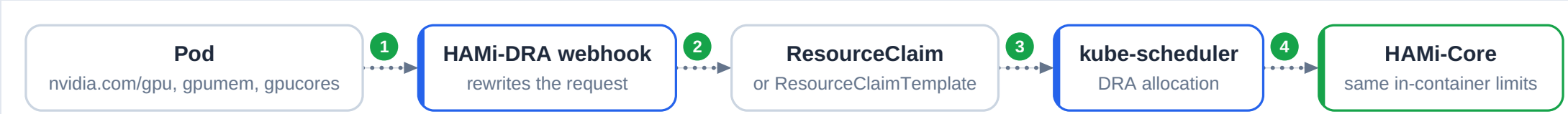
A claim for 4Gi and 30 cores

```yaml
apiVersion: resource.k8s.io/v1
kind: ResourceClaim
metadata: { name: single-gpu-0, namespace: test-dra }
spec:
  devices:
    requests:
    - name: single-gpu
      exactly:
        deviceClassName: hami-core-gpu.project-hami.io
        capacity:
          requests: { cores: 30, memory: "4Gi" }
```

```
| N/A   59C    P8   16W /  70W |   0MiB /   4096MiB |   0%   Default |
```

`nvidia-smi` in the Pod reports 4096 MiB: HAMi-Core enforces the claim. A second Pod's claim lands on the same card, same UUID, its own `shareID`.

HAMi-DRA: keep the HAMi Pod spec



- NVIDIA, Hygon HCU, AMD and Ascend in one deployment (`deviceVendors`)
- `resourceClaimTemplate: true` lets Kueue admit the Pods. `batch/v1` Jobs are not covered yet
- Needs Kubernetes 1.34+, `DRAConsumableCapacity`, and CDI in containerd or CRI-O

github.com/Project-HAMi/HAMi-DRA

DRA and the schedulers above it

DYNAMIA.AI
Creators of HAMi

Kueue

Admits ResourceClaimTemplates. Needs the HAMi DeviceClass in `deviceClassMappings`.

Volcano

Today: `deviceshare` plus the vGPU device plugin. Gang and queue capacity on shared GPUs.

KAI Scheduler

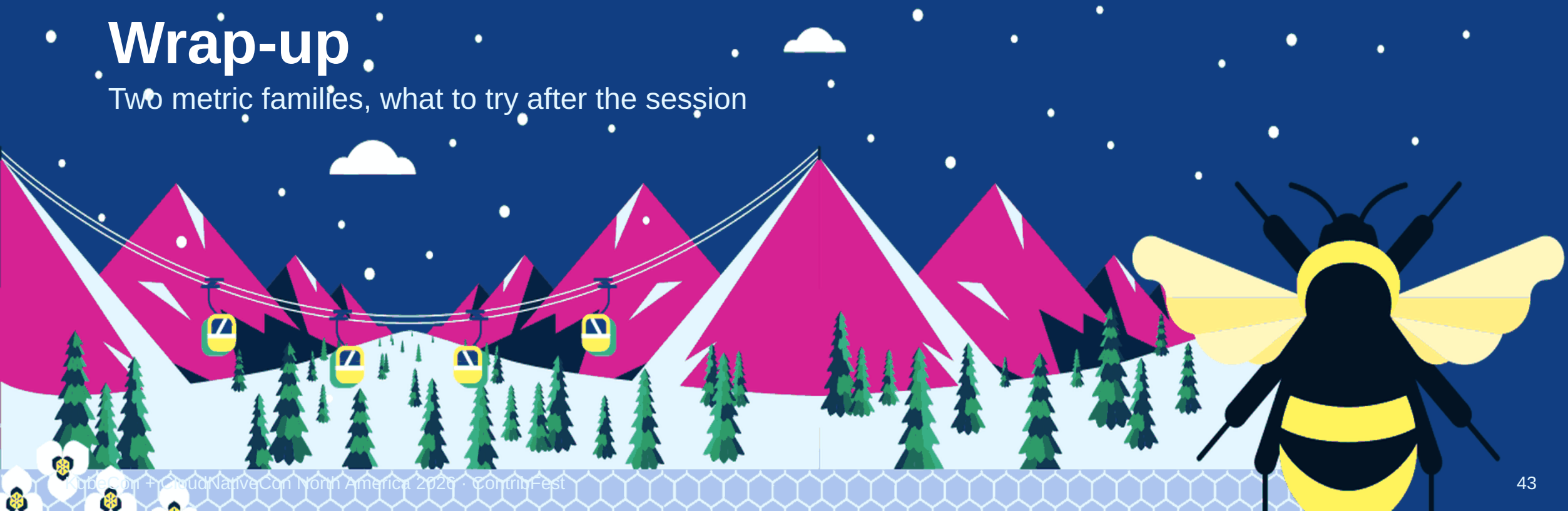
Today: GPU fractions with HAMi-Core as the isolation layer.

Open questions for the room: who decides partitioning, the driver or the scheduler? How do quotas count fractional devices?

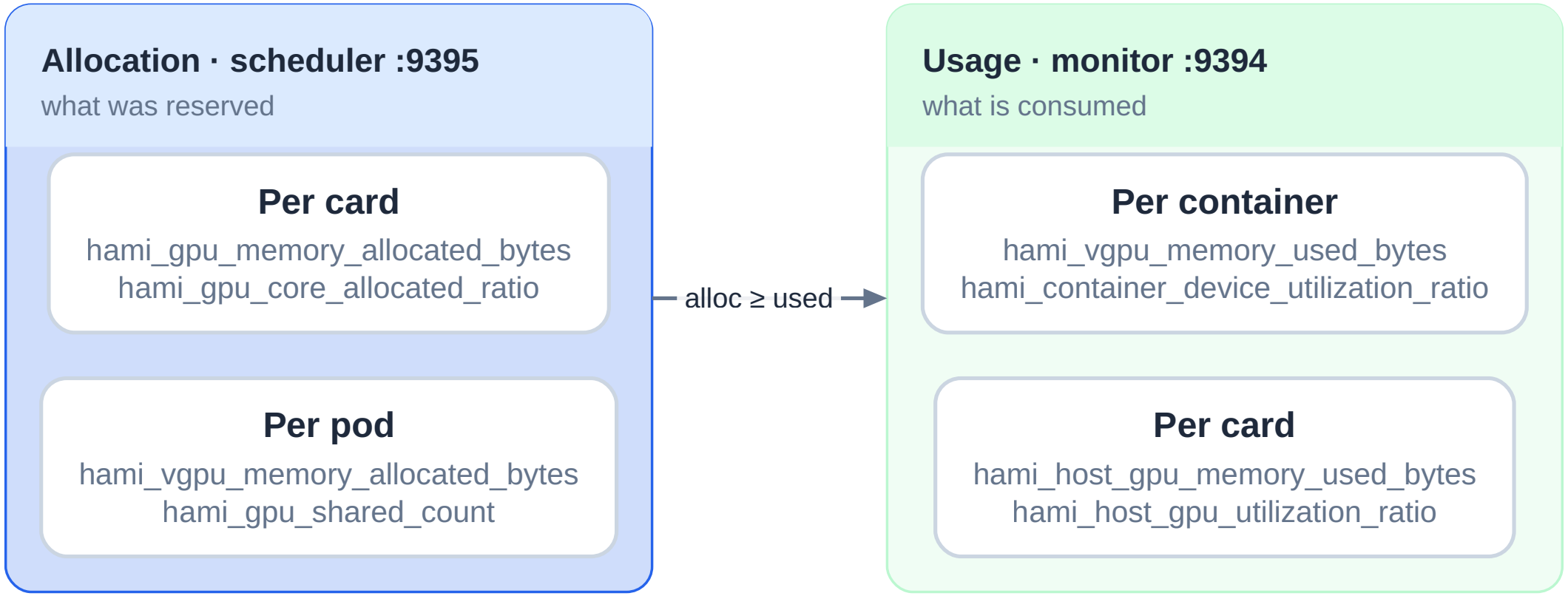
DYNAMIA.AI
Creators of HAMi

Wrap-up

Two metric families, what to try after the session



Reserved is not used




```
./metrics.sh scheduler   # allocation, port 9395
./metrics.sh monitor     # usage, port 9394
```

Two metric sources

Scheduler :9395, allocation

- `gpu_memory_allocated_bytes`
- `gpu_core_allocated_ratio`
- `node_gpu_memory_allocated_ratio`
- `gpu_shared_count`
- `vgpu_memory_allocated_bytes`
- `resource_quota_used`

vGPUmonitor :9394, usage

- `vgpu_memory_used_bytes`
- `vgpu_memory_limit_bytes`
- `container_device_utilization_ratio`
- `host_gpu_utilization_ratio`
- `host_gpu_memory_used_bytes`
- `container_last_kernel_elapsed_seconds`

All names start with `hami_`. Same labels: node, `device_uuid`, `namespace`, pod. Join them in PromQL.

Give the mock a realistic load

```
kubectl apply -f sim.yaml      # llm-d-inference-sim, requests oscillate 0 to 5
watch -n 5 "./metrics.sh monitor | grep 'pod=\"sim\"'"
```

```
hami_container_device_utilization_ratio{pod="sim"} 26.4
hami_node_gpu_memory_allocated_ratio{device_uuid="GPU-mock-1"} 0.73
hami_host_gpu_utilization_ratio{device_uuid="GPU-mock-1"}      20.0
```

- `sim` is a vLLM-compatible inference simulator: utilization moves between 0 and 33 (load x allocated cores)
- Memory stays at 0.9 x the limit, because vLLM preallocates its KV cache

Question. Card `GPU-mock-1` shows 73% memory allocated but 20% used. Which number goes on a capacity dashboard, and which on a cost report?

Live: isolation on a real GPU

DYNAMIA.AI
Creators of HAMi

Ask

`nvidia.com/gpumem: 3000`

See

`nvidia-smi` in the Pod reports 3000 MiB total.

Allocating past the limit fails inside the Pod with out of memory. Neighbours keep running.

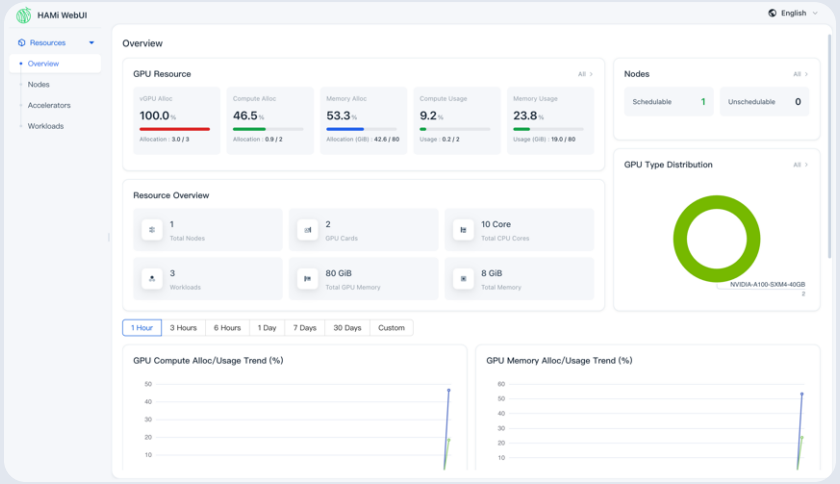
After the session: HAMi WebUI

`metrics.sh` is fine for a terminal, but the HAMi WebUI turns the same numbers into dashboards. It needs a Prometheus in the cluster:

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo add hami-webui https://project-hami.github.io/HAMi-WebUI
helm upgrade --install prometheus prometheus-community/kube-prometheus-stack \
  --version 75.15.1 -n monitoring --create-namespace --set grafana.enabled=false --wait
kubectl apply -f monitoring.yaml   # scrape :9395 and :9394
kubectl apply -f webui-rules.yaml  # translate metric names for the WebUI
helm upgrade --install my-hami-webui hami-webui/hami-webui \
  --set externalPrometheus.enabled=true --set dcgm-exporter.enabled=false \
  -n kube-system --wait
kubectl -n kube-system port-forward svc/my-hami-webui 3000:3000
```

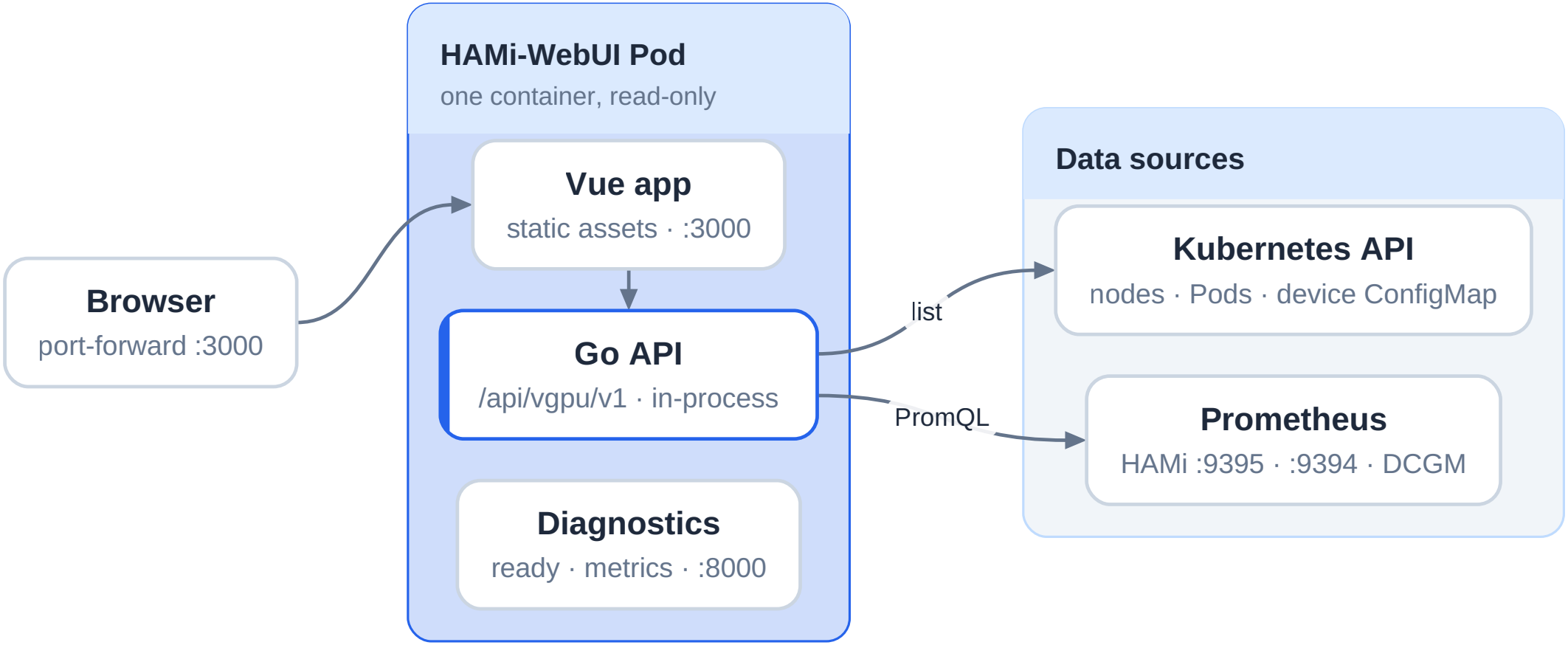
Open http://localhost:3000.

Why `webui-rules.yaml`



- The WebUI queries vGPUmonitor's metric names (`hami_memory_size`, `hami_core_util`, `hami_container_vgpu_allocated`, ...)
- This lab's scheduler and mock monitor expose the newer byte/ratio names (`hami_gpu_memory_limit_bytes`, `hami_host_gpu_utilization_ratio`, ...)
- The PrometheusRule records one set as the other. Without it every allocation and usage panel stays empty

How the WebUI integrates



Hack on the WebUI

```
git clone https://github.com/Project-HAMi/HAMi-WebUI && cd HAMi-WebUI
make bootstrap && make -C server bootstrap   # pnpm deps, Go generators (+ protoc)
make -C server run   # Go API on :8000, uses kubeconfig + server/config/config.yaml
make dev             # Vite on :3000, proxies /api/vgpu/v1 to :8000
make verify && make -C server verify         # before the PR
```

- Point `server/config/config.yaml` at the lab's Prometheus (port-forward `:9090`)
- Go, Node and pnpm versions are pinned in the repo. `git commit -s`

After the session: contribute

DYNAMIA.AI
Creators of HAMi

HAMi

`make test`. Start in `Fit`, `PatchAnnotations` (`pkg/device/nvidia/device.go`) or `Filter`, `Bind` (`pkg/scheduler/scheduler.go`). `git commit -s`.

HAMi-DRA

The webhook that turns `nvidia.com/gpu` requests into DRA ResourceClaims. Needs Kubernetes 1.34+.

mock-device-plugin

Complete the bind in `Allocate` (a patch is prepared in `hami-mock-monitor/upstream/mock-device-plugin`), or add a vendor.

hami-mock-monitor

Other vendors (Ascend, Hygon), a Grafana dashboard for the lab, a published image.

Look for good `first issue`. Open a draft PR before you leave. Disclose AI help in the PR.

Clean up

```
kind delete cluster --name hami-lab
```

Where HAMi Is Today

Production metrics from CNCF case studies

DYNAMIA.AI
Creators of HAMi

0

Driver, kernel, code changes needed
SF Technology

1-2 GB

Memory per small-model inference service
KE Holdings

4x

GPU utilization: 5-10% to 30-50%
NIO

90%

GPU infra managed by HAMi on RTX 4070/4090
Prep Edu

China Merchants Bank - SNOW Corp. - NIO - KE Holdings - DaoCloud - SF Technology - Prep Education - cncf.io/case-studies

DYNAMIA.AI
Creators of HAMi

Community & Adopters

Devices, integrations, and who uses HAMi

Open Source, CNCF Backed, Production Ready

5k	700k	500+	27
Github Stars	Docker Pulls	Contributors	Contributor Countries

kubernetes VOLCANO
Kueue koordinator
KAI Scheduler COZYSTACK

Ecosystem & Device Support

NVIDIA AMD Ascend Cambricon 寒武纪 HYGON 海光 天数智芯 Iluvatar CoreX METAX 沐曦
摩尔线程 MOORE THREADS 昆仑芯 KUNLUNXIN Enflame 燧原科技 壁仞科技 aws 瀚博半导体

Adopters

4Paradigm 百度智能云 贝壳 招商银行 China Merchants Bank 中国移动 China Mobile 中国联通 China unicom DaoCloud
DYNAMIA H3C HUAWEI LinkedIn 马上消费 NIO PPIO
PREP SAP SF TECHNOLOGY 顺丰科技 思特奇 SI-TECH SNOW viettel

DYNAMIA.AI
Creators of HAMi

Thank you

github.com/Project-HAMi/HAMi · project-hami.io · lab: github.com/Project-HAMi/hami-workshop

Reza Jelveh
Global Market & Solution Architecture, Dynamia AI
github.com/fishman linkedin.com/in/rezajelveh

Mengxuan Li
CTO, Dynamia AI - HAMi maintainer

