CDW Documentation

NCP-AIO Lab Walkthrough Study Guide

NCP-AIO Lab Walkthrough Study Guide

A hands-on command reference for the four major lab domains on the NVIDIA Certified Professional — AI Operations exam. Work through each domain as a sequence: review the concept, run the commands, then test yourself with the troubleshooting scenarios at the end of each section.

---

1. Base Command Manager (BCM) & Cluster Administration

BCM (formerly Bright Cluster Manager) is NVIDIA's central control plane for HPC/AI clusters. The two interfaces you must know are **`cmsh`** (the cluster management shell) and **Base View** (the web UI). The exam leans heavily on `cmsh`.

1.1 Getting into BCM

```
# SSH to the head node, then drop into the management shell
ssh root@head-node
cmsh

# Inside cmsh, get help and orient yourself
[head]% help
[head]% main              # return to the top-level prompt
[head]% status            # overall cluster status (services, license, HA)
[head]% partition use base
```

cmsh uses **modes**. You enter a mode (`device`, `category`, `softwareimage`, `user`, etc.), then operate on objects inside it. After making changes, you **must** `commit` — pending changes show with an asterisk in the prompt.

1.2 Deploying a system (provisioning a compute node)

The typical provisioning flow: pick or create a **software image**, attach it to a **category**, assign nodes to that category, then power them on so they PXE-boot and provision.

```
cmsh
[head]% softwareimage
[head->softwareimage]% list
[head->softwareimage]% clone default-image gpu-image
[head->softwareimage]% use gpu-image
[head->softwareimage[gpu-image]]% set kernelversion 5.15.0-...
[head->softwareimage[gpu-image]]% commit

# Create a category that uses this image
[head]% category
```

```
[head->category]% add gpu-nodes
[head->category[gpu-nodes*]]% set softwareimage gpu-image
[head->category[gpu-nodes*]]% commit

# Assign a node to the category and provision it
[head]% device
[head->device]% use node001
[head->device[node001]]% set category gpu-nodes
[head->device[node001]]% commit
[head->device[node001]]% power on        # or: reset, off
[head->device[node001]]% status          # watch state: INSTALLING -> UP

# Useful one-liners (run from cmsh prompt)
[head]% device list                       # all devices and state
[head]% device status -c gpu-nodes        # status filtered by category
[head]% device foreach -c gpu-nodes (status)
```

Outside cmsh, the equivalent inspection commands include:

```
cmha status              # HA status if head nodes are paired
module load shared       # load the BCM environment module
pdsh -g category=gpu-nodes uptime   # parallel ssh across a category
```

1.3 Aligning baseline images

“Aligning” means making sure the running nodes match the software image on the head node. After you edit packages or files in the image, you sync them out.

```
# Update packages inside an image (chroot-style)
cm-chroot-sw-img /cm/images/gpu-image
# ... inside the chroot ...
yum install -y datacenter-gpu-manager
exit

# Push the image to running nodes — choose ONE depending on the situation
[head]% device use node001
[head->device[node001]]% imageupdate               # incremental sync
(live)
[head->device[node001]]% reinstall                 # full PXE reprovision

# Verify nodes are in sync with the image
[head]% device imageupdate -c gpu-nodes --dry-run   # show what would change
```

Know the difference cold for the exam: **`imageupdate`** is a live rsync of the image to a running node (fast, but not all changes apply without reboot). **`reinstall`** wipes and PXE-provisions from scratch (clean but slow).

1.4 Firmware updates

```
# Check current firmware across a category
[head]% device foreach -c gpu-nodes (get bmcsettings)

# BCM ships cm-update-firmware for BMC/BIOS rollouts
cm-update-firmware --help
cm-update-firmware --category gpu-nodes --firmware-package
/root/fw/bios-1.2.bin

# GPU firmware (e.g., VBIOS) is usually handled by nvidia-smi or vendor
tools
nvidia-smi -q | grep -i "vbios"
# Vendor flashing is typically done in a maintenance window with nodes
drained
```

1.5 User and permission management

```
[head]% user
[head->user]% add alice
[head->user[alice*]]% set commonname "Alice Researcher"
[head->user[alice*]]% set groupname researchers
[head->user[alice*]]% set password           # interactive
[head->user[alice*]]% commit

# Profiles / RBAC — assign what a user can do in BCM itself
[head]% profile list
[head]% profile use readonly
[head->profile[readonly]]% show
[head]% user use alice
[head->user[alice]]% set profile readonly
[head->user[alice]]% commit
```

Built-in profiles to remember: **admin**, **readonly**, **portal**, **cloudjob**. You can clone and customize them with `profile clone admin custom-admin`.

1.6 Reading baseline metrics with nvidia-smi and dcgmi

`nvidia-smi` is the per-node quick look. `dcgmi` (Data Center GPU Manager) is the cluster-grade tool — it runs `nv-hostengine` as a service and supports persistent health checks, job-level stats, and policy.

```
# nvidia-smi — fast situational awareness
nvidia-smi                                      # default table
nvidia-smi -q                                   # exhaustive details
nvidia-smi -q -d TEMPERATURE,POWER,CLOCK,ECC    # only the sections you need
nvidia-smi --query-
```

```
gpu=index,name,utilization.gpu,memory.used,temperature.gpu \
           --format=csv -l 1                        # CSV, refresh every 1s
nvidia-smi dmon -s pucvmet -d 1                     # device monitor
(power/util/clk/mem/ecc/temp)
nvidia-smi pmon -c 5                                # per-process, 5 samples
nvidia-smi topo -m                                  # GPU/NIC topology matrix
(NVLink, PIX, SYS)
nvidia-smi -pm 1                                    # persistence mode on (keep
driver loaded)
nvidia-smi -i 0 -ac 1215,1410                       # set memory,graphics
application clocks

# dcgmi — cluster-grade health and diagnostics
systemctl status nvidia-dcgm                        # the host engine must be
running
dcgmi discovery -l                                  # list GPUs + entity IDs
dcgmi group -c mygroup --default                    # create a group with all
GPUs
dcgmi group -l                                      # list groups
dcgmi health -g <group_id> -s mpi                   # set health watches
(mem,power,thermal,nvlink...)
dcgmi health -g <group_id> -c                       # check current health
dcgmi diag -r 1                                     # quick diagnostic
(~seconds)
dcgmi diag -r 2                                     # medium (~2 min)
dcgmi diag -r 3                                     # long (~15-30 min, real
workload-like)
dcgmi diag -r 4                                     # extra long, stress-test
grade
dcgmi dmon -e 203,204,250,252 -d 1000               # monitor specific field IDs
every 1000ms
dcgmi stats -g <group_id> -e                        # enable job stats
collection
dcgmi stats -s myjob                                # start tracking job "myjob"
dcgmi stats -x myjob                                # stop and print summary
```

Field IDs worth memorizing: **150** SM clock, **155** Memory clock, **203** GPU util, **204** Memory util, **250** Memory used, **252** Memory free, **1001+** profiling metrics (DCGM Profiling — SM active, tensor active, etc.).

1.7 Practice scenario

> A user reports node005 is in the cluster but jobs aren't landing on it. Walk through the diagnosis.

```
cmsh
[head]% device use node005
[head->device[node005]]% status                     # is it UP / CLOSED / DOWN /
INSTALLER_FAILED?
[head->device[node005]]% get category               # right category?
[head->device[node005]]% get powerstatus
```

```
[head->device[node005]]% latesthealthdata       # BCM health checks
[head->device[node005]]% events                 # recent events for this
node
# If healthy in BCM but Slurm avoids it, check the scheduler (section 3)
```

---

2. Kubernetes for AI Workloads

The NCP-AIO exam tests the NVIDIA GPU Operator stack: **Node Feature Discovery → Driver → Container Toolkit → Device Plugin → DCGM Exporter → MIG Manager → GPU Feature Discovery**. You should be able to install it, inspect each component, and request GPUs in a pod spec.

2.1 Install the GPU Operator

```
# Prereqs: containerd or CRI-O, a running cluster, helm 3+, kubectl
kubectl create ns gpu-operator
kubectl label ns gpu-operator pod-security.kubernetes.io/enforce=privileged

helm repo add nvidia https://helm.ngc.nvidia.com/nvidia
helm repo update

# Most common install: let the operator manage drivers AND toolkit
helm install --wait gpu-operator nvidia/gpu-operator \
  -n gpu-operator \
  --set driver.enabled=true \
  --set toolkit.enabled=true

# If drivers are already on the host, disable the operator's driver
helm install --wait gpu-operator nvidia/gpu-operator \
  -n gpu-operator --set driver.enabled=false
```

2.2 Verify every layer is healthy

```
kubectl get pods -n gpu-operator                # all components
Running/Completed
kubectl get nodes -o json | jq '.items[].status.capacity' | grep nvidia.com
# Expect: "nvidia.com/gpu": "8" (or similar)

kubectl describe node <gpu-node> | grep -A5 -i "nvidia.com"
kubectl get clusterpolicies -n gpu-operator -o yaml | less   # the
operator's CR

# Look at GPU Feature Discovery labels (they drive scheduling)
kubectl get node <gpu-node> -o json | jq '.metadata.labels' | grep nvidia
# Examples: nvidia.com/gpu.product=A100-SXM4-80GB
#           nvidia.com/cuda.driver.major=535
```

```
#           nvidia.com/mig.strategy=single
```

2.3 Request GPUs in a Pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: cuda-smoke-test
spec:
  restartPolicy: OnFailure
  containers:
  - name: cuda
    image: nvcr.io/nvidia/cuda:12.4.1-base-ubuntu22.04
    command: ["nvidia-smi"]
    resources:
      limits:
        nvidia.com/gpu: 1
```

```
kubectl apply -f cuda-smoke-test.yaml
kubectl logs cuda-smoke-test                    # should show the GPU table
```

2.4 MIG with the GPU Operator

MIG is managed by the **MIG Manager** component. Switch a node into a profile by labeling it:

```
# List the available MIG configs from the configmap
kubectl get cm -n gpu-operator default-mig-parted-config -o yaml

# Apply a profile to a node — the MIG Manager will partition the GPU
kubectl label node <gpu-node> nvidia.com/mig.config=all-1g.10gb --overwrite

# Watch the MIG Manager pod do its work
kubectl logs -n gpu-operator -l app=nvidia-mig-manager -f

# After it succeeds, the node advertises sliced resources
kubectl describe node <gpu-node> | grep nvidia.com/mig
# e.g. nvidia.com/mig-1g.10gb: 7
```

In a pod, you request the sliced resource by name:

```yaml
resources:
  limits:
    nvidia.com/mig-1g.10gb: 1
```

Two **MIG strategies** controlled at install time — know both:

- **single** — node exposes one resource type, e.g. `nvidia.com/gpu` mapped to slices.
- **mixed** — node exposes each profile as its own resource (e.g. `nvidia.com/mig-1g.10gb`,

`nvidia.com/mig-2g.20gb`).

2.5 Time-slicing (oversubscribe a GPU without MIG)

```
# ConfigMap consumed by the device plugin
apiVersion: v1
kind: ConfigMap
metadata:
  name: time-slicing-config
  namespace: gpu-operator
data:
  any: |-
    version: v1
    sharing:
      timeSlicing:
        resources:
        - name: nvidia.com/gpu
          replicas: 4
```

```
kubectl apply -f time-slicing-config.yaml
# Tell the cluster policy to use it
kubectl patch clusterpolicy/cluster-policy -n gpu-operator --type merge \
  -p '{"spec": {"devicePlugin": {"config": {"name": "time-slicing-config",
"default": "any"}}}}'
```

Each physical GPU now advertises 4 logical GPUs — useful for inference and dev workloads but **not** isolated like MIG.

2.6 DCGM Exporter and metrics

The DCGM exporter pod scrapes GPU metrics and exposes them on `:9400/metrics` for Prometheus.

```
kubectl get svc -n gpu-operator nvidia-dcgm-exporter
kubectl port-forward -n gpu-operator svc/nvidia-dcgm-exporter 9400:9400
curl localhost:9400/metrics | grep DCGM_FI_DEV_GPU_UTIL
```

2.7 Troubleshooting checklist

```
# Pod stuck Pending — almost always a scheduling/resource issue
kubectl describe pod <name> | tail -30
# Look for: "0/3 nodes are available: 3 Insufficient nvidia.com/gpu"

# Operator components crash-looping
kubectl get pods -n gpu-operator
kubectl logs -n gpu-operator <pod>
kubectl logs -n gpu-operator <pod> --previous     # crashed container
```

```
# Driver pod stuck — check kernel module compile
kubectl logs -n gpu-operator -l app=nvidia-driver-daemonset

# Toolkit not wired into containerd
kubectl logs -n gpu-operator -l app=nvidia-container-toolkit-daemonset
# Verify the runtime config on the host:
cat /etc/containerd/config.toml | grep -A5 nvidia

# Validation pod runs at the end of install — its logs prove the stack works
kubectl logs -n gpu-operator -l app=nvidia-operator-validator -c nvidia-
operator-validator
```

---

3. Slurm & Workload Management

Slurm is the dominant HPC scheduler. The exam tests GPU job scheduling via **GRES**, MIG-aware scheduling, and queue troubleshooting.

3.1 Core commands

```
sinfo                              # partition + node state overview
sinfo -N -l                        # per-node detail
sinfo -o "%P %N %G %C %t"          # partition, nodes, GRES, CPUs, state

squeue                             # current queue
squeue -u alice                    # per user
squeue --start                     # estimated start times

scontrol show node nodeA100-01     # full node detail (GRES, state, reason)
scontrol show job 12345            # full job detail
scontrol show partition gpu

sacct -j 12345 --
format=JobID,State,ExitCode,Elapsed,MaxRSS,ReqTRES,AllocTRES
sacct -S 2026-05-01 -u alice -X  # accounting since a date
sreport cluster utilization start=2026-05-01
```

3.2 Submitting GPU jobs

```
# Interactive single-GPU shell
srun --gres=gpu:1 --pty bash

# Interactive, target a specific GPU model
srun --gres=gpu:a100:2 --pty bash
```

Batch script:

train.sbatch

```
#!/bin/bash
#SBATCH --job-name=resnet
#SBATCH --partition=gpu
#SBATCH --gres=gpu:a100:4
#SBATCH --cpus-per-task=16
#SBATCH --mem=128G
#SBATCH --time=04:00:00
#SBATCH --output=resnet-%j.out

module load cuda/12.4
nvidia-smi
srun python train.py
```

```
sbatch train.sbatch
```

Cancel and inspect:

```
scancel 12345
scancel -u alice                # all jobs by alice
scancel -t PENDING -u alice     # only pending
```

3.3 Configuring GRES for GPUs

Two files matter — both on every compute node and the controller.

`/etc/slurm/gres.conf` (per-node, declares what hardware exists):

gres.conf

```
# /etc/slurm/gres.conf on nodeA100-01
Name=gpu Type=a100 File=/dev/nvidia0 Cores=0-15
Name=gpu Type=a100 File=/dev/nvidia1 Cores=0-15
Name=gpu Type=a100 File=/dev/nvidia2 Cores=16-31
Name=gpu Type=a100 File=/dev/nvidia3 Cores=16-31
```

`/etc/slurm/slurm.conf` (cluster-wide, declares GRES is in use):

slurm.conf

```
GresTypes=gpu,mig
NodeName=nodeA100-[01-08] Gres=gpu:a100:4 CPUs=64 RealMemory=512000
State=UNKNOWN
PartitionName=gpu Nodes=nodeA100-[01-08] Default=YES MaxTime=24:00:00
State=UP
SelectType=select/cons_tres
```

```
SelectTypeParameters=CR_Core_Memory
AccountingStorageTRES=gres/gpu
```

Push and reload:

```
# After editing slurm.conf / gres.conf on the controller, sync to nodes,
# then reconfigure without restart if possible:
scontrol reconfigure
# Heavier changes (new partitions, GresTypes) need:
systemctl restart slurmctld         # on controller
systemctl restart slurmd            # on each node
```

3.4 MIG + Slurm

When MIG is enabled, each slice should appear as its own GRES type so Slurm can schedule against it.

gres.conf (MIG)

```
# gres.conf — one line per slice
Name=gpu Type=1g.10gb File=/dev/nvidia-caps/nvidia-cap21
Name=gpu Type=1g.10gb File=/dev/nvidia-caps/nvidia-cap30
Name=gpu Type=2g.20gb File=/dev/nvidia-caps/nvidia-cap39
```

slurm.conf (MIG)

```
NodeName=mig-node-01 Gres=gpu:1g.10gb:6,gpu:2g.20gb:1 ...
```

Submit against a slice:

```
srun --gres=gpu:1g.10gb:1 --pty nvidia-smi -L
```

The actual `/dev/nvidia-caps/` paths come from `ls /proc/driver/nvidia/capabilities/mig/` after MIG is enabled with `nvidia-smi mig -cgi … -C`.

3.5 Why is my job stuck? The reason codes

```
squeue --format="%.10i %.9P %.20j %.8u %.2t %.10M %.6D %R"
# The last column is the REASON. Common ones:
#   Resources             - waiting for nodes (normal queueing)
#   Priority              - other higher-priority jobs ahead
#   ReqNodeNotAvail       - requested node is down/drained
#   AssocGrpGRESLimit     - hit a per-association GRES quota
#   QOSMaxGRESPerUser     - QoS limit
#   PartitionTimeLimit    - asked for more time than partition allows
```

```
#   Dependency            - waiting on another job
#   InvalidQOS            - QoS not granted to user
#   ReqGresTypeNotAvail   - asked for gpu:h100 when only a100 exists
```

When a node is DRAIN or DOWN:

```
sinfo -R                                # nodes in drain + the reason
scontrol show node nodeA100-03 | grep -E "State|Reason"
scontrol update NodeName=nodeA100-03 State=RESUME   # bring it back after fixing
scontrol update NodeName=nodeA100-03 State=DRAIN Reason="bad GPU"
```

3.6 Practice scenario

> A 4-GPU H100 job sits in PD with reason `Resources` forever, even though `sinfo` shows idle H100 nodes. Diagnose.

```
# 1. Confirm the job actually requested the right TYPE
scontrol show job <id> | grep -E "TresPerNode|Gres"

# 2. Confirm Slurm knows the nodes have h100 GRES (not just "gpu")
scontrol show node nodeH100-01 | grep -E "Gres|CfgTRES"
# If you see Gres=gpu:4 (no type), gres.conf is missing Type=h100

# 3. Check for QoS / association limits eating the job
sacctmgr show assoc user=$USER format=Account,User,QOS,GrpTRES,MaxTRES
sacctmgr show qos format=Name,GrpTRES,MaxTRESPerUser

# 4. Check for reservations blocking the partition
scontrol show reservation
```

---

4. System Troubleshooting & Optimization

The exam puts you in front of a broken cluster and asks for the **next command**. Build a mental flowchart: GPU → driver → container runtime → scheduler → network.

4.1 GPU and driver layer

```
# Is the hardware visible?
lspci | grep -i nvidia

# Is the driver loaded?
lsmod | grep nvidia
nvidia-smi                              # if this fails, no driver / wrong driver
```

```
# Kernel messages — ECC errors, Xid events, fallen-off-bus, thermal
dmesg -T | grep -iE "nvidia|nvrm|xid"
journalctl -k --since "1 hour ago" | grep -i nvidia

# Xid codes you should know on sight:
#   13   - Graphics Engine Exception (often app bug or bad memory)
#   31   - GPU memory page fault (illegal address in CUDA code)
#   43   - Reset channel verif error
#   48   - Double-bit ECC error (uncorrectable, GPU usually needs reset)
#   63/64 - Row-remapper recording/failure (HBM page retirement)
#   74   - NVLink error
#   79   - GPU fell off the bus (PCIe/power — often hardware)
#   92   - High single-bit ECC (correctable but worth watching)
#   94/95 - Contained/uncontained ECC error
#   119  - GSP RPC timeout
nvidia-smi -q -d ECC,PAGE_RETIREMENT,REMAPPED_ROWS

# Generate a full bug report (attach to support tickets)
nvidia-bug-report.sh
```

4.2 DCGM diagnostics

`dcgmi diag` is the canonical "is this GPU healthy" tool — run it before blaming software.

```
dcgmi diag -r 1            # ~seconds, software checks
dcgmi diag -r 2            # ~2 min, includes targeted stress
dcgmi diag -r 3            # ~30 min, sustained workloads (memory bw, SM
stress, NVLink)
dcgmi diag -r 4            # extended, hours

# Per-GPU policy alerts (set once, leave running)
dcgmi policy --set 0,0 -p 250 -T 95 -M 5   # warn on power>250W, temp>95C,
mem errors
dcgmi policy --get -g 0
```

If `dcgmi diag` flags a GPU, capture the JSON:

```
dcgmi diag -r 3 -j > diag.json
```

4.3 Container runtime problems

```
# Does the toolkit see the GPU?
nvidia-container-cli info
nvidia-container-cli list

# Containerd config wiring
crictl info | jq '.config.containerd.runtimes'
```

```
cat /etc/containerd/config.toml | grep -A3 nvidia

# Run a known-good test container outside of k8s to isolate
ctr run --rm --gpus 0 \
  docker.io/nvidia/cuda:12.4.1-base-ubuntu22.04 smoke nvidia-smi
```

For Docker:

```
docker run --rm --gpus all nvcr.io/nvidia/cuda:12.4.1-base-ubuntu22.04
nvidia-smi
```

If this fails but `nvidia-smi` on the host works, the **container toolkit** (not the driver) is the problem.

4.4 Performance bottleneck triage

When a job runs but is slow, walk the stack:

```
# 1. Is the GPU actually busy?
nvidia-smi dmon -s u -d 1
# sm/mem util both low -> CPU/IO bound; mem high, sm low -> memory bound

# 2. Are clocks being throttled? Look at the THROTTLE reasons.
nvidia-smi -q -d PERFORMANCE
# Watch for: HW Slowdown=Active, SW Thermal Slowdown, Power Brake

# 3. Power and thermal headroom
nvidia-smi --query-gpu=power.draw,power.limit,temperature.gpu --format=csv -
l 1

# 4. Memory pressure / page retirement
nvidia-smi -q -d PAGE_RETIREMENT,REMAPPED_ROWS,ECC

# 5. NVLink / PCIe link health
nvidia-smi nvlink -s                 # state of each link
nvidia-smi nvlink -e                 # error counters
nvidia-smi -q -d PIDS,CLOCK          # processes and current clocks
lspci -s <bdf> -vvv | grep -i "lnksta"  # PCIe gen + width — is it at
advertised speed?

# 6. Topology — wrong GPU-to-NIC binding kills multinode perf
nvidia-smi topo -m
ibstat                               # IB ports, rates
ibdev2netdev                         # IB device <-> netdev mapping
```

4.5 Network layer (RDMA / InfiniBand / NCCL)

```
ibstat                               # link state, rate, port GUID
ibstatus
```

```
iblinkinfo                              # full fabric link summary
ibping -S; ibping -G <guid>             # one node as server, another pings
perfquery                               # port counters; rerun for deltas
ibdiagnet                               # subnet diagnosis (run on a single
node)

# NCCL — almost every multi-GPU AI failure shows up here
NCCL_DEBUG=INFO mpirun -np 8 ./my_nccl_test
# Important env vars to know:
#   NCCL_IB_HCA           - which HCAs to use
#   NCCL_SOCKET_IFNAME    - which Ethernet IFs to allow
#   NCCL_P2P_DISABLE      - disable peer-to-peer (debug only)
#   NCCL_IB_DISABLE       - force TCP fallback (debug only)
#   NCCL_TOPO_DUMP_FILE   - dump the topology NCCL discovered

# Standard health test
/opt/nccl-tests/build/all_reduce_perf -b 8 -e 8G -f 2 -g 8
```

4.6 Scheduling and capacity failures

```
# K8s: pod won't schedule on GPU node
kubectl describe pod <name>          # bottom Events section is the answer
kubectl get events -A --sort-by='.lastTimestamp' | tail -30

# Slurm: queue full but nodes "look" idle
sinfo -R                             # any drained nodes?
scontrol show node <node> | grep -E "Reason|State|Gres"
sdiag                                # scheduler internals - cycle times,
backlog
```

4.7 BCM-level health checks

```
cmsh
[head]% monitoring
[head->monitoring]% measurable list          # available metrics
[head->monitoring]% healthcheck list         # built-in checks (mounts,
gpu, ib, ...)
[head->monitoring]% latesthealthdata         # current health snapshot

[head]% device latesthealthdata -c gpu-nodes
[head]% events                               # cluster-wide event stream
```

4.8 The triage flowchart to memorize

Symptom	First command	Then
`nvidia-smi: command not found`	`which nvidia-smi`	Is driver installed?
“No devices were found”	`lsmod | grep nvidia`	`dmesg | grep -i nvidia`

Symptom	First command	Then
GPU shows in lspci, not nvidia-smi	`dmesg -T | grep Xid`	Driver / module / fallen off bus
Container can't see GPU	`nvidia-container-cli info`	Toolkit + runtime config
Pod Pending	`kubectl describe pod`	Resources / labels / taints
Slurm job PD forever	`squeue` + `scontrol show job`	GRES type? QoS? Reservation?
Job runs but slow	`nvidia-smi dmon`	Throttling reasons + topology
Multi-node training slow	`NCCL_DEBUG=INFO`	IB link state, topo, GPU↔NIC
ECC errors / Xid 48/63/64	`nvidia-smi -q -d ECC`	Drain node, page retirement

Final exam prep tips

- **Practice in cmsh blind.** The web UI is not what's being tested. Build muscle memory for `device`, `category`, `softwareimage`, `user`, `monitoring`, and `commit`.
- **Know the difference** between `imageupdate` (live sync) and `reinstall` (full PXE) — that's a classic question.
- **Memorize the GPU Operator components in order** and what each one does. If one is broken, what symptom appears?
- **Know MIG strategies** (`single` vs `mixed`) and how to request slices in both Kubernetes (`nvidia.com/mig-1g.10gb`) and Slurm (`--gres=gpu:1g.10gb:1`).
- **Recognize Xid codes** at least for 13, 31, 48, 63, 79, 94/95.
- **`dcgmi diag -r {1,2,3,4}`** levels — the time and depth differences come up.
- **Slurm reason codes** — `Resources`, `Priority`, `ReqNodeNotAvail`, `AssocGrpGRESLimit`, `QOSMaxGRESPerUser`, `ReqGresTypeNotAvail`.
- **NCCL env vars** — `NCCL_DEBUG`, `NCCL_IB_HCA`, `NCCL_SOCKET_IFNAME`.

Good luck — build the labs, break them on purpose, fix them, repeat.