HPS

Darius Heere

Containers in HPC: From Image Formats to Fast Delivery

Supervisor: Anila Ghazanfar

Why should you care about containers in HPC?

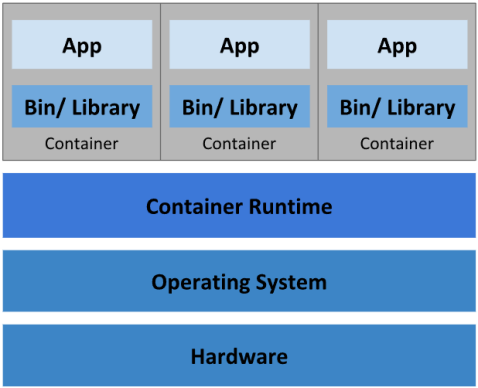
Because you can scale your **workflow** from a **laptop** to **1000 nodes**

... but the bottleneck is often the container image.

Outline

Container Basics

- App + dependencies, packaged as **one unit**
- Isolated user space + shared host kernel → fast and lightweight (VMs isolate more)
- OCI (Open Container Initiative) image = standard portable container format
- Layered images built from a `Dockerfile`



Container Deployment [1]

Why use Containers in the HPC Context?

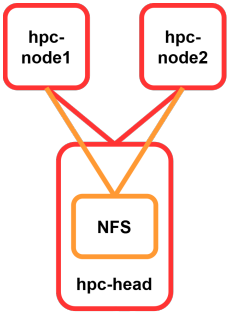
- Less reliance on cluster-provided software modules [2]
- Same environment across different systems and over time [2]

$\Rightarrow$ Portability and Reproducibility

- Install and bundle all required system packages and libraries inside container [2]
- Easy experimentation with different toolchains and dependency versions [2]

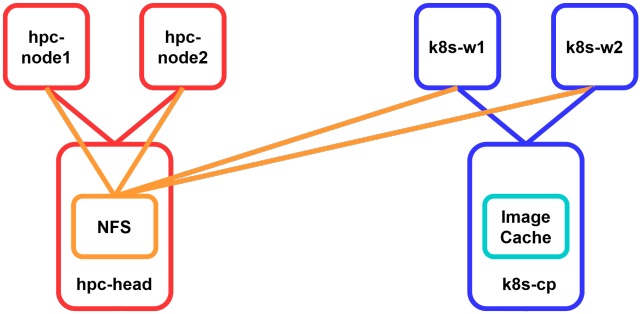
$\Rightarrow$ Flexibility

HPC and Kubernetes Cluster



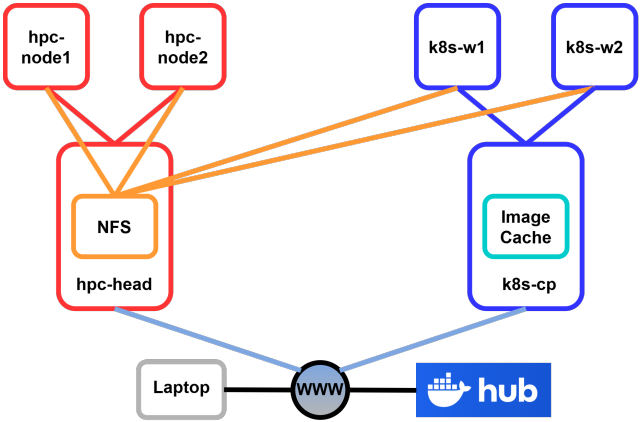
- HPC Cluster
  - Similar to the Stack in HPCSA-module [3]
  - Warewulf
  - Slurm
  - Apptainer
  - NFS Service (separate network)

HPC and Kubernetes Cluster



- HPC Cluster
  - Similar to the Stack in HPCSA-module [3]
  - Warewulf
  - Slurm
  - Apptainer
  - NFS Service (separate network)
- Kubernetes (K8s) Cluster
  - Based on K3s: Lightweight Kubernetes [4]
  - Pull-Through Cache
  - Stargz Snapshotter

HPC and Kubernetes Cluster



- HPC Cluster
  - Similar to the Stack in HPCSA-module [3]
  - Warewulf
  - Slurm
  - Apptainer
  - NFS Service (separate network)
- Kubernetes (K8s) Cluster
  - Based on K3s: Lightweight Kubernetes [4]
  - Pull-Through Cache
  - Stargz Snapshotter

Outline

Image Handling: Pull and Execution Process K8s

- **Prefetch phase:** Deploy a DaemonSet that forces both worker nodes to pull the OCI image from Docker Hub
  - OCI images run directly on Kubernetes (containerd) → no conversion step required
- **Execution phase:** Submit a Kubernetes Job that runs from the locally cached image

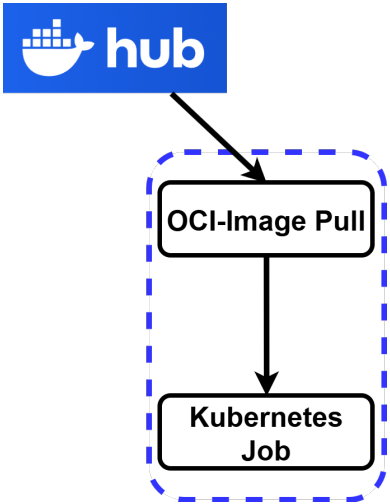


Image Handling: Pull and Execution Process HPC

- **Pull phase**: Apptainer downloads the OCI layers into the local cache and builds a single SIF (Singularity Image Format) file (default in the Apptainer workflow) and stores it on NFS

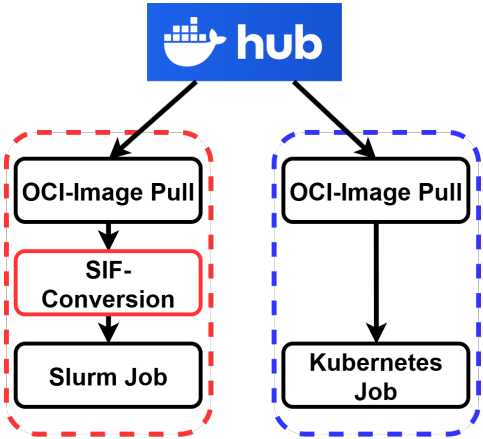
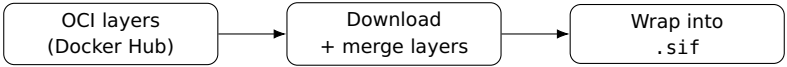


Image Handling: SIF-Conversion Details

- Apptainer can't run OCI images directly → SIF conversion required [5]

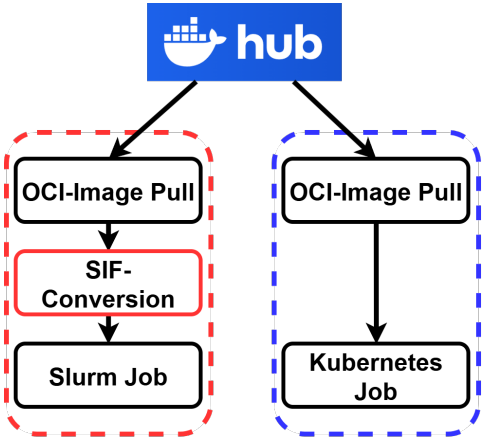


- Why convert to SIF?
  - Single, read-only image file that works well on shared storage (e.g., NFS)
  - No privileged daemon required on shared HPC nodes (plus signing/verification support)

- **The SIF file is an extra artefact to manage**

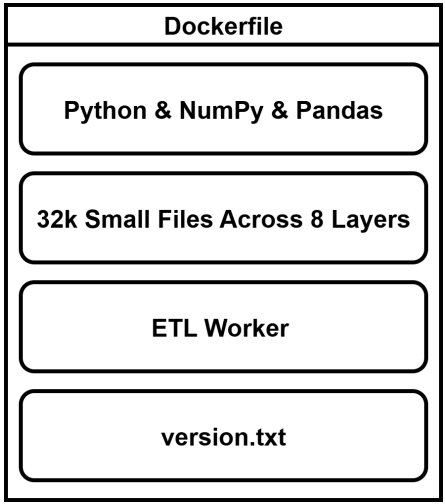
Image Handling: Pull and Execution Process HPC

- **Pull phase:** Apptainer downloads the OCI layers into the local cache and builds a single SIF file (default in the Apptainer workflow) and stores it on NFS
- **Execution phase:** Submit one Slurm batch job that launches the conatinerized workload; Apptainer runs the workload from the shared SIF image



Patch Tuesday Benchmark: Idea and Image Setup

- Proxy for an ETL workload that is redeployed with tiny updates (hotfixes, config tweaks, dependency bumps)
- The ETL worker reads data shards from NFS, aggregates them, computes hashes and writes the results back to NFS
- **Only the** `version.txt` **changes between the runs (v1-v5)**

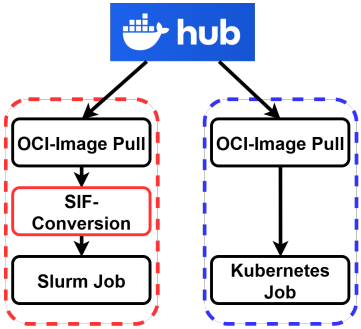


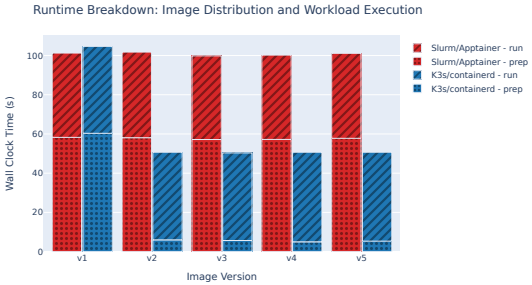
Patch Tuesday: Benchmark Results

Setup HPC: Cold Apptainer Cache (OCI and SIF), Cold Worker Nodes, Idle **Setup K3s**: Cold Pull-Through Cache, Cold Worker Nodes, Idle

SD across runs ($n = 3$; range across versions): Dist [s]: K3s 0.58–1.73, HPC/Slurm 0.38–0.94 Run [s]: K3s 0.58–2.89, HPC/Slurm 0.58–2.00

Patch Tuesday: Benchmark Summary

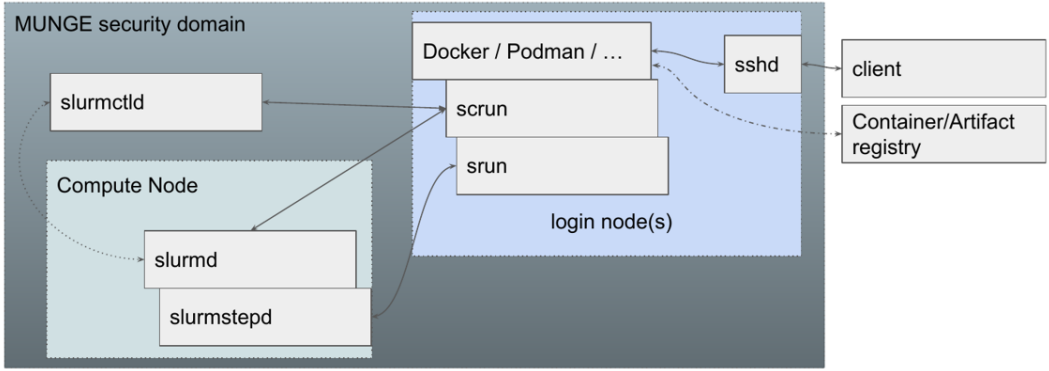




!

- Runtime performance is similar
- OCI → SIF conversion costs extra time
- The SIF file is an extra artefact to manage

Recent Developments: Native OCI Support in Slurm



Using container runtime frontend through scrun [6]

- **Docker/Podman** = user-facing CLI + image management [7]
- **scrun** = OCI runtime frontend that launches containers via Slurm [7]

Outline

Image Delivery: Regular Pull and Execution Process

- Build & push **OCI image** to Docker Hub
- Job is submitted on the control-plane node (k8s-cp)
- Scheduler places the Pod on a worker node
- Kubelet/containerd **pulls + unpacks full image** (via pull-through cache)
- Pod runs to completion

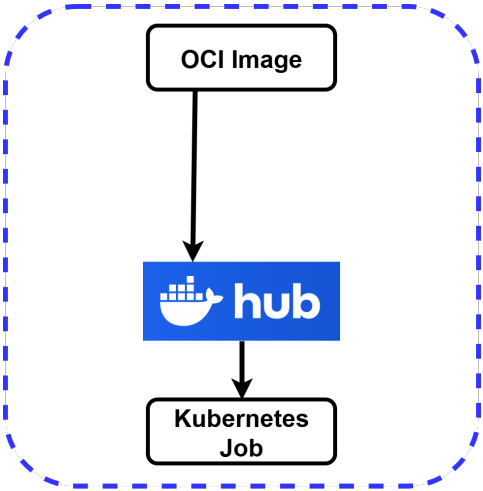


Image Delivery: Pull and Execution Process using (e)Stargz

- Convert **OCI** $\rightarrow$ **eStargz** and push to Docker Hub

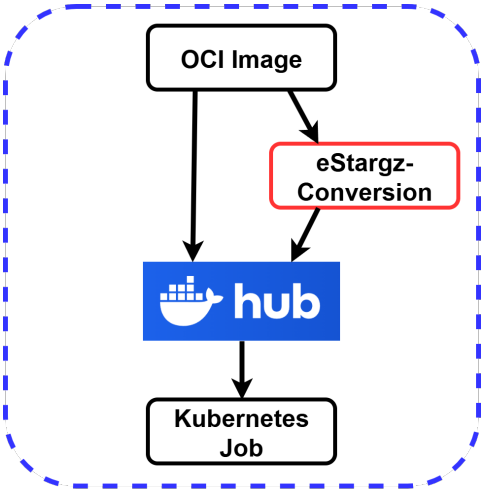
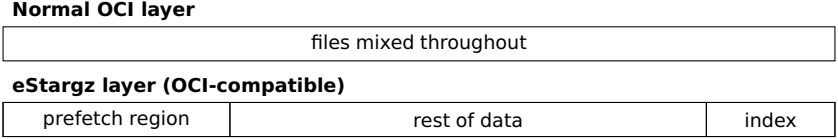


Image Delivery: (e)Stargz Details

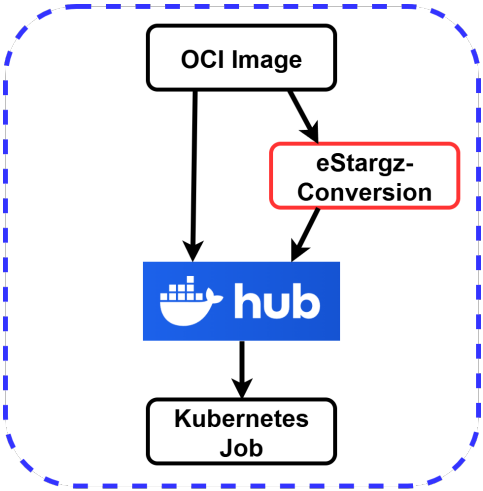
- eStargz is a special OCI image format, and the stargz snapshotter uses it to enable lazy pulling from standard registries [8]



- **HTTP Range requests:** Fetch index and files in prefetch region first, then rest of the data chunkwise if needed
- **Why it helps:** Start executing *before* the full image is downloaded $\rightarrow$ lower time-to-first-result for large images
- **Trade-off:** Extra image conversion step + snapshotter overhead

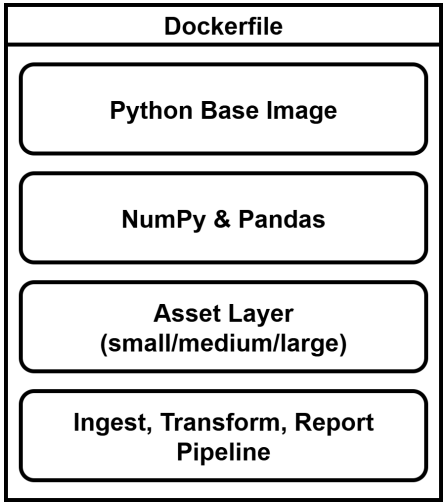
Image Delivery: Pull and Execution Process using (e)Stargz

- Convert **OCI** $\rightarrow$ **eStargz** and push to Docker Hub
- Job is submitted on k8s-cp; Pod scheduled to a worker
- containerd uses the **stargz snapshotter** for **lazy pulling** on that worker node
- eStargz is also cached in the pull-through cache

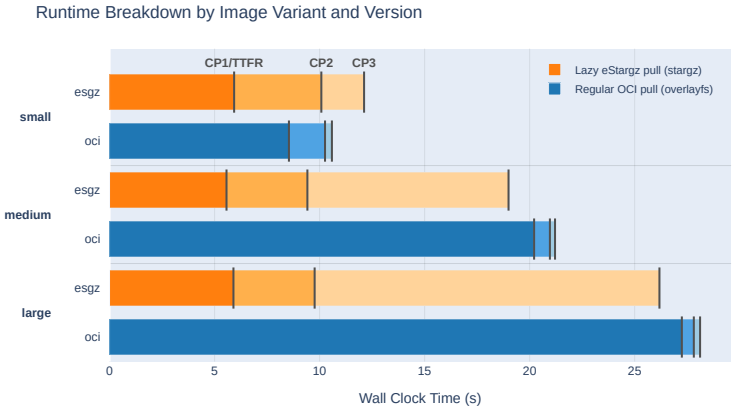


Lazy Pulling vs Full Pull Benchmark: Idea and Image Setup

- Proxy for a **3-stage ETL analytical workload**, each stage writes a checkpoint file to NFS
- CP1 after ingestion: Pure Python only (no heavy dependencies)
- CP2 after transform: Imports NumPy and Pandas to clean and compute KPIs
- CP3 after report: Reads a large reference asset
- **Compare OCI vs eStargz Checkpoint times**



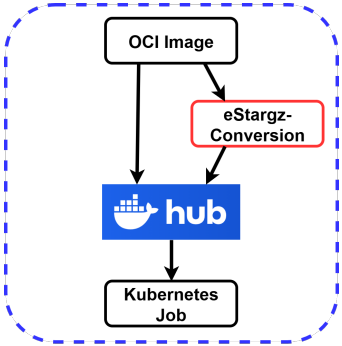
Lazy Pulling vs Full Pull: Benchmark Results



Setup K3s: Warm Pull-Through Cache, Cold Worker Nodes, Cluster in an idle State

SD (n=3, averaged over the 3 checkpoints): eStargz: small: 0.82, medium: 0.29, large: 0.31 OCI: small: 6.48, medium: 0.92, large: 0.25

Lazy Pulling vs Full Pull: Benchmark Summary

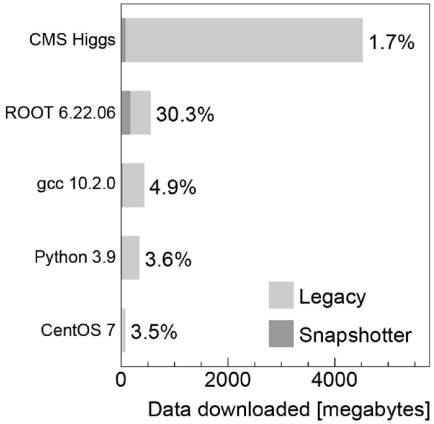


!

- More fine grained checkpoints needed to isolate variance source
- Lazy pulling can drastically cut the time to first result
- This benefit grows with image size

Recent Developments: File-level lazy pulling with CVMFS

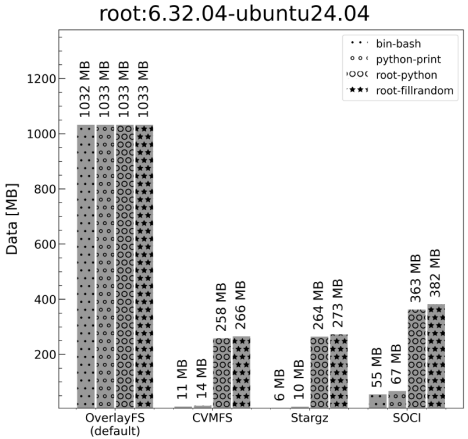
- CVMFS (CernVM File System) is a distributed, read-only filesystem
- CVMFS snapshotter: fetch **files on demand** + cache via Squid/CVMFS[9]
- Pulling dominates container startup, but most bytes are never used early [9]
- **HPC takeaway:** makes many-parallel-job launches feasible without saturating the network



Data downloaded (legacy vs CVMFS snapshotter) [9]

Recent Developments: Snapshotter comparison

- **Evaluation / Context:**
  - Impressive results: Download only 0.6 % to 37 % of the data
  - But: Workloads touch only a (very) small subset of files → Best case scenario for snapshotters
- **Workloads:**
  - **bin-bash**: start a minimal interactive shell
  - **python-print**: launch Python and print “Hello World”



Transferred data across snapshotters [10]

Outline

Summary

What to remember

- **Kubernetes** runs OCI images natively; **Apptainer** adds an OCI→SIF conversion step (extra artefact)
- **Lazy pulling** can reduce time-to-first-result, but introduces snapshotter overhead
- For short HPC jobs, image **pulling + unpacking + format conversion** can dominate runtime
- **Outlook:** Native OCI in Slurm + snapshotters are key steps toward faster container workflows

Outline

References I

[1] The Kubernetes Authors. *Overview - Kubernetes Documentation*. Kubernetes. URL: https://kubernetes.io/docs/concepts/overview/ (visited on 01/14/2026).

[2] Richard S Canon and Andrew Younge. "A case for portability and reproducibility of HPC containers". In: *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. IEEE. 2019, pp. 49–54.

[3] Julian Kunkel and Jonathan Decker. *Practical: High-Performance Computing System Administration*. URL: https://hps.vi4io.org/teaching/autumn_term_2025/hpcsa (visited on 01/14/2026).

[4] K3s Project Authors. *K3s: Lightweight Kubernetes*. URL: https://k3s.io/ (visited on 01/14/2026).

[5] Apptainer Project. *Apptainer Quick Start*. URL: https://apptainer.org/docs/user/main/quick_start.html (visited on 01/17/2026).

[6] Nathan Rini. *Containers in Slurm)*. Accessed: 2026-01-17. 2024. URL: https://slurm.schedmd.com/SC24/Containers.pdf (visited on 01/17/2026).

[7] SchedMD LLC. *Containers Guide - Slurm Workload Manager*. URL: https://slurm.schedmd.com/containers.html (visited on 01/14/2026).

References II

[8] containerd contributors. *eStargz (stargz-snapshotter documentation)*. Accessed: 2026-01-17. 2026. URL: https://github.com/containerd/stargz-snapshotter/blob/main/docs/estargz.md (visited on 01/17/2026).

[9] Simone Mosciatti, Clemens Lange, and Jakob Blomer. "Increasing the Execution Speed of Containerized Analysis Workflows Using an Image Snapshotter in Combination With CVMFS". In: *Frontiers in Big Data* Volume 4 - 2021 (2021). ISSN: 2624-909X. DOI: 10.3389/fdata.2021.673163. URL: https://www.frontiersin.org/journals/big-data/articles/10.3389/fdata.2021.673163.

[10] Max Fatouros et al. "Efficient and fast container execution using image snapshotters". In: *EPJ Web of Conferences*. Vol. 337. EDP Sciences. 2025, p. 01197.