Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Summer School on Supercomputing & AI for Tech Entrepreneurs
29-31 August 2025 Namal University Mianwali

HPC vs Cloud-Native AI Application Deployment: Approaches, Strengths, and Hybrid Strategies

Prof. Dr. Tassadaq Hussain
Specialization: Supercomputing and Artificial Intelligence
Director Centre for AI and BigData
Affiliated Member: Barcelona Supercomputing Centre

Agenda

- **Importance of Cloud Computing**
- Side-by-Side Cloud Software Stack Mapping (Same Hardware)
- Kubernetes: Hardware and Software Management
- Cloud-Native Application Deployment (vs Bare Metal)
- Developing Distributed Applications on Kubernetes
- Comparison: Bare Metal vs Cloud: Gains and Overheads
- Hybrid Cluster Strategy: Combining HPC and Cloud-Native

HPC and Cloud

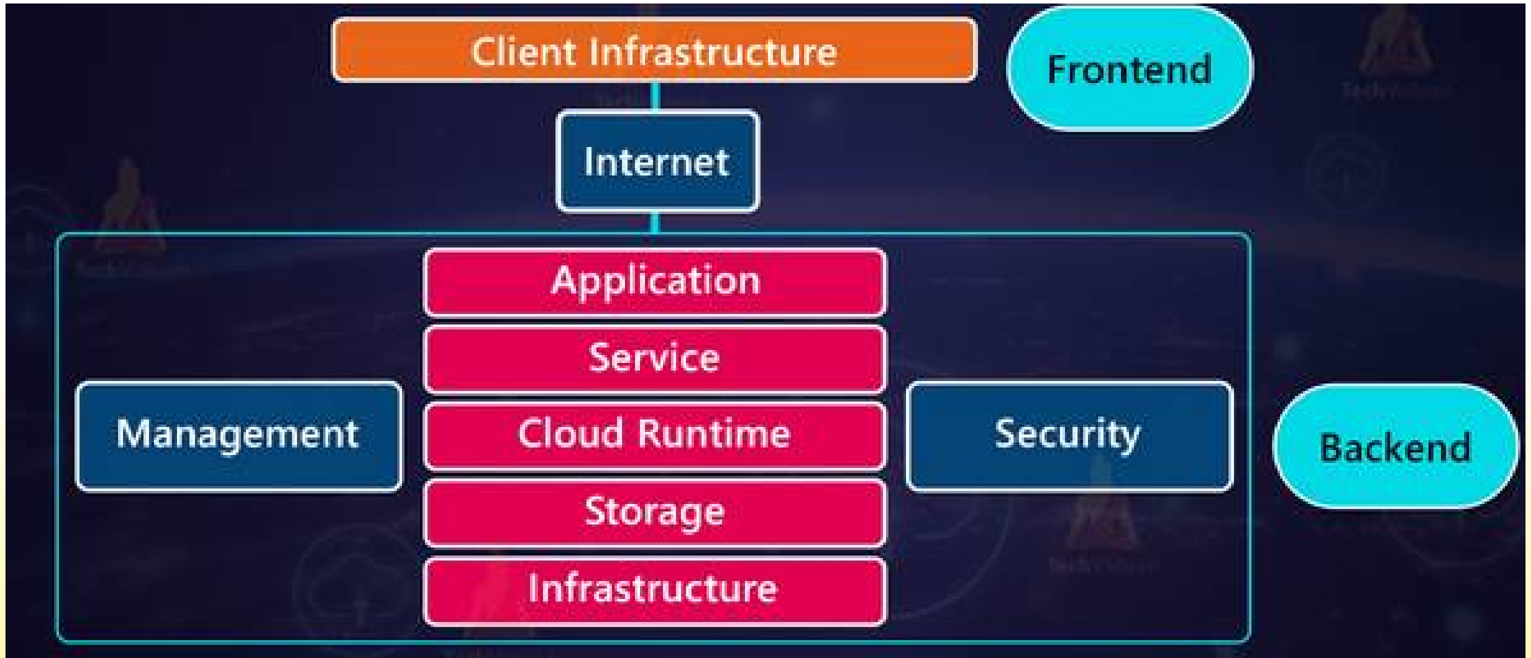
- HPC, is a bare-metal clusters optimized for maximum performance.
- Cloud computing offers flexible and elastic infrastructure — but with different overheads and trade-offs.
- Let’s explore how these two approaches compare.

What is Cloud Computing?

Cloud computing is the on-demand delivery of computing services (servers, storage, networking, databases, software, AI platforms, etc.) over the internet, with pay-as-you-go pricing.

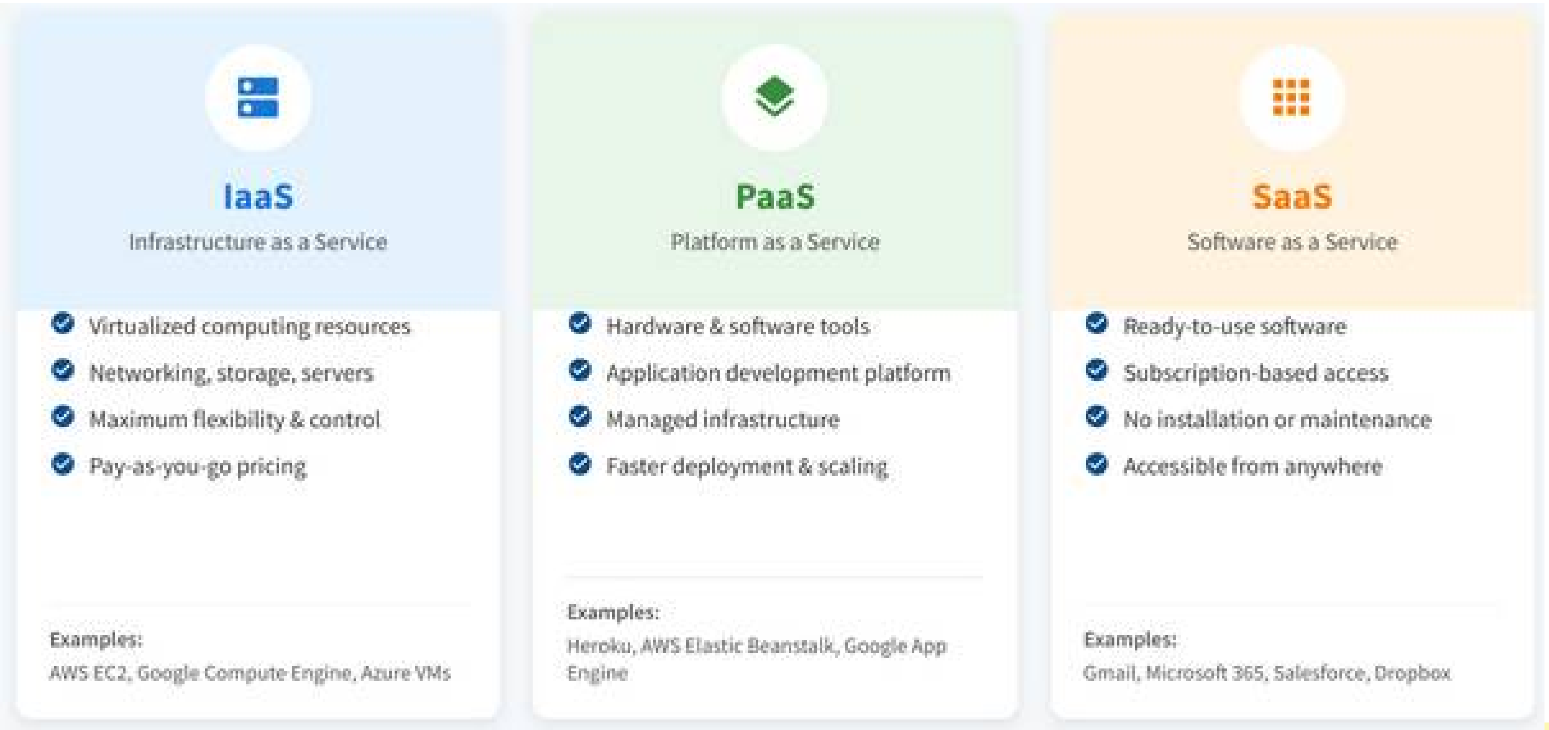
- Instead of owning and maintaining physical data centers or servers, users rent computing resources from cloud providers like AWS, Microsoft Azure, or Google Cloud.

Scalability, flexibility, global access, reduced upfront cost, and rapid innovation.



Cloud Service Models

- IaaS (Infrastructure as a Service) – raw computing power (VMs, storage, networking).
- PaaS (Platform as a Service) – managed environments for app development & deployment.
- SaaS (Software as a Service) – complete applications delivered over the internet.



IaaS
Infrastructure as a Service

- ✓ Virtualized computing resources
- ✓ Networking, storage, servers
- ✓ Maximum flexibility & control
- ✓ Pay-as-you-go pricing

Examples:
AWS EC2, Google Compute Engine, Azure VMs

PaaS
Platform as a Service

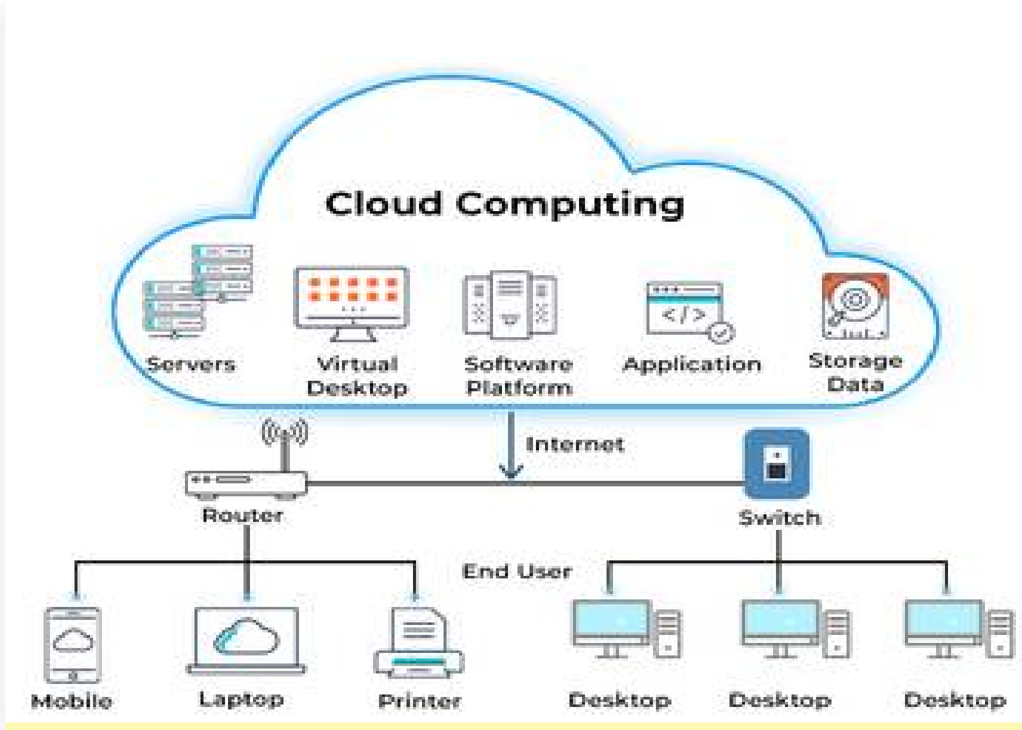
- ✓ Hardware & software tools
- ✓ Application development platform
- ✓ Managed infrastructure
- ✓ Faster deployment & scaling

Examples:
Heroku, AWS Elastic Beanstalk, Google App Engine

SaaS
Software as a Service

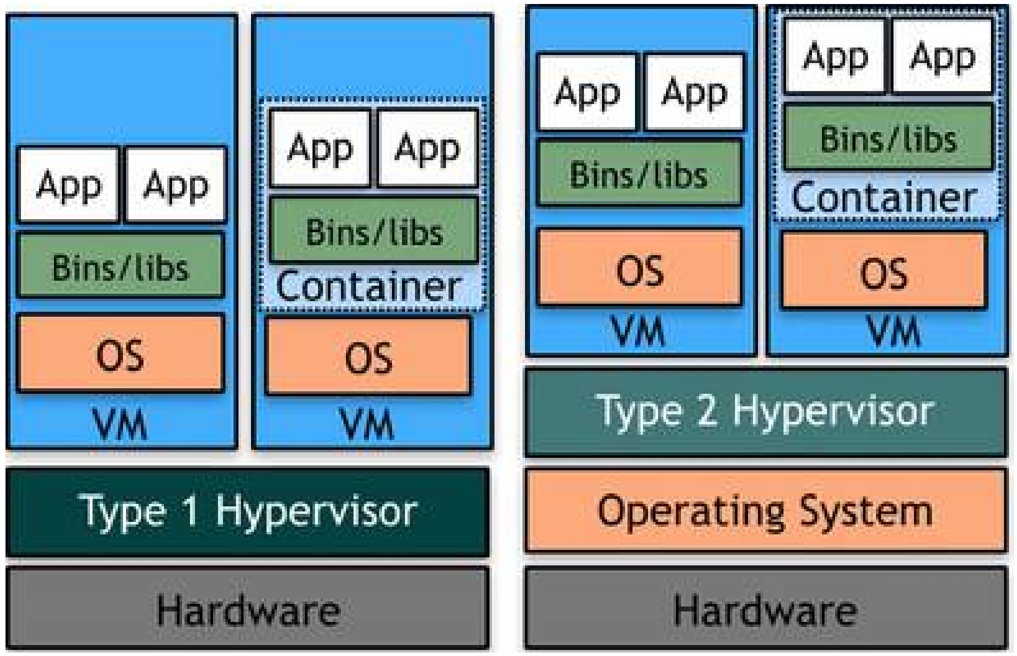
- ✓ Ready-to-use software
- ✓ Subscription-based access
- ✓ No installation or maintenance
- ✓ Accessible from anywhere

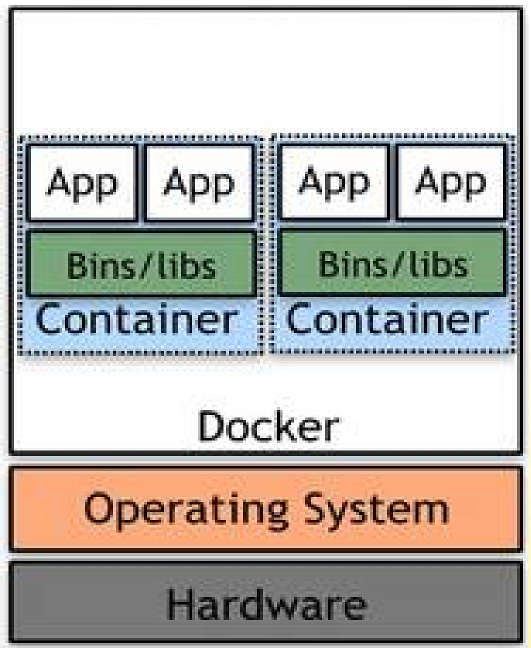
Examples:
Gmail, Microsoft 365, Salesforce, Dropbox



Cloud Working

- **Virtualization**
  - Physical servers are split into multiple Virtual Machines (Vms).
  - Each VM behaves like an independent computer.
  - Enables efficient hardware utilization & isolation.
- **Hypervisors**
  - Software layer that manages VMs on physical hardware.
  - Two types:
    - Type 1 (Bare-Metal) – runs directly on hardware (e.g., VMware ESXi, KVM).
    - Type 2 (Hosted) – runs on top of an OS.
- **Containers & Orchestration**
  - Containers (e.g., Docker) are lighter than VMs, sharing the same OS kernel.
  - Kubernetes (K8s) orchestrates thousands of containers:
  - Automates deployment, scaling, and networking.
  - Provides elasticity (scale up/down on demand).





Cloud Computing Approaches

- **Virtual Machine–Based Clusters:** Built on hypervisors (KVM, VMware ESXi, Hyper-V). Traditional enterprise cloud architecture.

  Each node runs multiple Vms. Managed by OpenStack, CloudStack, or VMware vSphere.
- **Container-Orchestrated Clusters:** Built around containers (Docker, Podman, CRI-O). Managed/orchestrated by: **Kubernetes (K8s) – most popular. OpenShift** (RedHat’s enterprise Kubernetes). **Nomad** (HashiCorp, lightweight alternative). Rancher (multi-cluster management). Favored for cloud-native applications.
- **Hybrid Cloud Clusters** (Combine on-premise HPC or private cloud with public cloud (AWS, Azure, GCP)). Managed using tools like Anthos (Google), Azure Arc, AWS Outposts. Provides flexibility + control.

Agenda

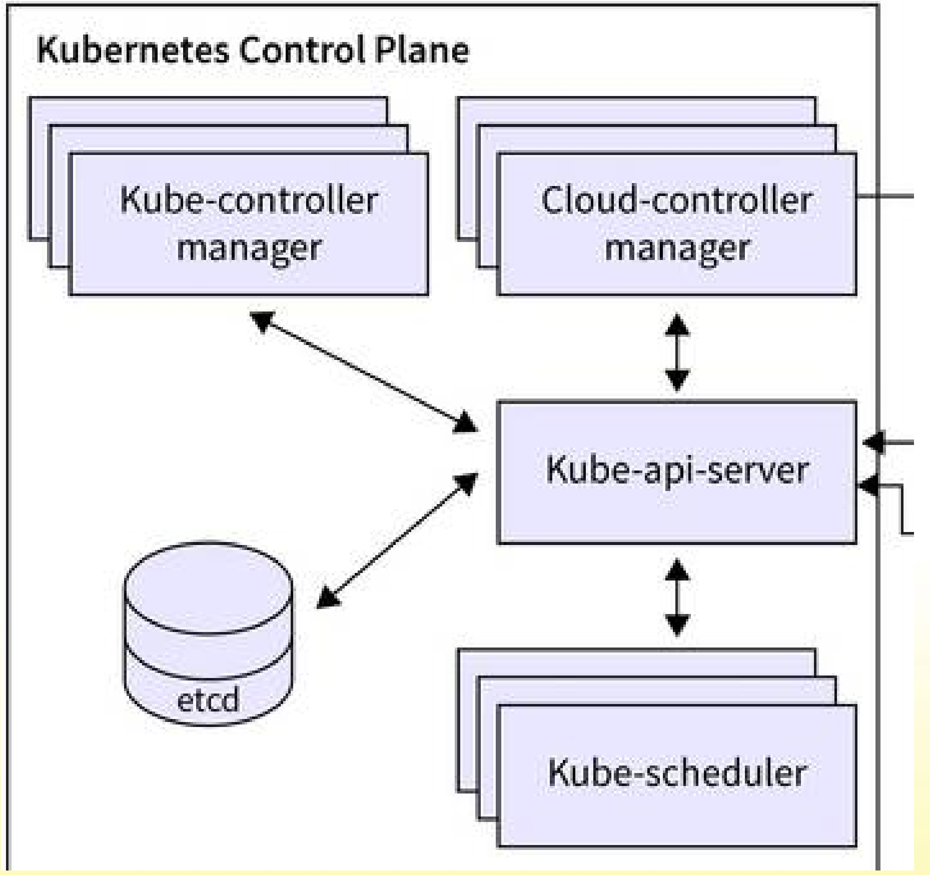
- Importance of Cloud Computing
- **Side-by-Side Cloud Software Stack Mapping (Same Hardware)**
- Kubernetes: Hardware and Software Management
- Cloud-Native Application Deployment (vs Bare Metal)
- Developing Distributed Applications on Kubernetes
- Comparison: Bare Metal vs Cloud: Gains and Overheads
- Hybrid Cluster Strategy: Combining HPC and Cloud-Native

Cloud Computing: Kuberneties Basics

- Application – code/model.
- Docker – tool to build the image.
- Image – read-only template of the app.
- Container – running instance of the image.
- Pod – one or more containers together (share IP/storage).
- Service – stable name/IP to reach Pods.
- Namespace – project/team boundary & quotas.
- Node – worker machine that runs Pods.
- Cluster – many nodes + control plane.
- Orchestrator (Kubernetes) – schedules/starts/stops/scales on the cluster.

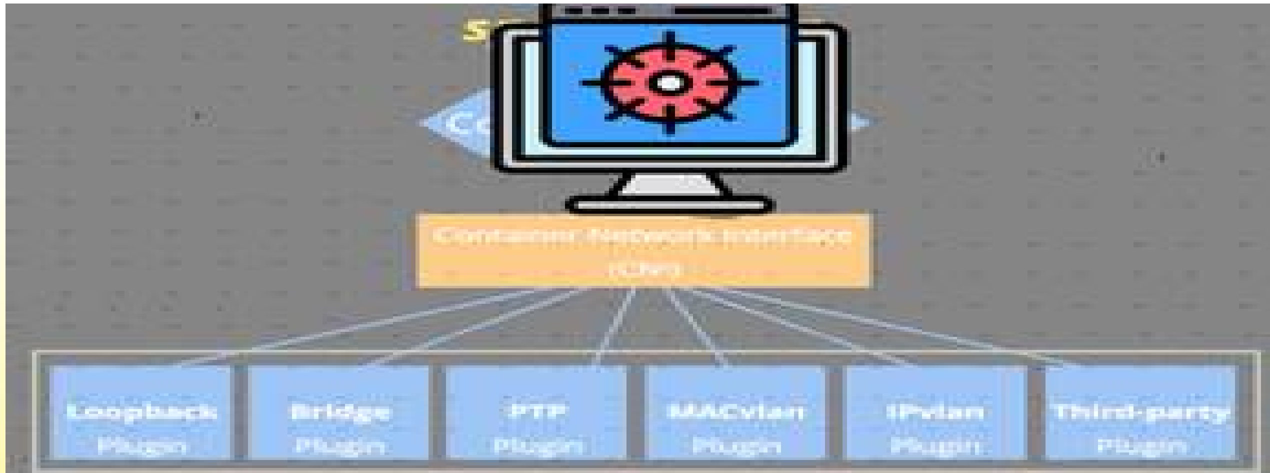
OS & Runtime Layer – HPC vs K8s

- HPC: Runs a standard Linux OS (e.g. RHEL, CentOS, or Debian) directly on each node.
- Cloud (K8s): Runs on Linux OS per node, but uses a container runtime on each node to run containerized apps. K8s nodes use tools like Node Feature Discovery to tag hardware features. The Control Plane (masters) schedules containers instead of a **batch scheduler.**



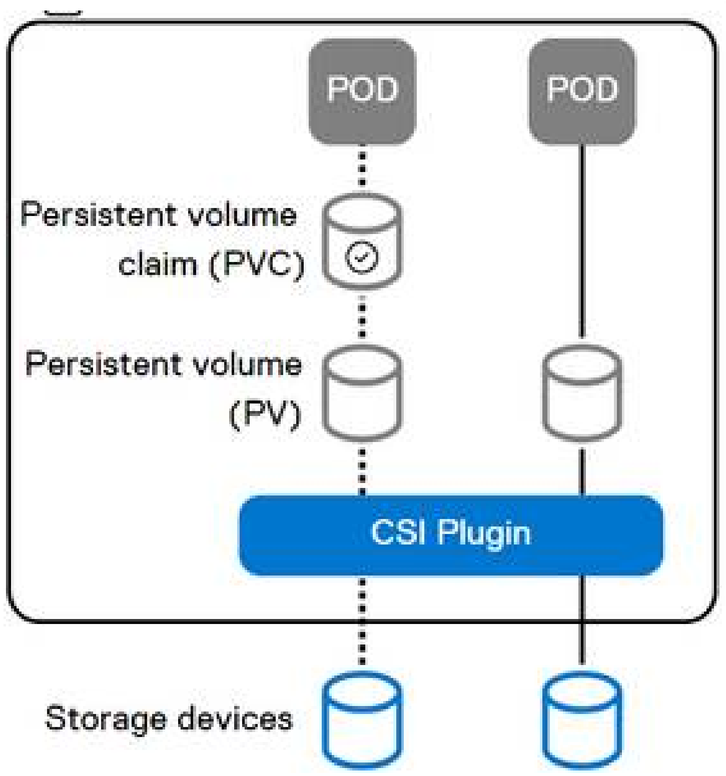
Network & Service Exposure – HPC vs K8s

- HPC: Uses high-performance networks (InfiniBand or RoCE) with low latency and high throughput for MPI communication. Flat host networking.
- Cloud (K8s): Uses a Container Network Interface (CNI) for pod networking give each Pod an IP and connect it to the cluster network.
  - Calico – fast routing + NetworkPolicy; common on-prem choice.
  - Cilium – eBPF-based, advanced observability/policy, great performance.
  - Weave Net – simple mesh overlay with encryption.
  - Multus – lets a Pod have multiple networks (e.g., add an RDMA/IB interface).



Storage & Data – HPC vs K8s

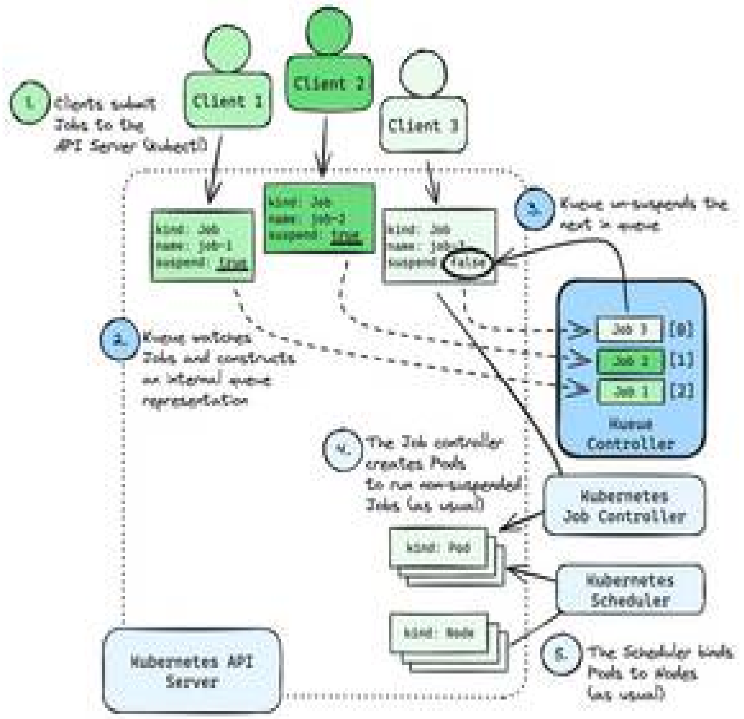
- HPC: Relies on shared POSIX file systems (NFS, Lustre, GPFS) mounted on all nodes for high-speed parallel I/O. Large data is often accessed via these network file systems.
- **Cloud (K8s):** Uses the CSI (Container Storage Interface) to provision volumes. For example, Rook-Ceph or Longhorn can provide distributed block or file storage to pods.
  - K8s can mount existing NFS as Persistent Volumes via a CSI driver.

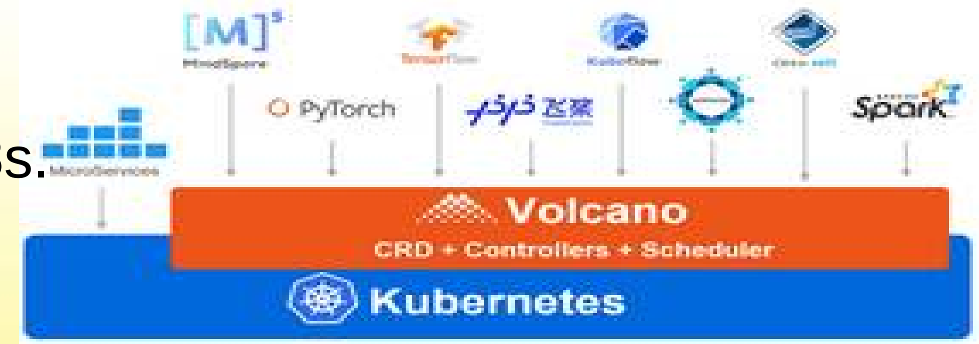


Big pool of storage that is fast, fault-tolerant, and self-healing.

Scheduling & Workflows – HPC vs K8s

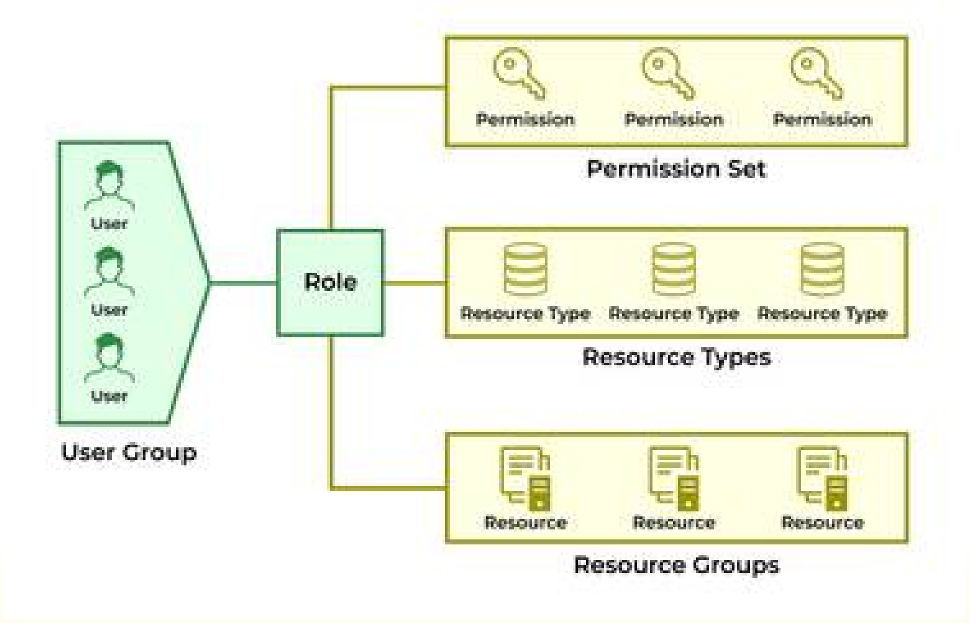
- HPC: A batch scheduler (Slurm, PBS, LSF) manages job queues. Users submit jobs with requested resources; the scheduler finds a slot where those nodes are free and launches the job across nodes (often using mpirun or srun for MPI).
- Cloud (K8s): Kubernetes has a built-in scheduler for podsfor batch jobs.
  - Kueue (K8s batch scheduling API) or Volcano add HPC-like batch queue capabilities.
  - K8s also has a native Job resource for run-to-completion tasks, and higher-level workflow managers.
  - There are K8s operators for MPI (MPIJob) and for distributed ML training, bridging HPC-style jobs into K8s.


Client 1
Client 2
Client 3
1. Clients submit Jobs to the API Server (kubectl)
kind: Job name: job-1 suspend: true
kind: Job name: job-2 suspend: true
kind: Job name: job-3 suspend: false
2. Kueue watches Jobs and constructs an internal queue representation
3. Kueue un-suspends the next in queue
Job 3 [0]
Job 2 [1]
Job 1 [2]
Kueue Controller
4. The Job controller creates Pods to run non-suspended Jobs (as usual)
Kubernetes Job Controller
kind: Pod
Kubernetes Scheduler
kind: Node
Kubernetes API Server
5. The Scheduler binds Pods to Nodes (as usual)



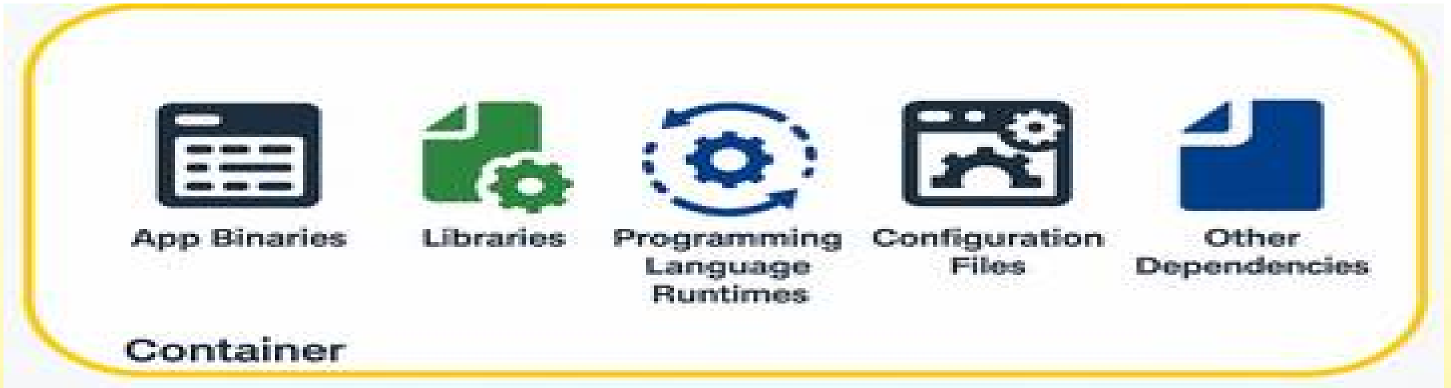
Security & Multi-Tenancy – HPC vs K8s

- HPC: Separation is typically by Unix users and groups. All jobs share the OS with equal privileges (user processes isolated by user ID).
- Cloud (K8s): Built for multi-tenancy:
  - uses Namespaces to isolate groups of resources by team or project,
  - RBAC (Role-Based Access Control) to define what each user/service account can do.
  - Pods run with security contexts, and cluster policies (Pod Security Standards or tools like Gatekeeper/Kyverno) enforce rules (e.g., restricting running privileged containers).
  - This allows multiple users/apps to share a cluster more safely, each in their sandbox.



Toolchains & Environments – HPC vs K8s

- HPC: Uses environment modules (e.g. Lmod) to load specific compiler versions, libraries (MPI, math libs, etc.) for each session or job. Software is often built from source or managed by package managers like Spack for optimized builds.
- Cloud (K8s): Uses Open Container Initiative (OCI) container images to encapsulate applications with all their dependencies. Instead of loading modules, users pull a container (from a registry) that already contains the needed runtime (e.g., a Python environment, specific library versions). Tools like Helm or Kustomize help deploy applications consistently. Private registries (e.g. Harbor) store images. Security scanning of images (SBOM, vulnerability scan) is done to ensure environment integrity.



Observability & Operations – HPC vs K8s

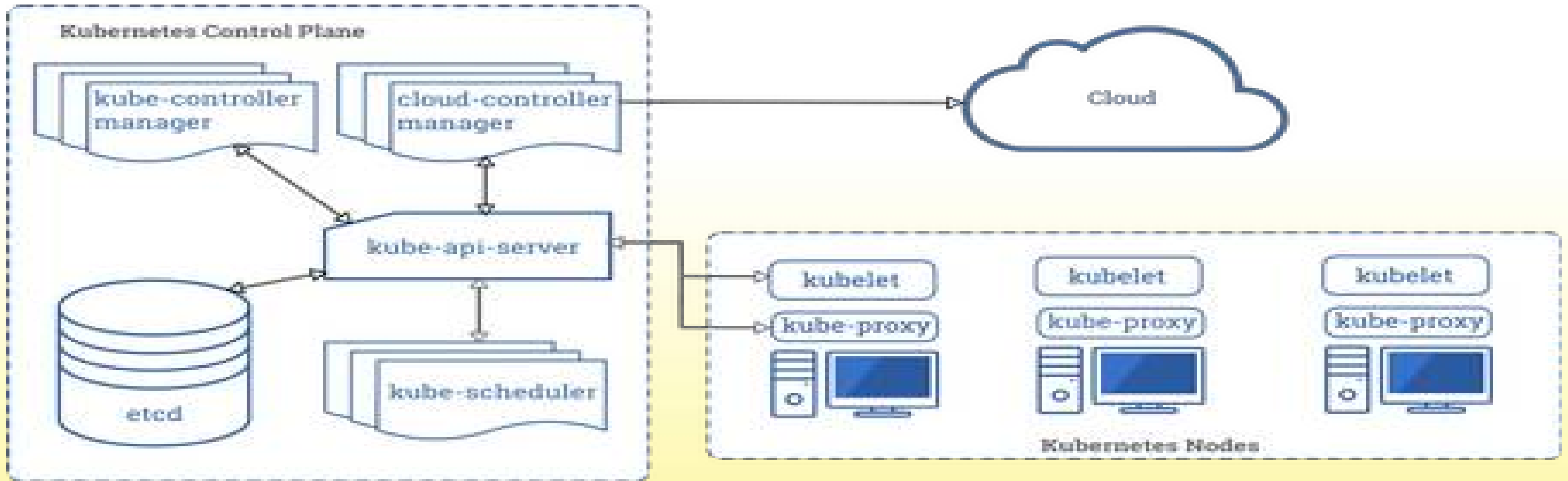
- HPC: Monitoring is often done with tools like Ganglia, Nagios or node-level exporters feeding into Grafana dashboards. Logs are usually per node or per job (written to files on a shared FS).
- Cloud (K8s): Comes with a rich observability stack.
  - Prometheus-based monitoring stack (Prometheus + Alertmanager + Grafana) is used to collect metrics from all pods and nodes.
  - Logs from applications can be aggregated through logging systems (e.g., EFK or Loki).
  - Traces can be collected with OpenTelemetry for distributed applications.
  - Give easier cluster-wide insights at a glance. K8s also provides events for each object (e.g., if a pod was evicted due to memory pressure, etc., it’s recorded), aiding in debugging.

Agenda

- Importance of Cloud Computing
- Side-by-Side Cloud Software Stack Mapping (Same Hardware)
- **Kubernetes: Hardware and Software Management**
- Cloud-Native Application Deployment (vs Bare Metal)
- Developing Distributed Applications on Kubernetes
- Comparison: Bare Metal vs Cloud: Gains and Overheads
- Hybrid Cluster Strategy: Combining HPC and Cloud-Native

Kubernetes Control Plane & Nodes

- Control Plane Components: The **brain of K8s running on master node(s).** Maintains desired state, e.g. ensures required number of pods.
- Node Components: Each worker **node runs a Kubelet,** a **Container Runtime** (like containerd to run the containers), and a **kube-proxy** (for networking, routing traffic to pods). **Nodes register** with the control plane and report their capacity (CPU, memory, etc.).
  - Admins: install/upgrade cluster, CNI (network), CSI (storage), GPU plugins; set labels/taints; set quotas & limits.
  - Developers: write YAML; declare CPU/RAM/GPU, storage, placement; choose Job/Deployment/StatefulSet.
  - At runtime: Scheduler picks a node → Kubelet starts pod, mounts storage, wires network, assigns devices.

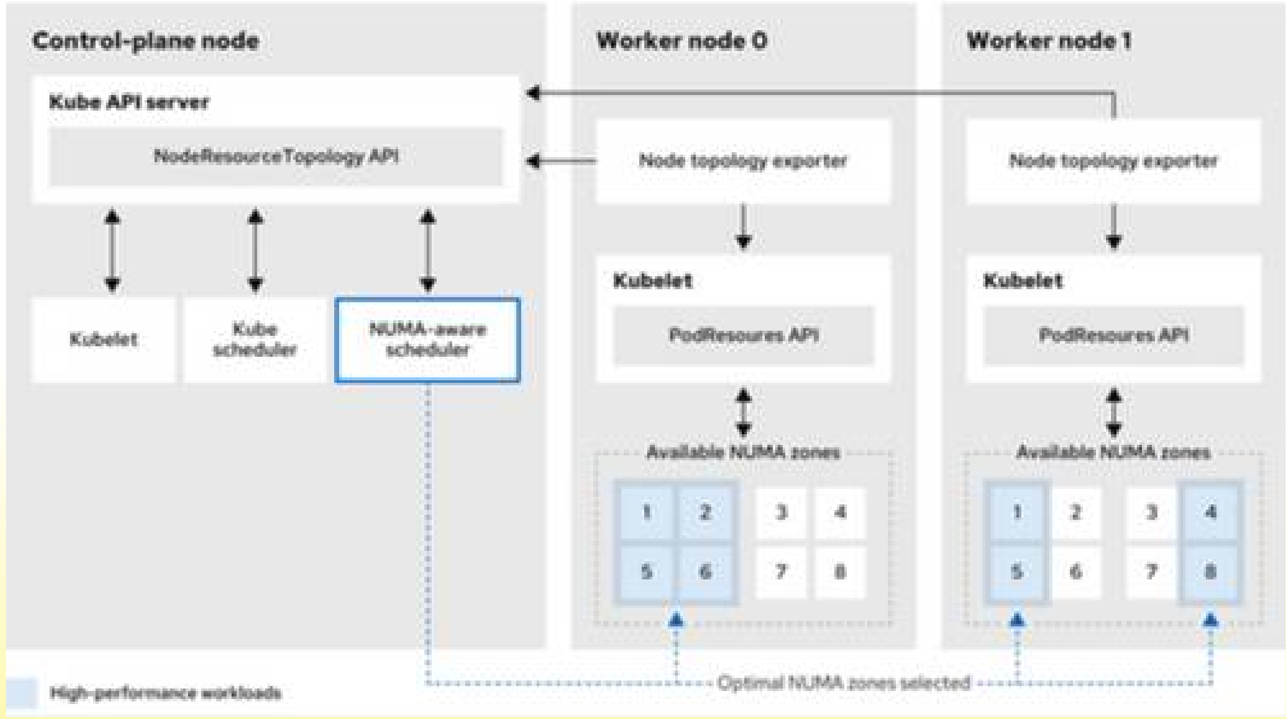


Compute & Memory Scheduling in K8s

- Kubernetes schedules at the container level with resource awareness. Each pod can specify resource requests (guaranteed minimum) and limits (max usage) for CPU/memory.
- Pods are assigned Quality-of-Service classes (Guaranteed, Burstable, Best-Effort) based on whether they have requests/limits – this influences eviction priority when resources are scarce.
- Affinity/Anti-affinity: To run **muliple pods together** or apart (e.g. keep replicas on different nodes for HA using anti-affinity).
- Node Pools: Nodes might be in groups (e.g. a pool of GPU nodes, a pool of high-memory nodes). K8s label nodes and target specific node.
- Policy hooks (per namespace)
  - ResourceQuota: total CPU/RAM/GPU caps for a team.
  - LimitRange: default requests/limits per container.
  - PriorityClass: who wins if cluster is full (preemption).

Topology & CPU/Memory Tuning

- NUMA Awareness: By default, K8s treats a node as a single unit, but with the Topology Manager feature, K8s can coordinate CPU and device assignment for optimal NUMA locality.
- HugePages: HPC apps that use large memory pages hugepages in K8s – nodes can be configured to allocate some memory as hugepages, and pods can request them, enabling high-memory-bandwidth workloads to perform well.
- CPU Pinning: Allocate dedicated CPU cores to certain pods (static policy).
- Node Feature Discovery (NFD): A K8s add-on that labels nodes with hardware details (CPU model, GPU present, NIC features). This helps scheduling decisions .
- Turn on performance knobs
  - Topology Manager: keep CPU, memory, and GPU on same NUMA side.
  - CPU Manager (static): give dedicated cores to latency-sensitive pods.
  - Node Feature Discovery labels: schedule only to nodes that have needed features.



Accelerators & High-Speed I/O in K8s

In HPC, admins install drivers on each node manually; in K8s, an Operator can manage that as part of the cluster.

- Device Plugins: Kubernetes supports GPUs, FPGAs, high-performance NICs, etc. via device plugins.
- NVIDIA GPU Operator: Simplifies GPU management – it can install GPU drivers, GPU monitoring, and other necessary components on each node automatically.
- High-Speed Networking: Using Multus (a multi-network plugin), pods can attach to multiple networks.

RDMA & DPDK: There are plugins to enable RDMA (Remote Direct Memory Access) for pods and to allow user-space packet processing using Data Plane Development Kit

- Who allocates accelerators
  - Device plugin advertises GPUs/FPGA to K8s.
  - Scheduler finds a node with free device(s).
  - Kubelet Device Manager assigns real device (e.g., /dev/nvidia0) into the pod.

Storage & Data Paths in K8s

- Persistent Volumes (PVs): K8s uses PVs to represent storage that pods can use.
- Ephemeral vs Persistent: Some storage is ephemeral (lives only with the pod, e.g. emptyDir scratch space on a node’s disk or memory). Persistent storage (via PVCs – Persistent Volume Claims) outlives pods, so data remains even if the application restarts or moves.
- Object Storage: K8s applications often use object storage for large data (instead of only POSIX FS).
- Data Locality: K8s doesn’t automatically move data to where the compute is. Instead, it relies on fast networks and distributed storage.
- Data practices
  - One default StorageClass (fast SSD) + one “capacity” class.
  - Keep a shared data lake (POSIX or CephFS) + object store for models/artifacts.
  - PVCs (volumes) for jobs; object store for large, versioned data.

Resource Allocation Flow (Who & How)

- Nodes report capacity (CPU/RAM/devices); NFD adds labels.
- User submit pod/job with requests/limits, PVCs, placement.
- Admission checks quotas/limits; mutates defaults if missing.
- Scheduler filters/scores nodes (resources, taints, affinity, volume).
- Kubelet pulls image, mounts PVs (CSI), attaches NICs (CNI), assigns GPU, enforces limits (cgroups).
- While running: HPA/VPA/autoscaler/preemption may act; QoS decides eviction.

Agenda

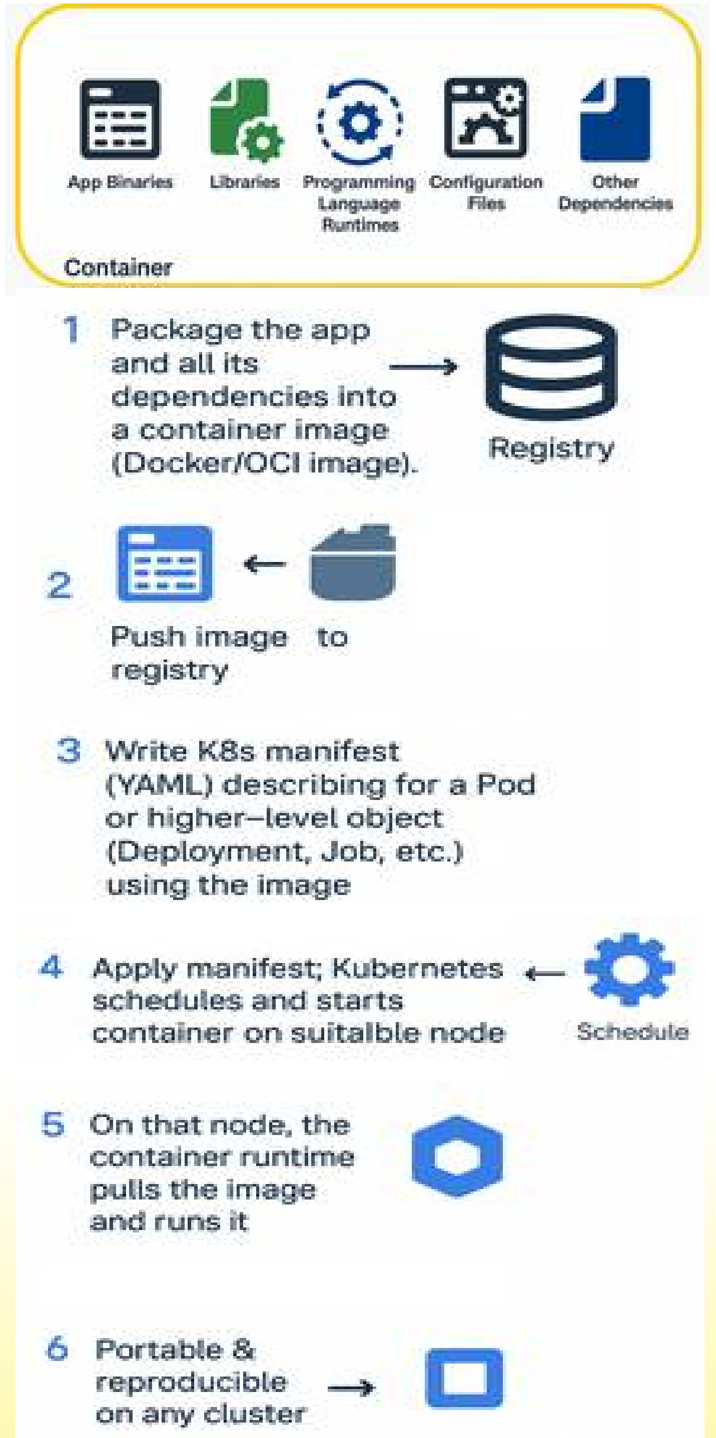
- Importance of Cloud Computing
- Side-by-Side Cloud Software Stack Mapping (Same Hardware)
- Kubernetes: Hardware and Software Management
- **Cloud-Native Application Deployment (vs Bare Metal)**
- Developing Distributed Applications on Kubernetes
- Comparison: Bare Metal vs Cloud: Gains and Overheads
- Hybrid Cluster Strategy: Combining HPC and Cloud-Native

Containerization Deployment

- Traditional HPC Deployment: Compile or install software on each node (or a shared filesystem). Ensure all library dependencies are satisfied on the OS using environment modules.
- K8s, deploy a pod and the system places it immediately if resources are available (or waits if using a batch queue controller). K8s also handles restarts or rescheduling of failed pods automatically, which is a different model from HPC’s one-and-done jobs.

Kubernetes Deployment

- Package the application and all its dependencies into a container image (Docker/OCI image).
- Push this image to a registry.
- Write a K8s manifest (YAML) describing a Pod or higher-level object (Deployment, Job, etc.) that uses this image.
- When applied, K8s schedule and start the container on a suitable node.
- The container runtime on that node pulls the image and runs it.
- This encapsulation means the app's environment is portable and reproducible across any cluster.


App Binaries
Libraries
Programming Language Runtimes
Configuration Files
Other Dependencies
Container
1 Package the app and all its dependencies into a container image (Docker/OCI image).
Registry
2 Push image to registry
3 Write K8s manifest (YAML) describing for a Pod or higher–level object (Deployment, Job, etc.) using the image
4 Apply manifest; Kubernetes schedules and starts container on suitalble node
Schedule
5 On that node, the container runtime pulls the image and runs it
6 Portable & reproducible on any cluster

Portability: Build Once, Run Anywhere

- With containers, the application build into an image and can run it on any server/cluster that supports that container runtime.
- Separation of Config and Code: Kubernetes encourages keeping configuration and data outside of the image.
- Consistent Environment: Whether on on-premises or cloud, as long as the node has a compatible OS and container runtime, run the same.
- Versioning: Container images are versioned enabling easy rollbacks or parallel testing of different versions of an application.

Multi-Architecture Images & Drivers

- Multi-Arch Containers: Support images that support multiple CPU architectures (e.g., x86_64, ARM64) using manifest lists.
- GPU Drivers in Containers: GPU-accelerated applications need NVIDIA drivers. In K8s, the node’s GPU driver is installed. Containers typically use NVIDIA’s CUDA base images which expect the driver on host.
- Compatibility Considerations: When containerizing HPC apps, make sure the container’s OS and drivers are compatible with the host (e.g., if using Infiniband, the container might need certain libraries installed).

Placement Controls in Kubernetes

- Node Selection: In K8s, specify the node by labels on nodes and nodeSelector or node affinity in the pod spec.
- Affinity/Anti-Affinity: Certain pods run on the same node (affinity) or on different nodes (anti-affinity). This is useful for spreading workloads or for grouping
- Resource Requests/Limits: By setting precise requests for CPU, memory, and extended resources (like GPUs), where the scheduler places the pod. The scheduler won’t over-commit a node beyond its capacity of requests.
- Topology Spread: K8s enforce spreading pods across failure domains (like different racks or zones) so that a single hardware failure doesn’t take out all replicas.

Data Attachment Patterns

- Persistent Volume Claims (PVCs): In K8s, to use storage, usually create a PVC which binds to a PV. The PVC can then be mounted into any pod that needs that data.
- Storage Classes: Different performance storage can be offered (SSD vs HDD, local vs network).
- Read-Only Data Sets: K8s can mount config data or large data sets as read-only volumes (e.g., using ConfigMap for small configs, or a read-only many PVC for shared reference data).
- Object Store Access: Applications might also directly read from object storage (using S3 APIs) instead of using file mounts.

Agenda

- Importance of Cloud Computing
- Side-by-Side Cloud Software Stack Mapping (Same Hardware)
- Kubernetes: Hardware and Software Management
- Cloud-Native Application Deployment (vs Bare Metal)
- **Developing Distributed Applications on Kubernetes**
- Comparison: Bare Metal vs Cloud: Gains and Overheads
- Hybrid Cluster Strategy: Combining HPC and Cloud-Native

Parallel & Distributed Workloads – HPC Style on K8s

- Array Jobs (Indexed Jobs): For embarrassingly parallel tasks (many independent runs), K8s Jobs can be indexed. Submit one Job with, say, 100 completions, and it will run 100 pods with index 0..99.
- MPI Jobs: For tightly-coupled parallel tasks (like MPI programs that need all pods to start together), Kubernetes can use an MPI Operator (a controller that sets up an MPI job).
  - It creates the launcher and worker pods, handles the mpirun orchestration, and ensures pods can find each other (often through a headless Service for DNS or environment variables).
  - The MPI Operator essentially brings HPC-style MPI execution to K8s. Run the MPI pods and communicate (some implement “gang scheduling” at a job level using K8s pod readiness gates).
  - Not as mature as Slurm for MPI.

Parallel & Distributed Workloads – AI/Big Data on K8s

- Kubeflow / ML Operators: Kubeflow provides CRDs (custom resources) for common ML patterns like TFJob (for TensorFlow) or PyTorchJob. These operators manage the complexity of launching multiple pods for training, setting roles (master/worker/ps), and handling failure. They’re built on Kubernetes but abstract away a lot of YAML.
- Ray, Spark, Dask on K8s: Big data and AI workflows often use these frameworks. All have K8s integration.
- Scaling & Fault Tolerance: These frameworks utilizes K8s features for resiliency – e.g., if a Spark executor pod dies, K8s can restart it, and Spark can recover the task.

Workflows & Batch Job Management

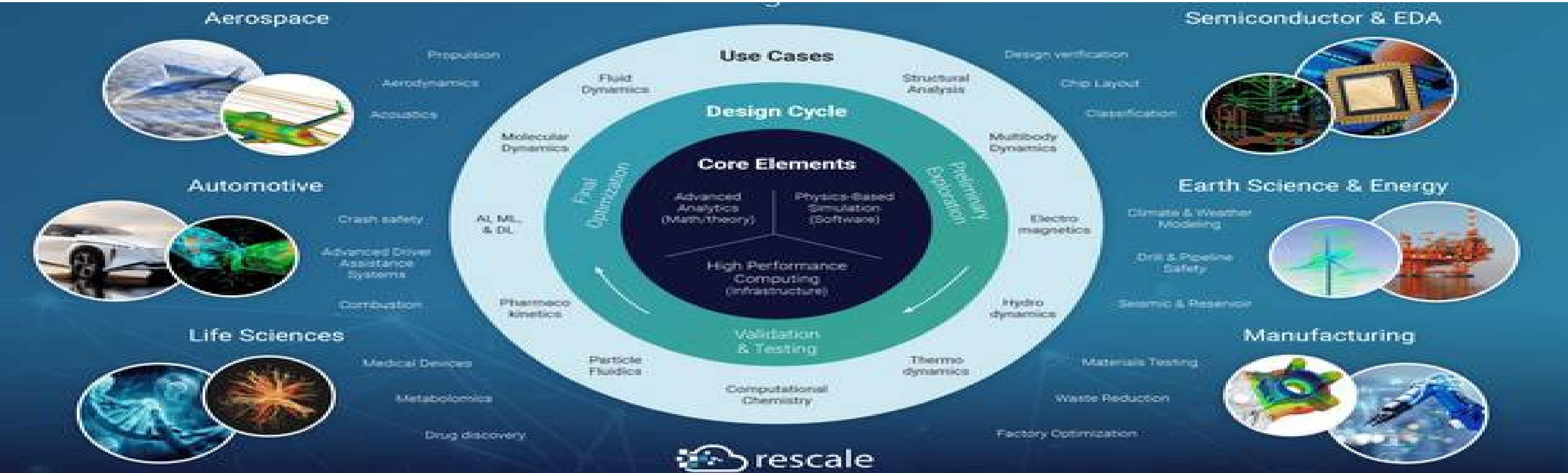
- **Argo Workflows / Kubeflow Pipelines:** These are higher-level workflow engines on top of K8s. They allow you to define multi-step workflows where each step is a container.
- **Batch Scheduling Extensions:** While K8s default scheduling handles most, for batch jobs requiring queuing or fairness, Kueue (a Kubernetes project) can integrate with the scheduler to delay starting jobs until resources are available (like an HPC scheduler would). Volcano is another project adding features like job priority, quotas, gang scheduling (co-scheduling pods together) on top of K8s. These are evolving to make K8s more HPC-like in scheduling policies.
- Quotas and Fairness: Namespaces in K8s can have ResourceQuota objects that ensure one team doesn’t use all resources.

Agenda

- Importance of Cloud Computing
- Side-by-Side Cloud Software Stack Mapping (Same Hardware)
- Kubernetes: Hardware and Software Management
- Cloud-Native Application Deployment (vs Bare Metal)
- Developing Distributed Applications on Kubernetes
- **Comparison: Bare Metal vs Cloud: Gains and Overheads**
- Hybrid Cluster Strategy: Combining HPC and Cloud-Native

HPC: Optimized for

- **Low-Latency Networking**
- **Bare-Metal Performance & Determinism**
- **Batch Job Scheduling**



Cloud-Native (Kubernetes): Optimized for

Declarative Operations: Uses GitOps and declarative manifests including all infrastructure configurations (like Kubernetes manifests, Terraform files, Helm charts).

- Applications run in reproducible containers, providing consistent environments across deployments.
- Elastic Scaling & Self-Service.
- MLOps & Microservices: Supports modern AI/ML pipelines (CI/CD, automated retraining) and microservice architectures.

Scalability
Flexibility
Cost Savings
Benefits of Cloud Computing
Security
Reliability
Improved Collaboration
Rapid Deployment

Cloud-Native Importance

- Reproducibility
- Easy Updates (CI/CD)
- Multi-Tenancy & Collaboration
- GitOps (Declarative Management)
- Advanced Observability
- Unified Platform
- No need for separate specialized clusters

Cloud – Overheads and Challenges

- Performance Overheads:
  - CPU and memory overhead for the container runtime and kernel namespaces,
  - Network overlays (like Flannel or Calico) can introduce extra latency
- Startup Latency: Launching a container pod can be slower than running a process on bare metal.
  - Image download time and container init time.
- Resource Fragmentation:
  - If not carefully configured, there might have many small pods that leave resources fragmented
- Operational Complexity:
  - Need new skills (DevOps, YAML, container debugging). The cluster requires maintenance (upgrading K8s versions, monitoring etc.).
- Storage and Data Management:
  - POSIX filesystem on K8s requires CSI drivers or NFS provisioners which might not expose all features or performance of the raw filesystem.

Cloud not Ideal for

- Ultra Low-Latency, Synchronized Jobs: If you have a job where dozens of nodes must step in perfect sync (e.g., tightly-coupled MPI with frequent small messages), any added latency is problematic.
- While K8s can do MPI, it might struggle with scheduling all pods at once or add tiny delays that HPC wouldn’t.
- Extremely Short Tasks at Huge Scale
- Special Hardware/OS Tuning: If you need direct control over hardware (e.g., specialized network routing, kernel tweaks, real-time OS settings),
- Simpler Workloads Without Need for Cloud Features

Workload Taxonomy – Examples

- Tightly Coupled HPC Jobs: e.g. Computational Fluid Dynamics (CFD), Finite Element Analysis (FEA), large MPI-based simulations.
- Throughput AI Training: e.g. distributed deep learning on multiple GPUs. Characteristics: heavy compute like HPC, but may benefit from cloud’s flexibility.
- AI Inference & Services: e.g. serving ML models via APIs, data preprocessing pipelines, feature stores for AI. Characteristics: loTs of smaller tasks, possibly user-facing services – tends to favor Kubernetes (scalable, long-running services).
- Analytics & ETL: e.g. Spark, Ray, Dask jobs for data analytics. Characteristics: can run on HPC or K8s. Increasingly run on K8s using frameworks (Spark on K8s, Ray on K8s) for easier resource pooling – a middle ground between HPC and cloud.

Performance Comparison – Bare Metal vs Kubernetes

- Example: Initial results found performance is very close. At 8 GPUs, bare metal was about ~8% faster
- The slight slowdown in K8s is attributed to network overhead

Agenda

- Importance of Cloud Computing
- Side-by-Side Cloud Software Stack Mapping (Same Hardware)
- Kubernetes: Hardware and Software Management
- Cloud-Native Application Deployment (vs Bare Metal)
- Developing Distributed Applications on Kubernetes
- Comparison: Bare Metal vs Cloud: Gains and Overheads
- **Hybrid Cluster Strategy: Combining HPC and Cloud-Native**

Why a Hybrid Model?

- A hybrid cluster lets you use HPC methods for what they do best (e.g., tightly-coupled simulations) and cloud-native methods for what they excel at (e.g., deploying persistent services, scaling AI experiments). You don’t have to pick one or the other exclusively – you allocate each workload to the environment that suits it, on the same physical infrastructure.
- High Utilization:
- Smooth Transition:
- Unified Infrastructure,

Hybrid Deployment Models

- Side-by-Side Clusters: One approach is to keep HPC and K8s logically separate but physically adjacent.
- Co-Located Schedulers (Same Nodes): A more integrated model is running K8s and an HPC scheduler on the same nodes.
- Bursting Between HPC and Cloud: Another “hybrid” angle is keeping HPC on-prem and using cloud K8s for overflow or particular workloads. E.g., if an HPC job needs more nodes than on-prem has, or an AI workflow needs cloud-specific services (like BigQuery), you burst that portion to cloud. This requires federating or bridging the environments (maybe with a workload manager that submits jobs to cloud or a VPN linking on-prem K8s with cloud)

Mapping an Existing HPC Cluster to Cloud-Native Stack

- Containerize Key Applications
- Set Up Kubernetes Environment:
- Integrate Authentication & Access
- Data Sharing:
- Pilot and Validate:
- Training and Workflow Migration
- Iterate & Expand

Challenges in Managing a Hybrid Cluster

- Resource Arbitration
- Complexity & Skill Set
- Software Environment Consistency
- Performance Tuning
- Cultural Adoption

Summer School on Supercomputing & AI for Tech Entrepreneurs
29-31 August 2025 Namal University Mianwali

HPC vs Cloud-Native AI Application Deployment: Approaches, Strengths, and Hybrid Strategies

Prof. Dr. Tassadaq Hussain
Specialization: Supercomputing and Artificial Intelligence
Director Centre for AI and BigData
Affiliated Member: Barcelona Supercomputing Centre