Elastic HPC and AI Cloud Orchestration

Bridging Traditional HPC Workflows with Cloud-Native Kubernetes Infrastructure

Rajesh Narayanan | Portland, OR | Engineering Leader in AI & HPC Infrastructure

About Rajesh Narayanan

Rajesh Narayanan (Portland, OR): An engineering leader with 25+ years of experience across Cloud, HPC, AI Infrastructure, and Semiconductor Systems. Formerly an Engineering Manager at Intel, he specialized in scalable design and technical leadership for AI & HPC workloads.

Led the delivery of the **Aurora Supercomputer** at Argonne National Laboratory and the **Intel Tiber AI Developer Cloud**.

Connect

linkedin.com/in/rajesh-n-268ba4

Key Expertise

- Kubernetes orchestration
- Ansible automation
- Prometheus monitoring
- VoltDB analytics
- Slurm scheduling
- Embedded software development
- Telemetry systems

Project Highlights

- **Intel Tiber AI Cloud:** 25K+ customers onboardings, hybrid GPU orchestration.
- **Aurora Supercomputer:** >1 ExaFLOPS, 10,624 nodes, 21,248 CPUs, 63,744 GPUs.
- **HPC Infrastructure:** Kubernetes orchestration, AIML enablement.
- **Telemetry System:** Prometheus + VoltDB, Rapid MQ, petabyte/year anomaly detection.

The Challenge: Traditional HPC Limitations

Lack of Elasticity

Traditional HPC clusters had fixed resource allocation, lacking dynamic scalability. Resources idled during low utilization; jobs queued during peak.

Poor Resource Efficiency

Static provisioning resulted in over- or underprovisioning, creating bottlenecks. Suboptimal GPU utilization meant expensive hardware was underutilized.

Integration Gaps

HPC and AI workloads were siloed, using separate orchestration systems. No unified platform connected traditional batch scheduling with cloud-native container orchestration.

Strategic Goal: Integrate HPC batch workloads and AI training jobs into a unified, elastic, cloud-native infrastructure that delivers both reliability and performance at scale.

Solution: Elastic Kubernetes Pods for HPC

Architecture Overview

Our solution introduces **Elastic Kubernetes Pods (EKP)** — a hybrid orchestration model that enables Slurm workload manager to dynamically provision Kubernetes pods as compute nodes. Instead of waiting for bare-metal or VM provisioning, Slurm launches containerized compute pods on-demand, executes jobs, and tears down resources automatically.

Key Innovation: This approach merges HPC-style batch scheduling with elastic, container-based resource management, delivering cloud-native benefits to traditional HPC workflows.

01

User submits job via Jupyter or CLI

02

Slurm Controller receives request

03

Kubernetes provisions compute pods

04

Pods execute workload with slurmd

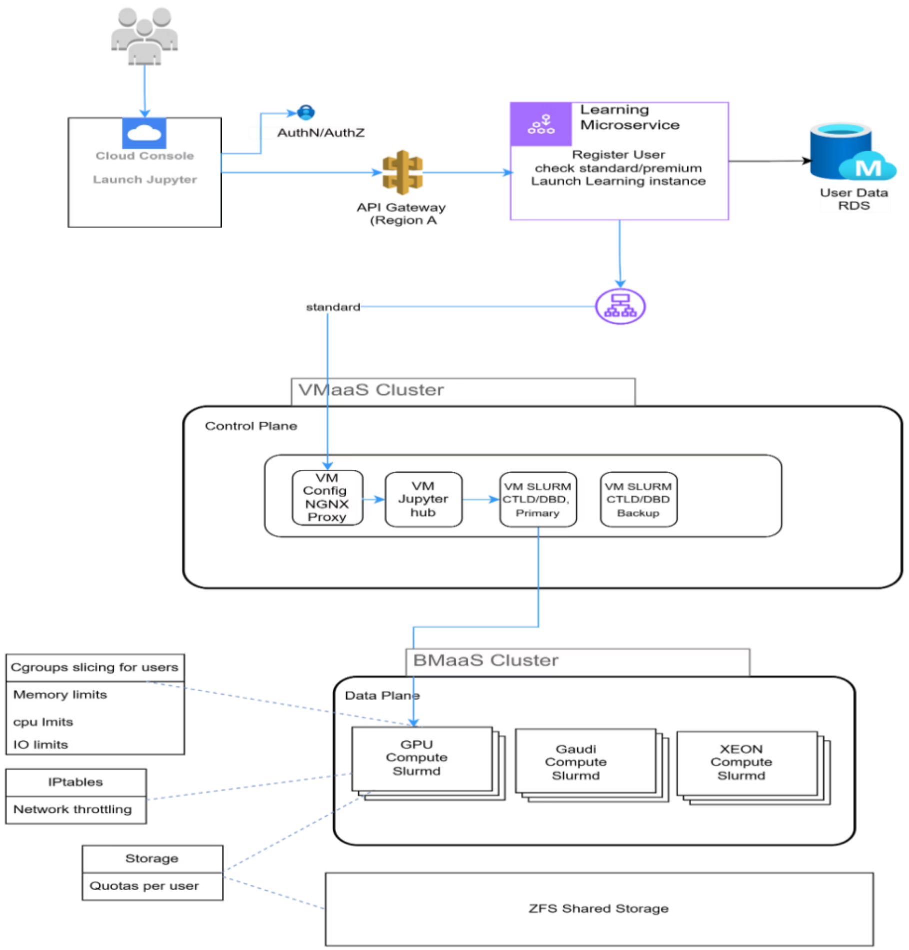
05

Resources terminate post-completion

Technical Specifications: 50 compute nodes | 400 GPUs | 200 Gbps network fabric | 1TB RAM per node | 128GB HBM per GPU | NFS/ZFS shared volumes | Support for 2 parallel jobs per GPU | 8 HPC parallel jobs per user

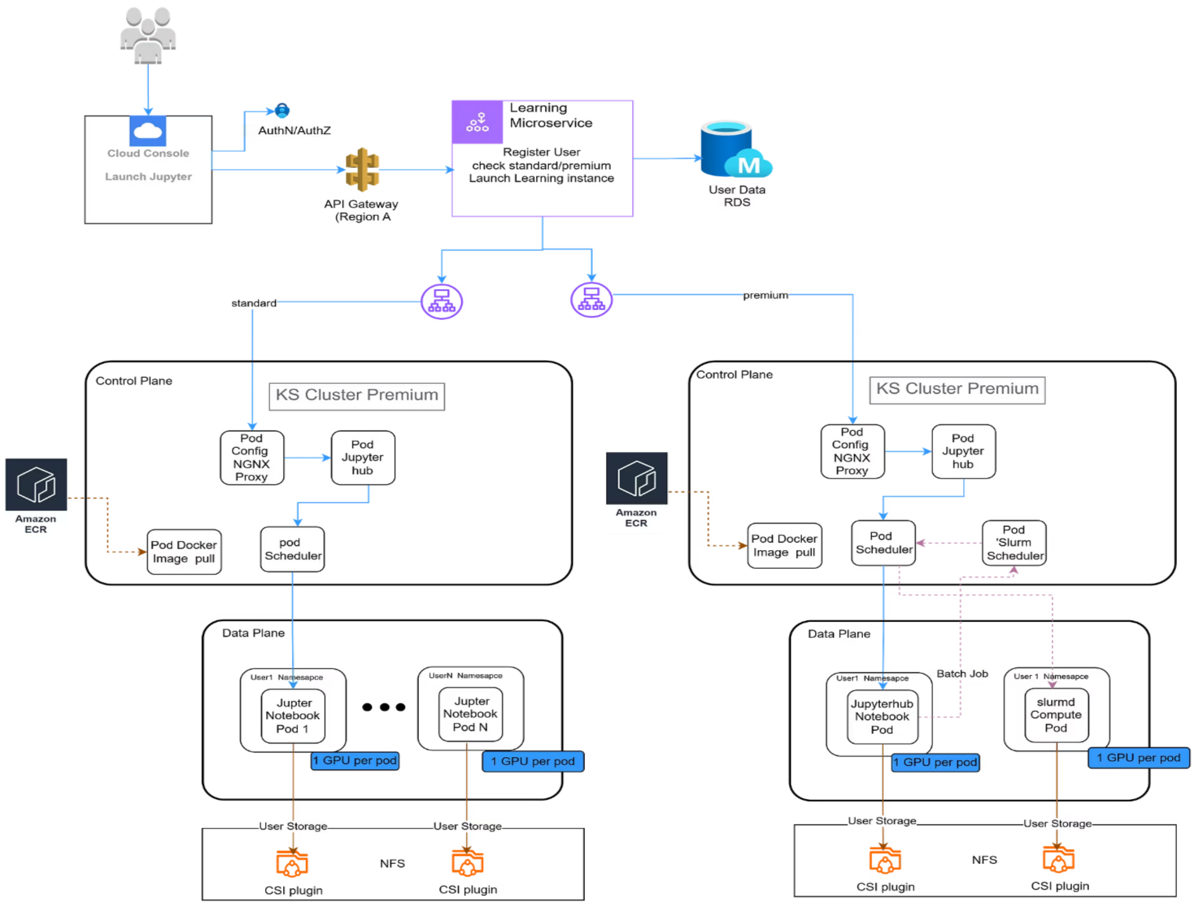
Architecture Evolution: From Bare Metal to Elastic Pods

Previous State



Traditional architecture relied on bare-metal servers with static provisioning. Jobs waited in queue for physical resources to become available, leading to poor utilization and extended wait times.

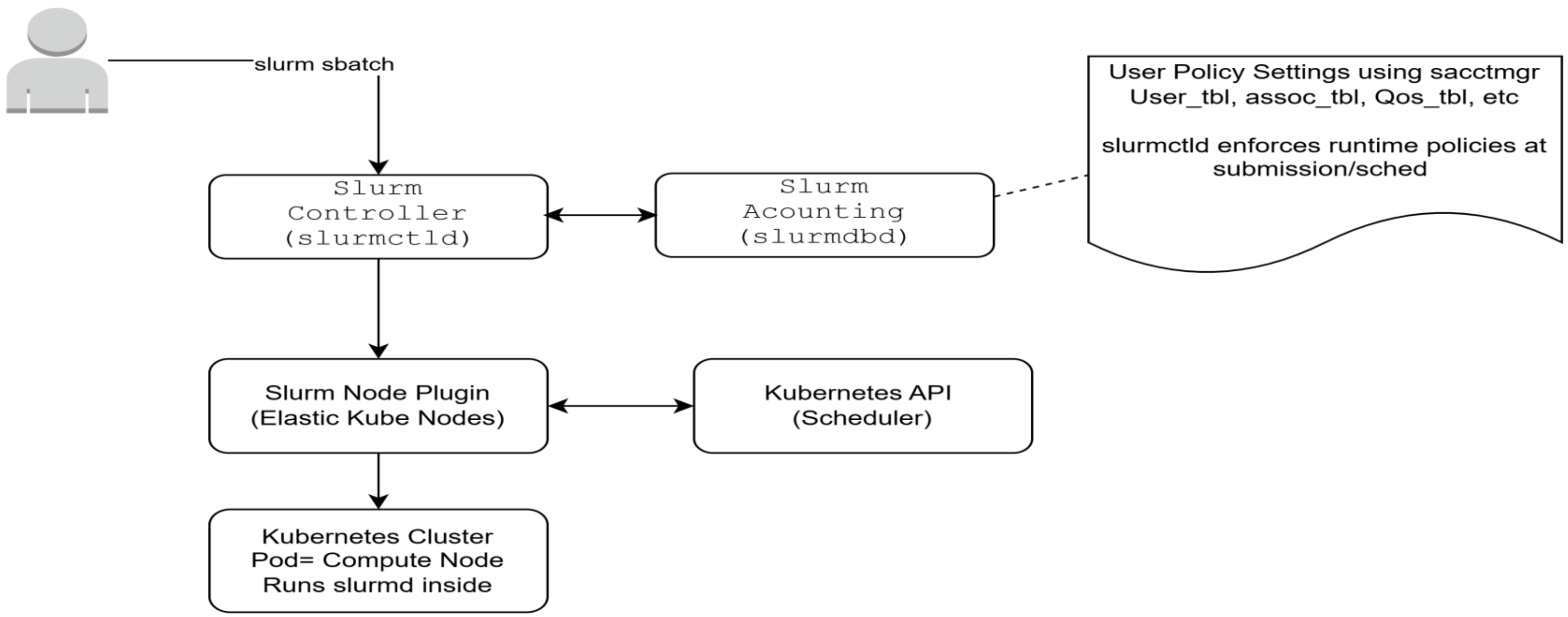
Current State with EKP



Modern elastic architecture leverages Kubernetes pod orchestration. Compute resources provision dynamically in seconds, scale based on demand, and terminate automatically, maximizing efficiency and throughput.

Transformation Impact: Migration from static bare-metal provisioning to dynamic pod orchestration reduced job start times by 50% while improving GPU utilization by 3×.

System Architecture: Component Interaction



Slurm Controller

Manages job queues, triggers elastic provisioning, communicates with Kubernetes, monitors jobs.

Kubernetes Cluster

Schedules compute pods, provides elastic scaling, manages pod lifecycle and cleanup.

Elastic Compute Pods

Run slurmd daemon, register as compute nodes, execute workloads, terminate post-job.

Each pod runs a lightweight **slurmd** daemon, dynamically joining the Slurm cluster as a compute node capable of executing MPI, GPU, and HPC workloads, terminating automatically post-job.

Implementation: Configuration Highlights

Step 1: Build or pull a Slurm Pod Image

Create container image with **slurmd**, **munge** authentication, and minimal dependencies. Store in ECR image repository for deployment. Configure **slurm.conf** to point to the controller.

Step 2: Create Kubernetes Service Account and RBAC

Give Slurm permission to create pods using the following Kubernetes YAML configurations:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
name: slurm-elastic

rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["create", "delete", "list", "get"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: slurm-elastic-binding
roleRef:
  kind: Role
  name: slurm-elastic
  apiGroup: rbac.authorization.k8s.io
subjects:
- kind: ServiceAccount
  name: slurm
  namespace: default
```

Step 3: Configure slurm.conf for Elastic Nodes

Update **slurm.conf** with elastic node definitions. Add the following example snippet:

```
# slurm.conf snippet for elastic nodes
NodeName=elastic[1-100] State=CLOUD
PartitionName=ekp Nodes=elastic[1-100] Default=YES
MaxTime=INFINITE State=UP
```

Step 4: Add Slurm Prolog Plugin

Configure **slurm.conf** to use a Prolog script for dynamic pod provisioning. Example:

```
# slurm.conf snippet for prologProlog=/etc/slurm/prolog.sh
```

Example **prolog.sh** script (simplified):

```
#!/bin/bash
# Example prolog.sh script
NODELIST=$SLURM_JOB_NODELIST
# Logic to call kubectl to provision pods for NODELIST
# e.g., kubectl apply -f pod-template.yaml --selector=slurm.node=$NODELIST
```

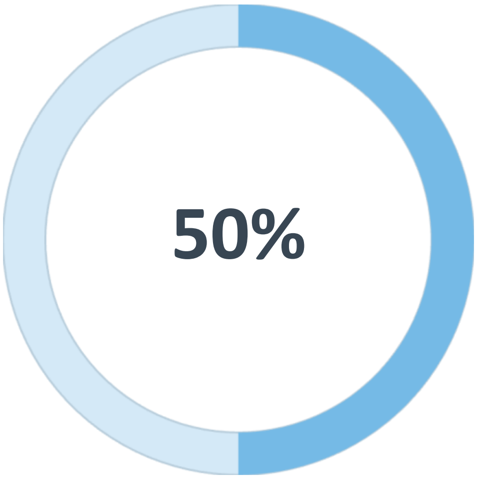
Step 5: Test Job Submission

Submit a test job via **sbatch** and verify successful execution on elastic nodes.

```
# Example sbatch command
sbatch --nodes=2 --wrap="hostname"
```

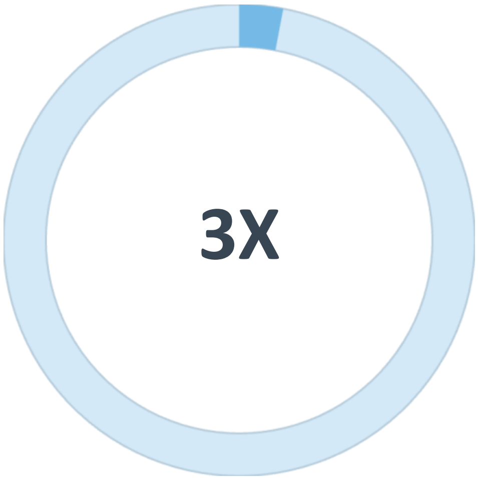
Verify pod creation in Kubernetes, **slurmd** registration, workload execution, and automatic pod termination after job completion.

Results and Business Impact



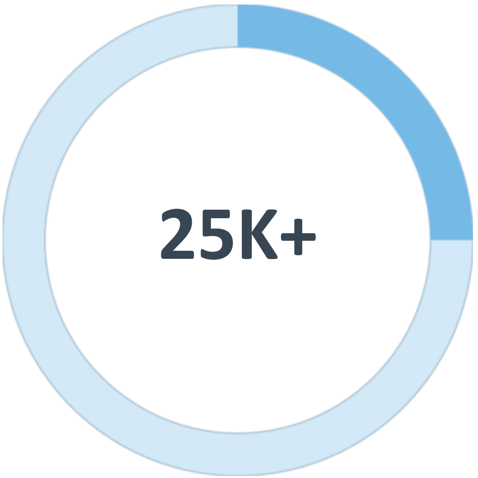
Faster Job Start

Reduced provisioning time from minutes to seconds with dynamic pod creation



GPU Utilization

Improved resource efficiency through elastic scaling and auto-termination



Active Users

Successfully scaled Intel Tiber AI Cloud to support enterprise workloads

Technical Achievements

- Unified orchestration for HPC batch jobs and AI training workloads
- Auto-scaling compute infrastructure based on real-time demand
- Hybrid cloud capabilities enabling multi-region deployments
- Seamless integration with existing Slurm-based workflows

Operational Benefits

- Reduced infrastructure costs through efficient resource utilization
- Improved time-to-science for researchers and data scientists
- Enhanced system reliability with container-based isolation
- Simplified operations with Kubernetes-native tooling

Alternative Approach: Native Slurm Operator

Kubernetes-Native Slurm Deployment

An alternative architecture involves running Slurm entirely inside Kubernetes using the [SchedMD Slurm Operator](). This approach handles the complete lifecycle of controllers and compute pods through Kubernetes Custom Resource Definitions (CRDs).

Advantages: Simpler for new deployments, leverages Kubernetes-native constructs, reduces external dependencies, provides declarative job definitions.

Trade-offs: Locks architecture to Slurm scheduler, limiting future flexibility to adopt alternative schedulers. Less suitable for organizations with existing Slurm infrastructure requiring gradual migration.

```
apiVersion: slurm.schedmd.com/v1alpha1
kind: SlurmJob
metadata:
  name: mpi-testspec:
  partition: default
  tasks: 4
   image: ghcr.io/schedmd/slurm:latest  command: ["mpirun",
"hostname"]
```

Best Use Case: Greenfield deployments starting fresh with Kubernetes-first architecture and no legacy Slurm infrastructure to maintain.

Case Study: Aurora Supercomputer at Argonne

System Architecture

Exascale compute system delivered through Intel + HPE + DOE partnership, achieving breakthrough performance milestones for scientific research and national security applications.

10.6K

Compute Nodes

63.7K

GPUs

1.012

ExaFLOPS

FP64 Linpack

Advanced Capabilities

- **Power Management:** GEOPM framework for power-aware scheduling under 20 MW budget with real-time energy calculations
- **Storage Innovation:** DAOS Object Store with burst buffers and pre-fetching libraries for extreme I/O performance
- **Multi-OS Architecture:** Hybrid kernel design (fat + lightweight) optimized for extreme-scale computing workloads
- **Telemetry System:** Prometheus + VoltDB infrastructure sampling every 10 minutes (716KB/node, ~1.6 PB over 4 years)
- **Predictive Analytics:** Health modeling and reliability optimization using machine learning on telemetry data

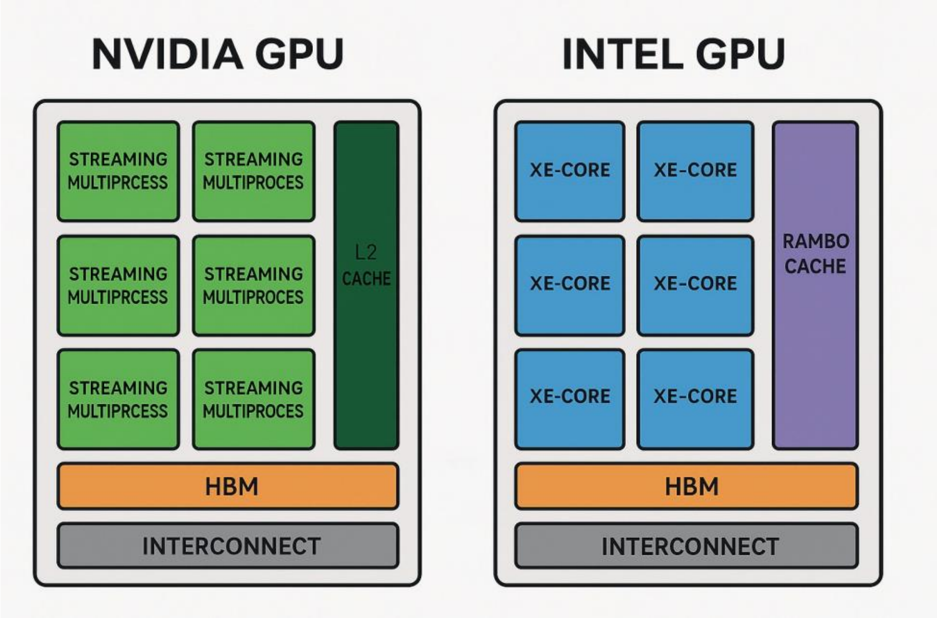
Backup Slides

GPU Driver Task Isolation

Overview of mechanisms used by GPU drivers to securely and efficiently isolate user jobs on shared GPUs

Overview

- Modern GPU clusters often share one physical GPU among many jobs.
- Isolation prevents one job from interfering with another in terms of security, performance, or stability.
- Drivers and hardware enforce this with memory protection, context switching, and partitioning.



Key Isolation Mechanisms

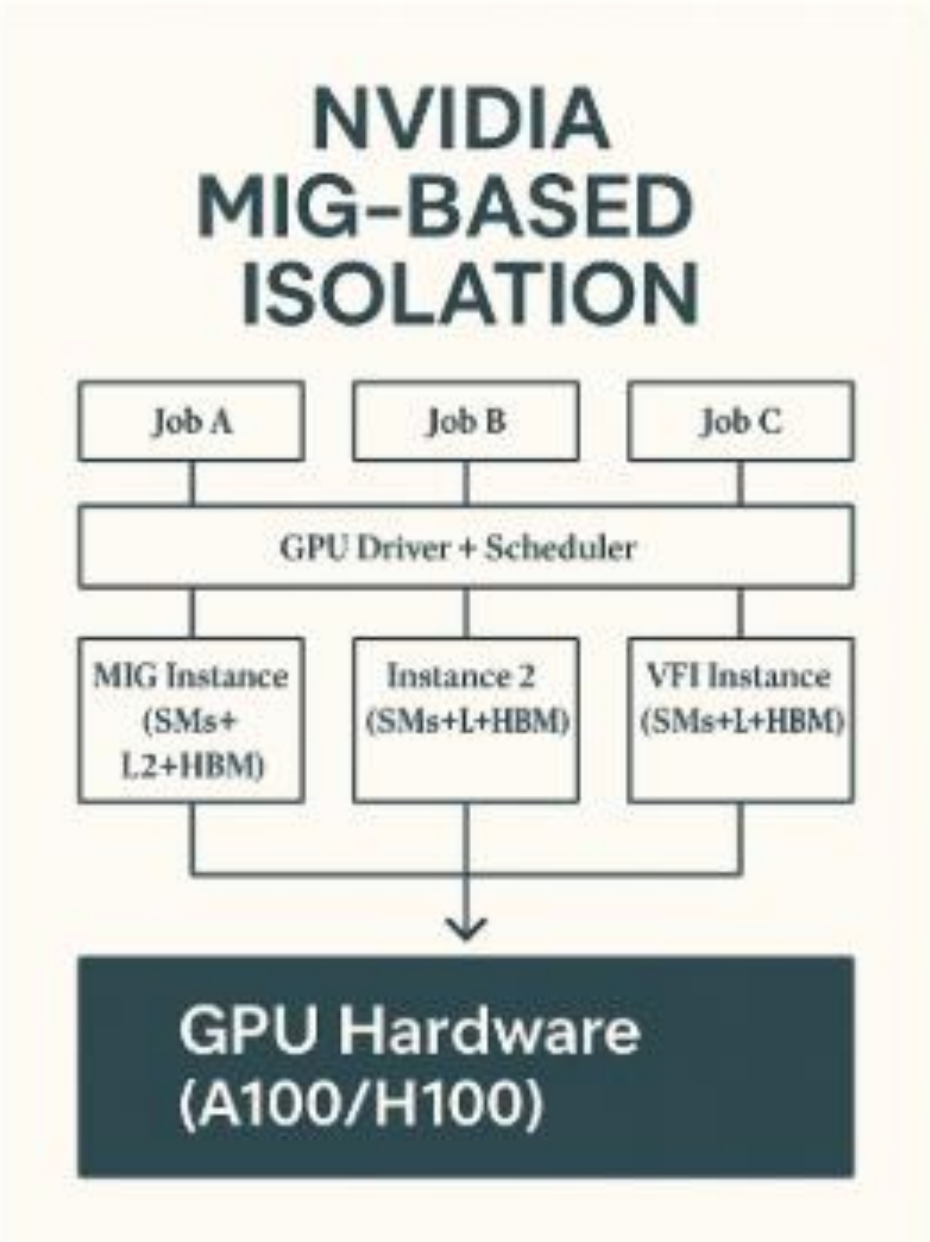
1. Context Management – per-job GPU context (registers, queues, state)
2. Virtual Memory – per-context page tables with IOMMU/SMMU
3. Scheduling & Partitioning – time-slicing and hardware partitioning (MIG, SR-IOV)
4. Command Queue Isolation – separate, validated queues
5. Preemption & Fault Containment – contain faulty kernels
6. Host-Level Controls – containers, cgroups, schedulers restrict GPU access

Common Flow of GPU Isolation

1. Scheduler assigns a GPU partition or VF to the job
2. Driver creates GPU context and maps memory
3. Commands submitted to isolated queues
4. GPU hardware enforces memory & execution isolation
5. Preemption and error handling contain faults per job

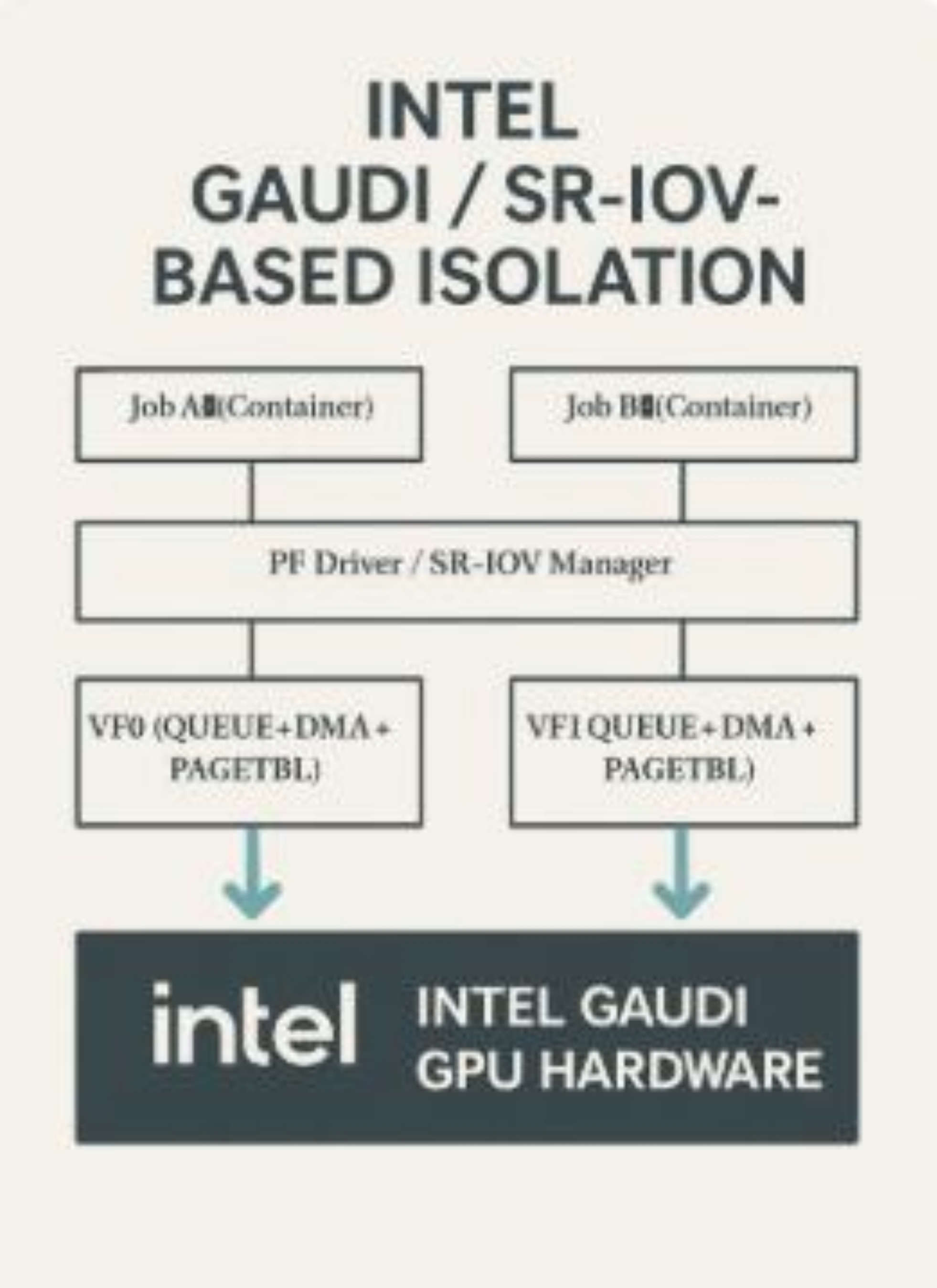
NVIDIA MIG-Based Isolation

- NVIDIA Multi-Instance GPU (MIG) partitions GPU into hardware-isolated slices.
- Each slice has dedicated SMs, L2 cache, and memory.
- Each job's context binds to one MIG slice, ensuring secure and predictable workload separation.

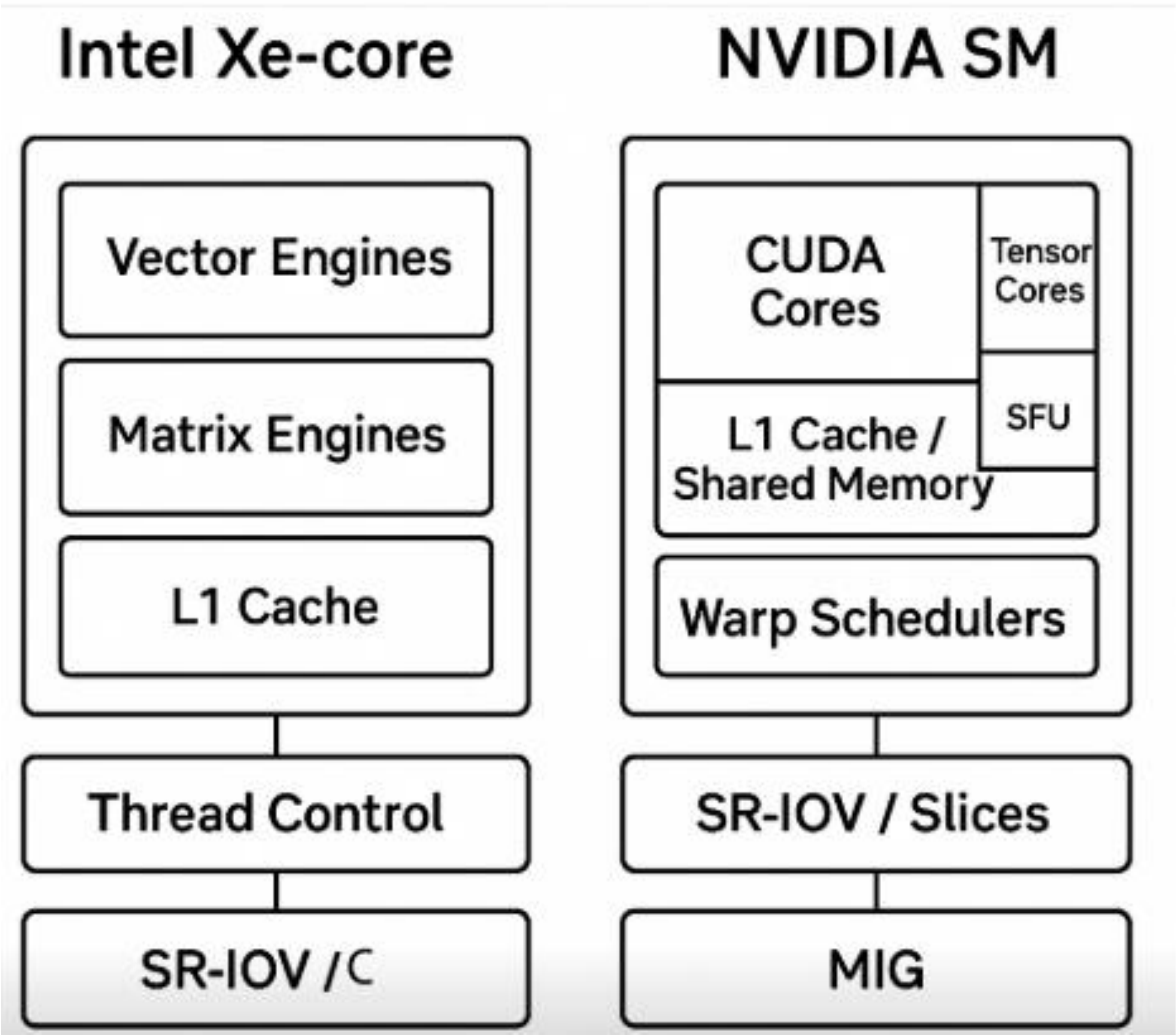

NVIDIA
MIG-BASED
ISOLATION
Job A
Job B
Job C
GPU Driver + Scheduler
MIG Instance (SMs+ L2+HBM)
Instance 2 (SMs+L+HBM)
VFI Instance (SMs+L+HBM)
GPU Hardware (A100/H100)

Intel Gaudi / SR-IOV-Based Isolation

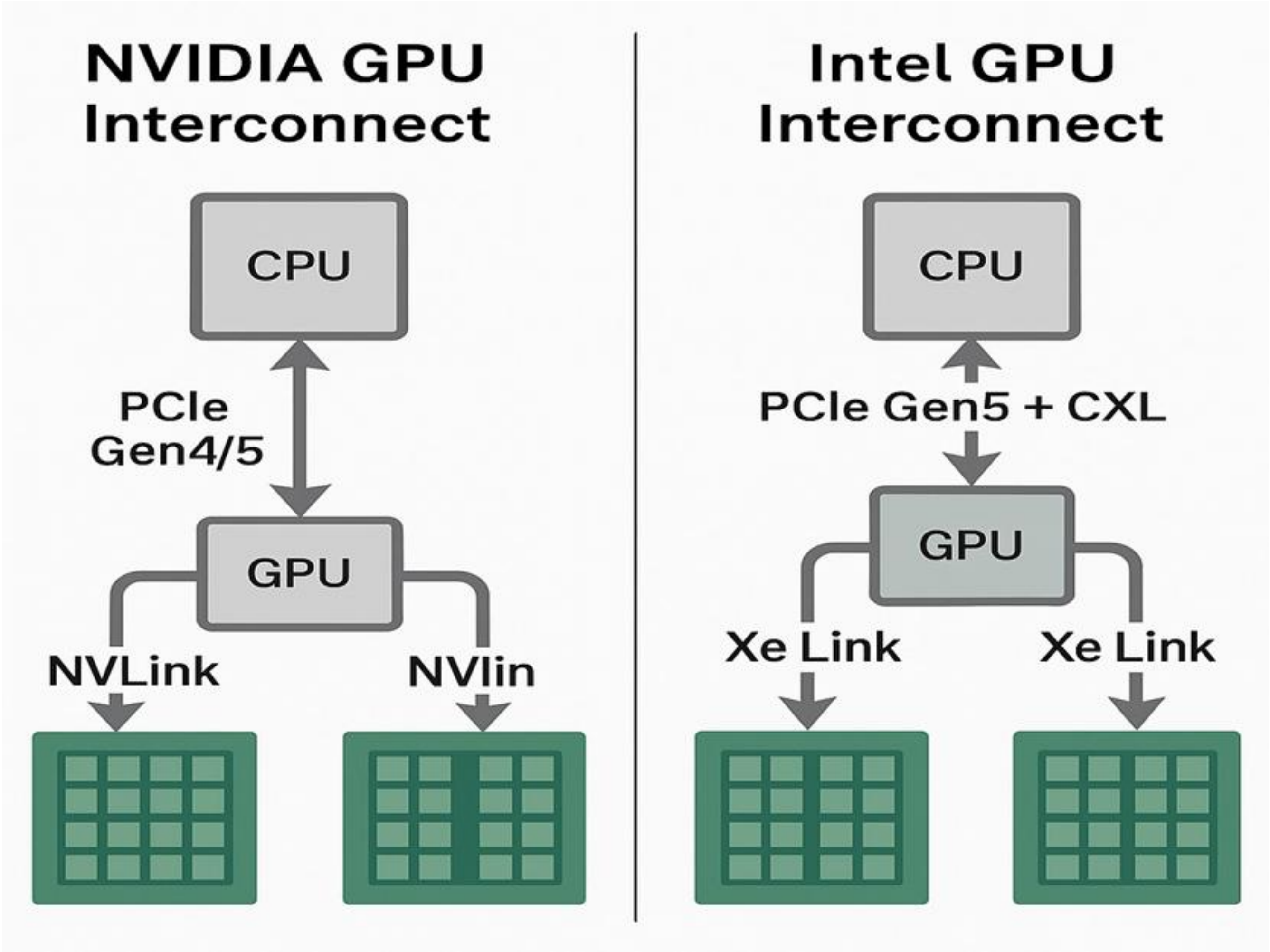
- Intel Gaudi GPUs use PCIe SR-IOV to expose multiple Virtual Functions (VFs).
- Each VF has its own queue, DMA engine, and page tables.
- Jobs in containers or VMs bind to separate VFs, managed by the PF driver for secure isolation.


INTEL
GAUDI / SR-IOV-
BASED ISOLATION
Job A (Container)
Job B (Container)
PF Driver / SR-IOV Manager
VF0 (QUEUE+DMA + PAGETBL)
VF1 QUEUE+DMA + PAGETBL)
intel
INTEL GAUDI
GPU HARDWARE

GPU Core Architecture Comparison



Interconnect Architecture Comparison



Appendix: Acronyms

- GPU: Graphics Processing Unit – hardware accelerator for compute and graphics
- SM: Streaming Multiprocessor – core compute unit in NVIDIA GPUs
- HBM: High-Bandwidth Memory – fast on-package memory for GPUs
- MIG: Multi-Instance GPU – NVIDIA feature that partitions a GPU into isolated slices
- SR-IOV: Single Root I/O Virtualization – PCIe technology to virtualize a device into multiple functions
- VF: Virtual Function – virtualized GPU instance presented to a VM or container
- PF: Physical Function – host-level GPU function managing all VFs
- IOMMU / SMMU: I/O Memory Management Unit / System MMU – hardware for memory protection of DMA
- DMA: Direct Memory Access – allows GPU to access host or device memory directly
- RAS: Reliability, Availability, and Serviceability – hardware error detection and reporting framework
- SMT / Preemption: Simultaneous Multithreading / Preemption – techniques for scheduling and isolating GPU workloads