GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Containers in HPC: From Image Formats to Fast Delivery

Darius Heere

Supervisor: Anila Ghazanfar

Georg-August-Universität Göttingen
Institute of Computer Science

March 30, 2026

Abstract

Containers have become an important part of modern high-performance computing (HPC) because they support portable, reproducible, and flexible software deployment while preserving much of the performance expected in scientific computing environments. As the use of containers in HPC grows, attention shifts from the application itself to the workflows through which container images are created, distributed, and accessed on compute systems.

This is especially relevant for short-running and iterative analytical workloads, where image-handling overhead can form a noticeable part of the overall turnaround time. In such settings, performance is shaped by the image workflow and startup mechanism as much as by the execution itself. Different approaches exist: Some HPC-oriented solutions rely on converting OCI images into formats such as SIF before execution, while others use native OCI handling and increasingly support techniques such as lazy pulling to reduce startup overhead.

This work investigates two current trends in containerized HPC workflows: The OCI→SIF workflow in comparison to native OCI-based execution, and snapshotter-based lazy pulling as a mechanism for improving startup efficiency. The study focuses on the trade-off between compatibility with established HPC practices, operational simplicity, and the cost of image preparation, transfer, and access at runtime.

To evaluate these aspects, we conduct an experimental comparison using representative containerized workloads, with a focus on startup behavior and end-to-end turnaround time. The results highlight the importance of image-handling decisions for practical HPC usage and show under which workload characteristics and execution scenarios the different approaches are advantageous. Overall, the report provides a systems-oriented perspective on how container workflow design influences the efficiency and usability of containerized workloads in HPC.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☑ During brainstorming
- ☐ When creating the outline
- ☐ To write individual passages, altogether to the extent of 0% of the entire text
- ☑ For the development of software source texts
- ☑ For optimizing or restructuring software source texts
- ☑ For proofreading or optimizing
- ☐ Further, namely: -

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Figures

List of Abbreviations

HPC High-Performance Computing

OCI Open Containers Initiative

NFS Network File System

k8s Kubernetes

ETL Extract, Transform, Load

SIF Singularity Image Format

JSON JavaScript Object Notation

OS Operating System

NAT Network Address Translation

SSH Secure Shell

I/O Input/Output

VNFS Virtual Node File System

CVMFS CernVM File System

HPDA High-Performance Data Analytics

VM Virtual Machine

PXE Preboot Execution Environment

CRI Container Runtime Interface

1 Introduction

1.1 Motivation

Containers have become increasingly relevant in HPC because they offer low overhead, portability, reproducibility, and flexible software deployment [KB22]. Unlike traditional cluster-provided software environments, containers allow users to bundle application code with the required libraries and dependencies, making it easier to reproduce experiments and run the same workload across systems and over time [CY19]. As a result, containers are becoming an integral part of modern HPC and High-Performance Data Analytics (HPDA) workflows [KB22]. At the same time, their use also shifts attention to container image-handling workflows, because performance and usability depend on how container images are prepared, transferred, and accessed at runtime [ZZH23]. These aspects are especially relevant for short-running, iterative analytical workloads, where image handling can account for a substantial share of total turnaround time. This motivates the investigation of two recent trends in containers for HPC: Native Open Containers Initiative (OCI) image handling versus Apptainer's conversion to the Singularity Image Format (SIF), and snapshotter-based lazy pulling to improve startup efficiency.

1.2 Research Questions and Scope of Objectives

This report is guided by two research questions. They define the scope by focusing on image-handling and startup-related effects of containers in short-running analytical HPC workloads.

- RQ1: What overhead does the OCI→SIF conversion workflow introduce compared to native OCI handling in containerized HPDA environments?
- RQ2: Under which conditions can lazy pulling reduce the time-to-first-result for OCI-containerized analytical workloads?

1.3 Contributions and Report Structure

This report contributes an empirical comparison of two current trends in container handling for HPC: The OCI→SIF workflow versus native OCI handling, and snapshotter-based lazy pulling for OCI images. The results show that the OCI→SIF workflow introduces noticeable overhead in short-running and frequent-update scenarios, mainly because image conversion and redistribution delay execution. The lazy-pulling experiments further show that startup times can be reduced substantially when workloads access only a limited subset of image contents early. Overall, the findings indicate that container image handling can strongly influence turnaround time in short analytical HPC workloads.

After this introduction, the report provides the technical background for the later analysis in Section 2. The experimental platforms and fairness considerations of the two testbeds are then introduced in Section 3. Based on this foundation, the OCI→SIF workflow is examined in Section 4, followed by the evaluation of lazy pulling with snapshotters in Chapter 5. Afterwards, the findings of both experimental parts are brought together in the cross-topic synthesis in Chapter 6, before the report closes with the conclusion in Chapter 7.

2 Background

2.1 OCI Images and Layer Semantics

The OCI image format is the most important standard for portability in modern container ecosystems [Fat+25]. The Open Container Initiative specifies this standard[1]. The format makes container workloads portable across tools like Docker[2]/Podman[3] for building images, image registries like Docker Hub[4] and execution runtimes like containerd[5]. But also HPC-related tooling like Apptainer[6].

An OCI image is a set of blobs referenced by a digest, that is, a content identifier derived by hashing the blob content. This includes an image manifest in JavaScript Object Notation (JSON) format, which references a configuration object and an ordered list of layer blobs, that are tar archives that are typically compressed with gzip[7]. The layers contain the actual filesystem content. Each referenced object is identified by a cryptographic digest, which allows for integrity checking and deduplication.

OCI images are layer-based, because layers allow for efficient reuse and incremental updates. This means that only those layers need to be fetched, which are not yet present on the node, and that if two images share identical layers, a node needs to store and download those blobs only once.

A typical container image is built from a Dockerfile, in which each instruction like `RUN`, `COPY` and `ADD` produces a new layer. As the specification describes OCI images as an *"ordered collection of root filesystem changes"*[8], the Dockerfile layers are stacked in order. This means the first layer is the base, and each subsequent layer modifies what came before. The final container filesystem, that the application sees at runtime, is the result of applying all layers from bottom to top.

From a systems perspective, OCI layers are the fundamental unit of transfer and reuse in registries and runtimes. The merged root filesystem view is only materialized locally by the runtime.

2.2 Apptainer and SIF

Apptainer[9] is a container runtime that is common in the HPC context, because it matches how HPC systems are operated. Apptainer's native image format is SIF, and in the context of this work, constructed OCI images need to be converted to SIF if they are to be run using Apptainer.

[1] *The OpenContainers Image Spec.* URL: `https://specs.opencontainers.org/image-spec/?v=v1.1.1` (visited on Feb. 28, 2026).

[2] *Docker.* URL: `https://www.docker.com/` (visited on Feb. 28, 2026).

[3] *Podman.* URL: `https://podman.io/` (visited on Feb. 28, 2026).

[4] *Docker Hub.* URL: `https://hub.docker.com/` (visited on Feb. 28, 2026).

[5] *containerd.* URL: `https://containerd.io/` (visited on Feb. 28, 2026).

[6] *Apptainer.* URL: `https://apptainer.org/` (visited on Feb. 28, 2026).

[7] *The OpenContainers Layer Spec.* URL: `https://specs.opencontainers.org/image-spec/layer/` (visited on Feb. 28, 2026).

[8] *The OpenContainers Image Spec.* URL: `https://specs.opencontainers.org/image-spec/?v=v1.1.1` (visited on Feb. 28, 2026).

[9] *Apptainer.* URL: `https://apptainer.org/` (visited on Feb. 28, 2026).

The conversion to SIF can be described as a series of steps[10]. First, Apptainer downloads the referenced compressed layer blobs and unpacks them into a local directory (so that the Operating System (OS) has access to all files). Then, Apptainer applies the layers in order to construct the final filesystem view (also called a "flattened" filesystem). The flattened filesystem is then compressed into a SquashFS image, which is read-only and optimized for fast reads. Finally, the SquashFS filesystem is packaged into a SIF container file. At this point, Apptainer can also include metadata and signatures, where the latter are part of Apptainer's integrity and trust features.

This conversion has a strong operational motivation in the HPC context: A single, read-only image file is much easier to handle on shared storage, for example on Network File System (NFS), than a native OCI image[11], which consists of multiple OCI blobs and metadata, as described in Subsection 2.1. In practice, Apptainer can cache the individual OCI layers, but any change to the OCI image requires a new SIF conversion and a new SIF artifact.

2.3 Snapshotters and Lazy Pulling

A snapshotter is the containerd[12] component, that manages the representations of image layers on disk. Based on the OCI image, it provides the mounts that will become the container's rootfs. This happens in the form of two kinds of mounted snapshots. The committed snapshot is a stored, read-only filesystem state, which the snapshotter can use for multiple containers. There is usually a chain of read-only snapshots for the image layers. The active snapshot is a per-container, writable space. It sits on top of the committed snapshots when the container runs, which means that a container's mountable rootfs is the combination of both[13].

The default snapshotter that containerd uses for Linux is called `overlayfs`[14]. For this snapshotter, the regular flow starts by pulling the image descriptor (manifest/index), the config, and the layer blobs from the registry into the containerd content store. This content is still in the same compressed format as in the registry. Then, containerd unpacks the image by reading the layers and applying them in order to a filesystem, creating committed snapshots on disk. Then, the snapshotter creates an active snapshot as a per-container writable layer on top. A container can only be started once this mountable root filesystem (rootfs) is available, with the active snapshot serving as the container's writable layer[15].

The stargz snapshotter is a non-core containerd snapshotter plugin that implements the concept of remote snapshots and lazy pulling. In this workflow, containerd still resolves the image manifest, but then asks the stargz snapshotter, which runs as a separate proxy-plugin daemon (`containerd-stargz-grpc`) to prepare the rootfs layers. However, unlike

[10] *Support for Docker and OCI Containers*. URL: `https://apptainer.org/docs/user/main/docker_and_oci.html` (visited on Feb. 28, 2026).

[11] *Security in Apptainer: Singularity Image Format (SIF)*. URL: `https://apptainer.org/docs/admin/1.0/security.html` (visited on Feb. 28, 2026).

[12] *containerd*. URL: `https://containerd.io/` (visited on Feb. 28, 2026).

[13] *containerd snapshots package documentation*. URL: `https://pkg.go.dev/github.com/containerd/containerd/snapshots` (visited on Feb. 28, 2026).

[14] *containerd Snapshotters*. URL: `https://github.com/containerd/containerd/blob/main/docs/snapshotters/README.md` (visited on Feb. 28, 2026).

[15] *containerd Documentation: Content Flow*. URL: `https://github.com/containerd/containerd/blob/main/docs/content-flow.md` (visited on Feb. 28, 2026).

`overlayfs`, these layers are not prepared from fully unpacked local files. Instead, they are mounted and accessed directly from the remote registry. This is possible because of the *eStargz* layer format. eStargz is still OCI-compatible, because it is just an alternative way of organizing OCI layer blobs so that they can be accessed efficiently via random reads.

The key idea is that an eStargz layer contains an index that lists files and their byte offsets within the compressed blob. This index is stored at the end of the layer, so the snapshotter can fetch a small tail portion of the blob first, read the index, and then locate individual files without downloading or decompressing the entire layer.

To reduce startup latency, eStargz additionally supports a "prefetch" region near the beginning of the layer. The stargz snapshotter can fetch this region first, start the container early, and then fetch remaining files on demand. During execution, when the container accesses a file that is not yet available locally, the stargz snapshotter retrieves only the required byte ranges from the registry (typically using HTTP range requests). It then decompresses only the needed chunks and serves them through the mounted filesystem view[16].

To enable this workflow, the OCI image must be converted to eStargz before it is pushed to a registry such as Docker Hub.

[16] *containerd Documentation: Content Flow.* URL: `https://raw.githubusercontent.com/containerd/containerd/main/docs/content-flow.md` (visited on Feb. 28, 2026).

3 Experimental Platforms

3.1 Testbed Overview and Fairness Choices

This work aims to evaluate container workflows in the form of benchmarks on two small, controlled clusters. One cluster represents an HPC-like environment and the other a cloud-like environment. Both clusters are deployed on the same OpenStack environment[17], where the six Virtual Machine (VM) in total are split into two independent 3-node clusters, as shown in Figure 1. As the goal is to keep the platforms as comparable as possible, the following setup choices were made. All six nodes use the same `m1.large` flavor with 4 vCPUs and 8 GB RAM.

Rocky Linux 9[18] is used as the base OS across the testbed. The management/control-plane nodes and the Kubernetes nodes are provisioned as standard OpenStack instances from a Rocky Linux 9 image. The HPC compute nodes are diskless/stateless and network-boot into a Rocky Linux 9 userspace/root filesystem image at each boot, without persisting OS state locally.

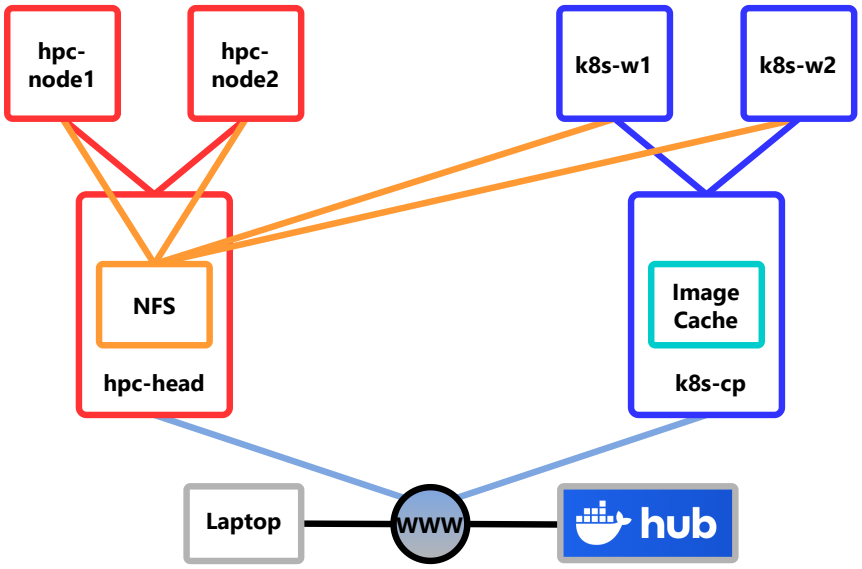


Figure 1: HPC and Kubernetes cluster stack overview.

There are three networks in play to separate concerns. The `hpc-head` and the three Kubernetes nodes (`k8s-cp`, `k8s-w1` and `k8s-w2`) are inside the same routed network with Network Address Translation (NAT). However, due to the configured OpenStack guidelines of our project, only those nodes with a FloatingIP attached actually have internet access, namely `hpc-head` and `k8s-cp`. These two nodes are also the entry points for access and configuration via Secure Shell (SSH). During operation, `k8s-w1` and `k8s-w2` do not have internet access, they only get a FloatingIP attached for configuration purposes.

There is also a non-routed L2 network for iPXE boot and provisioning of the diskless HPC compute nodes. Only the nodes of the HPC-like cluster participate in this network (`hpc-head`, `hpc-node1`, `hpc-node2`).

[17] *OpenStack: Open Source Cloud Computing Infrastructure (Project Overview)*. URL: `https://www.openstack.org/` (visited on Mar. 1, 2026).

[18] *Rocky Linux Documentation: Rocky Linux 9*. URL: `https://docs.rockylinux.org/9/` (visited on Mar. 1, 2026).

The third network is a non-routed L2 storage network, which is used for NFS traffic. The NFS server runs on `hpc-head`, and the worker nodes of both clusters access the same `nfs` export over this network so that benchmark Input/Output (I/O) follows an identical storage path and network topology. Furthermore, the benchmark I/O is isolated from the Preboot Execution Environment (PXE)/provisioning traffic and routed/NAT traffic, which reduces interference during measurements. The `k8s-cp` node technically also uses the NFS network, but only to write benchmark metadata to a shared place after a run.

Both clusters are also configured in a way, that benchmark workloads can run only on the respective compute/worker nodes. This prevents jobs from being scheduled onto head/control-plane nodes that handle cluster management or other services. This ensures that measurements reflect performance on dedicated, otherwise idle worker nodes.

3.2 HPC-like Cluster

The cluster that represents an HPC-like environment is mostly configured based on the instructions from the High-Performance Computing System Administration Course[19]. This cluster models a "classic" HPC setup with a central head (management) node, and diskless compute nodes. There is a batch scheduler and workload manager in the form of Slurm[20] installed, as well as an HPC container runtime in the form of Apptainer[21].

The `hpc-head` node is the single management/control node that provides all cluster-wide services, including the Warewulf[22] provisioning stack with DHCP, TFTP, and iPXE. The boot artifacts/root image that is used for the compute nodes' diskless boot over the PXE network is also configured via Warewulf. The head node also provides an NFS service used by both clusters. On the HPC side, the compute nodes mount `/nfs` at boot.

The Slurm controller (`slurmctld`) accepts job submissions, tracks node state, and decides where and when jobs run. For accounting purposes, namely the maintenance of a consistent record of job metadata and timestamps, `slurmdbd` relies on a MariaDB backend[23]. Authentication within the cluster happens via Munge[24]. The same Munge key is distributed across the head and compute nodes, so that Slurm components can trust each other.

Both worker nodes (`hpc-node1`, `hpc-node2`) do not behave like normal long-lived VMs. Instead, they boot into a centrally defined image and can be reprovisioned into a known state via iPXE/PXE on the dedicated network. They load a Warewulf-provided Virtual Node File System (VNFS) derived from a Rocky Linux 9 userspace/container filesystem. Node-specific configurations are applied via Warewulf overlays, not by manually configuring each node.

[19] *Practical: High-Performance Computing System Administration*. URL: `https://hps.vi4io.org/teaching/autumn_term_2025/hpcsa` (visited on Mar. 1, 2026).

[20] *Slurm Workload Manager: Overview*. URL: `https://slurm.schedmd.com/overview.html` (visited on Mar. 1, 2026).

[21] *Apptainer*. URL: `https://apptainer.org/` (visited on Feb. 28, 2026).

[22] *Warewulf: Stateless and Diskless Cluster Provisioning*. URL: `https://warewulf.org/` (visited on Mar. 1, 2026).

[23] *Rocky Linux Documentation: MariaDB database server*. URL: `https://docs.rockylinux.org/9/guides/database/database_mariadb-server/` (visited on Mar. 1, 2026).

[24] *MUNGE (MUNGE Uid 'N' Gid Emporium): Overview*. URL: `https://dun.github.io/munge/` (visited on Mar. 1, 2026).

Apptainer is installed on `hpc-head` and is also included in the VNFS image, so it is available on the compute nodes at runtime. As described in Section 2.2, the container images originate as OCI images (from Docker Hub) and are converted to SIF, which is stored on the shared NFS export. Benchmark workloads are submitted and scheduled by Slurm and executed on the compute nodes using `srun` job steps, where Apptainer is invoked to execute the SIF image as part of the Slurm job step.

3.3 Cloud-like Cluster

This cloud-like environment is implemented as a lightweight Kubernetes[25] cluster using K3s[26]. In contrast to the HPC-like platform, containerized workloads run as Kubernetes Pods and execute standard OCI images directly via the container runtime (containerd).

The `k8s-cp` node hosts the Kubernetes control-plane components (API server and cluster state management) and is configured such that it does not run user workloads. The worker nodes `k8s-w1` and `k8s-w2` run the K3s agent and are responsible for pulling images and executing benchmark workloads under Kubernetes scheduling.

Networking follows the same three-network design described in the testbed overview (Section 3.1). Kubernetes control-plane and node-to-node traffic use the routed network, while benchmark I/O is placed on the dedicated storage network. All Kubernetes nodes mount the shared NFS export (from `hpc-head`) on the host OS, so benchmark data reads and writes follow the same storage path and network topology as on the HPC-like cluster.

Because the worker nodes do not have reliable direct outbound internet access, a local Docker Registry (v2) is deployed on `k8s-cp` as an OS-level service (because `k8s-cp` is configured with the agent disabled) and configured as a pull-through (proxy) cache for Docker Hub. The worker nodes are configured to fetch container images through this internal registry. This allows Docker Hub pulls to be transparently routed via the cache endpoint while preserving the usual Docker Hub image references in Kubernetes workload manifests (for example, `docker.io/namespace/image:tag`).

The workers are additionally configured to use the stargz snapshotter[27] to enable lazy pulling as described in Section 2.3. Concretely, the stargz snapshotter is already included in the containerd runtime that ships with K3s, and only needs to be enabled on each worker node via configuration.

For the lazy pulling via a snapshotter to work in this setup, the pull-through cache needs to support HTTP Range requests against image layer blobs, because the stargz snapshotter retrieves only the required byte ranges from the registry (see Section 2.3). In the environment, this was validated directly from a worker by issuing HTTP `GET` requests with a `Range` header against blob URLs via the cache endpoint and confirming both the `206` responses and the small downloaded byte counts.

[25] *Kubernetes Documentation: Overview.* URL: `https://kubernetes.io/docs/concepts/overview/` (visited on Mar. 1, 2026).

[26] *K3s Documentation: K3s - Lightweight Kubernetes.* URL: `https://docs.k3s.io/` (visited on Mar. 1, 2026).

[27] *containerd Documentation: Content Flow.* URL: `https://raw.githubusercontent.com/containerd/containerd/main/docs/content-flow.md` (visited on Feb. 28, 2026).

4 OCI vs SIF Workflow in Practice

4.1 Benchmark Methodology

As defined in the research question in Section 1.2, this work aims to quantify the cost of the OCI→SIF workflow compared to native OCI handling. In practice, this leads to the question of how costly frequent small image updates in HPC versus cloud-native workflows are.

This leads to the benchmark concept "Patch Tuesday". The idea is to simulate repeated deployments of the same analytics workload with tiny "patch" updates. We create five image versions (v1–v5) of an Extract, Transform, Load (ETL)-related workload, where only `version.txt` changes and is placed in the last layer (an intentionally minimal OCI delta). Both clusters pull the same images from Docker Hub and run them within their respective infrastructure. We measure the wall-clock time of different phases of the runs.

Given the native OCI image usage capabilities described in Section 3.3, the K3s cluster with containerd should benefit from OCI layer reuse. This means that, after the first pull, only the changed layer should be distributed across the cluster. The HPC-like cluster with Slurm and the conversion to the SIF format as described in Section 2.2 and 3.2 should be penalized, because each new OCI digest triggers a new OCI→SIF conversion.

4.1.1 Workload Definition

The workload is defined in a `Dockerfile`. The image versions are built locally on our laptop and are made available to both clusters via Docker Hub. The created workload container is a proxy for an analytical repository that could run as containerized workload. As the base layer, there is a lightweight Python image. The next layer contains a stack of scientific libraries, in our case NumPy, Pandas, and PyArrow.

The next layers defined in the `Dockerfile` are used for multi-layer shard generation. We create 8 layers, each containing approximately 16 MiB of small, relatively incompressible binary shards produced via a deterministic SHA-256 stream, which yields a total shard payload of $8 \times 16\,\text{MiB} = 128\,\text{MiB}$. The intent is for these many small files to mimic "pipeline asset" artifacts (e.g. per-source configurations, schemas, and reference shards), which emulate a metadata-heavy analytics container. This setup is designed to make the OCI→SIF conversion non-trivial.

The next layer defined in the `Dockerfile` is an "ETL worker", which is a Python script that reads pre-generated shards from the NFS, computes aggregations and hashes on them, and writes the results back to the NFS.

Finally, as the last layer, the `version.txt` is added. This simulates the tiny "patch" update of the image. When creating the five image versions (v1–v5), we just write a different string ("v1", ..., "v5") into the text-file and build an image each time. Because only the content of the last layer changes, only the last layer gets a new digest (as described in Section 2.1).

4.1.2 Orchestration and Measurement

The benchmark is executed under controlled cache conditions. On the `hpc-head`, we clear the Apptainer cache as well as deleting all SIF artifacts related to the workload image from the NFS. We also drop the Linux filesystem caches. For the K3s cluster, we clear the pull-through cache on `k8s-cp` as well as deleting all images present on the worker nodes (`k8s-w1` and `k8s-w2`). We also clear the filesystem caches.

For each image version `v1-v5`, we perform two consecutive phases: (i) An image preparation phase and (ii) a workload execution phase. In the preparation phase, the image is made available on all worker nodes and the wall-clock duration of this step is recorded. In the execution phase, the workload is launched and we record the end-to-end job wall time until completion. After finishing v1, we proceed directly with `v2` until `v5` without resetting caches, so that incremental update behavior (layer reuse vs. repackaging) is reflected in the measured preparation time.

Here, the workload configuration is the same on both clusters. We run two container instances of an image version in parallel, with one instance on each of the two worker nodes that each cluster has. Via the job definition, each task (Kubernetes Pod or Slurm task) is configured to request and be allocated 1 CPU core and 1 GB of RAM. Because the two ETL worker instances run in parallel, the dataset is partitioned by task index so that each worker processes a distinct subset of shards. Since we measure the total job duration rather than individual node runtimes (see below), the end-to-end job time corresponds to the job's makespan. This implies that we use the maximum of the task runtimes.

We also made sure that both clusters were in idle state.

For Slurm and Apptainer in the HPC-like cluster, the pull-and-distribute phase is realized via a script, that runs Apptainer to pull the OCI image and convert it to a SIF image. The SIF artifact is placed on the NFS, where the worker nodes can access it. The script writes log files to NFS with the elapsed wall-clock duration of the pull-and-conversion step.

In the run phase, a script schedules a 2-node Slurm job for the parallel execution of the two containerized worker instances.The timings for the run, such as `Start` or `ElapsedRaw` are taken from the Slurm accounting export (`exec_sacct.psv`). Prior to the measured ETL run, an optional warmup step forces both allocated nodes to read the SIF from the NFS to standardize cache state. However, the reported run-time measurements refer to the ETL job itself.

For containerd in the K3s cluster, the pull-and-distribute phase is realized via a DaemonSet, that guarantees the image to be present on each node by pulling it. The DaemonSet is applied by a script, and the prefetch time is measured from applying the manifest until the DaemonSet rollout has completed.

In the run phase, a script applies the Kubernetes Job that defines the two containerized ETL-workload instances. Again, the time is measured when applying the job and directly after the job finishes (we use `kubectl wait` with the condition that the job is completed, which blocks the script's next command, i.e. the time measurement instruction).

4.2 Benchmark Results

The benchmark results in Figure 2 show the wall-clock times of the pull-and-distribute phase and run phase for both clusters across the different image versions. They show that, the ETL workload runtime is generally stable on both systems across all image versions.

For image version `v1`, the combined wall-clock time on the Kubernetes cluster is slightly higher than on the HPC cluster. From `v2` to `v5`, the Kubernetes measurements show a clear reduction in the pull-and-distribute phase compared to `v1`, while the run phase remains at a similar level. On the HPC system, the pull-and-distribute phase varies only slightly between `v1` and `v5` and stays in a similar range across versions.

As a result, the gap in total wall-clock time between Kubernetes and HPC is small for `v1` but increases for later versions, driven by differences in the pull-and-distribute phase. The standard deviation across repetitions (reported in the caption) indicates low variability for the pull-and-distribute phase and higher variability for the run phase, while the overall trends across versions remain consistent.

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

Figure 2: Patch Tuesday benchmark results.

4.3 Benchmark Discussion

In Figure 2, the main separation between the two stacks occurs in the image preparation part of the workflow. After running the initial cold version, small incremental updates of the last layer (image versions `v2` to `v5`) can be distributed within the Kubernetes stack much more quickly. This is, because it consumes OCI images in their layered form during execution, as described in Section 2.1 and 3.3. The same representation that is stored in the registry is also what is distributed and used on the nodes. In practice, only the small last layer containing the new `version.txt` file needs to be pulled and distributed across the cluster, before the next run starts. This shows in the minimal distribution time of only a few seconds.

In the HPC stack used here, the workflow centers around a SIF file as the execution artifact. This means in practice, that the HPC cluster can pull the same small updated OCI layer as the Kubernetes cluster, but needs to do all the steps described in Section 2.2 to create the SIF file. Given that we defined the image with 128 MiB of many small and relatively incompressible binary shards (see Section 4.1.1) the disadvantage of this execution method becomes clearly visible.

From a practical workflow perspective, this leads to a clear trade-off. The SIF file, as a single read-only artifact can be placed on shared storage and mounted consistently across many nodes and many concurrent jobs. This fits typical HPC operational patterns very well (shared filesystems, many parallel ranks, minimal per-node state). It avoids having to distribute and keep track of multiple OCI layer blobs and their metadata across the cluster.

At the same time, SIF becomes an additional artifact that must be generated, versioned, stored, and cleaned up alongside the original OCI image references. This can be particularly relevant in contexts where image preparation dominates end-to-end turnaround time, for example for short jobs, frequent redeployments, or cold-start situations. It can also be relevant whenever caches are ineffective (e.g. cache eviction, new nodes joining the cluster, or a larger-than-usual update layer).

4.4 Recent Developments: Native OCI support in Slurm

A recent trend on the HPC side is that Slurm is moving toward workflows where jobs can run OCI images directly, instead of requiring an OCI→SIF conversion step before execution[28]. Since Slurm 21.08 (Aug. 2021), Slurm can run OCI images directly, and since Slurm 23.02 (Feb. 2023), this workflow has been complemented by the `scrun` command[29].

Users can keep using familiar tools such as Docker or Podman for image handling (pulling from registries, tags, registry authentication), while `scrun` acts as an OCI runtime frontend, that launches the container via Slurm on the allocated compute nodes[30][31]. A practical limitation is that Slurm does not provide Kubernetes-like image distribution. The OCI bundle must still be staged on the compute nodes by the site (e.g. via a shared filesystem or managed caching)[32]. Overall, native OCI support in Slurm reduces format friction compared to OCI→SIF workflows while keeping execution under scheduler control.

4.5 Feedback Discussion: Absolute Benchmark Times

After the presentation, the absolute values of the pull-and-distribute times were questioned, in particular the ≈ 60 seconds observed for both stacks in the first version, for an

[28] *Containers Guide - Slurm Workload Manager*. URL: `https://slurm.schedmd.com/containers.html` (visited on Jan. 14, 2026).

[29] *Containers in Slurm)*. URL: `https://slurm.schedmd.com/SC24/Containers.pdf` (visited on Jan. 17, 2026).

[30] *Containers Guide - Slurm Workload Manager*. URL: `https://slurm.schedmd.com/containers.html` (visited on Jan. 14, 2026).

[31] *Containers in Slurm)*. URL: `https://slurm.schedmd.com/SC24/Containers.pdf` (visited on Jan. 17, 2026).

[32] *Containers Guide - Slurm Workload Manager*. URL: `https://slurm.schedmd.com/containers.html` (visited on Jan. 14, 2026).

image that has a compressed size of 258.05 MB on Docker Hub[33]. To put these numbers in context, we performed a small set of additional timing experiments. The goal was to establish (i) an optimistic baseline for "download + extraction", (ii) where the time is spent in the Apptainer OCI→SIF path, and (iii) a containerd-native "floor" that is closer to what K3s actually does during image materialization.

As a simplified baseline on the K3s control-plane node, we pulled the image into a local OCI layout using `skopeo`[34] and then extracted the layer tarballs into a root filesystem directory. On `k8s-cp`, this resulted in 7.9 s wall time for the download step and 4.1 s for extraction (about 12 s total). This baseline is intentionally optimistic, because it captures only transfer and plain filesystem extraction on a single node. It also does not represent the full containerd/Kubernetes execution path (e.g. snapshotter semantics and rollout-related work are not included).

On the HPC testbed, we drilled down into the Apptainer workflow by separating the OCI→SIF process into two explicit phases under cold-cache conditions. In the first phase, the OCI image was fetched from Docker Hub and unpacked into an intermediate filesystem directory (a sandbox), which took 16.8 s. In the second phase, this intermediate filesystem tree was repackaged into a single SIF container file, which took 51.1 s. In this split measurement, the repackaging step clearly dominates the overall preparation time. This is consistent with the benchmark observation that the HPC pull-and-distribute phase remains near ≈ 60 s even when the change between image versions is limited to a small last-layer patch.

Finally, to approximate "what the cluster really does" on Kubernetes more closely, we timed containerd-level image materialization directly on a K3s worker node. Instead of using a Kubernetes rollout or a higher-level Container Runtime Interface (CRI) pull, we interacted with the underlying container runtime (containerd) via the K3s-provided `ctr` interface and separated the workflow into two phases. The first phase is fetching the image `v1` into the containerd content store, and the second phase is unpacking and applying it in the snapshotter. For a cold pull-through-cache path, we observed 25.4 s for fetch and 18.6 s for unpack (about 44 s total). Directly repeating the same measurement with image `v2` (the patched image variant with only a minimal last-layer change), the split measured 3.8 s (fetch) and 1.6 s (unpack), i.e. 5.4 s total.

Overall, these additionally recorded timings suggest that the benchmark values are mainly shaped by the image-handling mechanisms used by the clusters, not by a bare "download + unpack" baseline. The containerd-level measurements on the K3s worker and the Apptainer OCI→SIF split are much closer to the benchmark pull-and-distribute times than the optimistic `skopeo`+extract result on `k8s-cp`. However, this still does not answer whether the cluster components are efficient in absolute terms. Both stacks do additional work beyond plain pulling and extraction, and judging whether the remaining overhead is "good" would require broader comparisons and tuning experiments, which are outside the scope of this work.

[33] *Patch Tuesday benchmark Docker Hub.* URL: `https://hub.docker.com/r/dariusheereuni/nthpda-w02/tags` (visited on Mar. 3, 2026).

[34] *Skopeo Documentation.* URL: `https://github.com/containers/skopeo` (visited on Mar. 3, 2026).

5 Lazy Pulling with Snapshotters

5.1 Benchmark Methodology

Building on the research question outlined in Section 1.2, this section investigates and quantifies how lazy pulling can shorten the time to first result for OCI-containerized analytical workloads. In contrast to the end-to-end workflow comparison in Section 4, we isolate the image delivery mechanism within the cloud-like Kubernetes stack and compare regular OCI pulls against lazy pulling of eStargz-formatted images.

This leads to our benchmark concept. The idea is to create different image versions (`small`, `medium`, `large`) of a three-stage analytical containerized workload, where each workload writes a timestamp (a checkpoint) after finishing a stage. Checkpoint 1 is reached after an ingestion step, checkpoint 2 after a transform step, and checkpoint 3 after a final report step. The Kubernetes cluster pulls and runs all image versions twice: Once using the K3s default snapshotter (`overlayfs`) on the regular OCI images, and once using the stargz snapshotter on images that were previously converted to the eStargz format.

We focus on checkpointed stages, and lazy pulling allows container execution to begin while only a subset of the image files is present (as described in Section 2.3 and 3.3). As a result, the eStargz + stargz combination should have an advantage in time to first result and intermediate checkpoint times. Consequently, the expected benefit should grow with increasing image size, especially when the image contains large "asset" artifacts that are only needed later in the pipeline.

5.1.1 Workload Definition

The workload is defined in a `Dockerfile`. The image variants are built locally and published to Docker Hub, from where they are pulled by the Kubernetes cluster. The container represents a small analytical three-stage pipeline that writes checkpoint timestamps after each stage to enable time to first result measurements.

As the base layer, the image uses a lightweight Python runtime. The next layer installs the scientific Python stack needed for the later stages of the pipeline (NumPy and Pandas). The next layer adds the workload assets. The key design choice is, that this layer contains a small ingestion asset that is constant across all versions, and a larger reference asset whose size defines the image tier: `small` (≈ 40 MiB), `medium` (≈ 120 MiB), and `large` (≈ 200 MiB). This setup ensures that the early pipeline stage can run with minimal data, while the final stage requires a large "cold" artifact. In the final layers, the pipeline code is added, consisting of an ingestion, transform, and report stage.

At runtime, the three stages are executed sequentially, and each stage writes its checkpoint timestamp to the NFS output directory. Checkpoint 1 is written by the ingestion stage after ingestion (this stage only uses basic Python and no other libraries), checkpoint 2 by the transform stage after the transform step (including NumPy/Pandas usage), and checkpoint 3 by the report stage after the final report step. This stage accesses the large, tier-defining reference asset. Just like in the workload definition for the previous benchmark in Section 4.1.1, the pipelines perform proxy ETL work on prepared data shards on the NFS.

For each tier (`small`/`medium`/`large`), we publish two image variants to Docker Hub:

A regular OCI image and a corresponding eStargz-converted image. As described in Section 2.3, eStargz preserves OCI compatibility, but reorganizes layer blobs such that file offsets can be resolved via an embedded index, enabling the stargz snapshotter to fetch only the required parts of a layer on demand.

5.1.2 Orchestration and Measurement

The benchmark is executed on the cloud-like Kubernetes cluster under controlled cache conditions. The pull-through cache on `k8s-cp` is kept warm across the whole benchmark. This means that required image blobs are served from the local registry cache rather than from Docker Hub. For every run, we enforce a cold state on the worker nodes. In practice, before each repetition, we remove all benchmark-related images from the workers and drop the Linux filesystem caches, so that each run triggers a fresh image fetch and unpack on the node, but from the warm pull-through cache. We also ensure that the cluster is idle during measurements.

For each image tier (`small`/`medium`/`large`), we run two configurations: the regular OCI image using the default containerd snapshotter (`overlayfs`) and the corresponding eStargz-converted image using the stargz snapshotter. Each configuration is executed $n = 3$ times. Each run is submitted as a Kubernetes Job that launches a single container instance. The container writes its checkpoint files to a run-specific output directory on the shared NFS mount, so that checkpoint timestamps can be collected after completion.

We measure wall-clock checkpoint times relative to a common baseline, CP0. CP0 is defined as the job submission timestamp recorded immediately before applying the Job manifest. The workload then produces checkpoint files for CP1, CP2, and CP3, each containing an end timestamp written after finishing the respective stage. The reported metrics are $\Delta t_{CPx} = t_{CPx} - t_{CP0}$ for $x \in \{1, 2, 3\}$. As a result, Δt_{CP1} captures time to first result, while Δt_{CP2} and Δt_{CP3} capture time to intermediate and final pipeline completion. We aggregate these values across repetitions and report the mean and variability per image tier and pull mode.

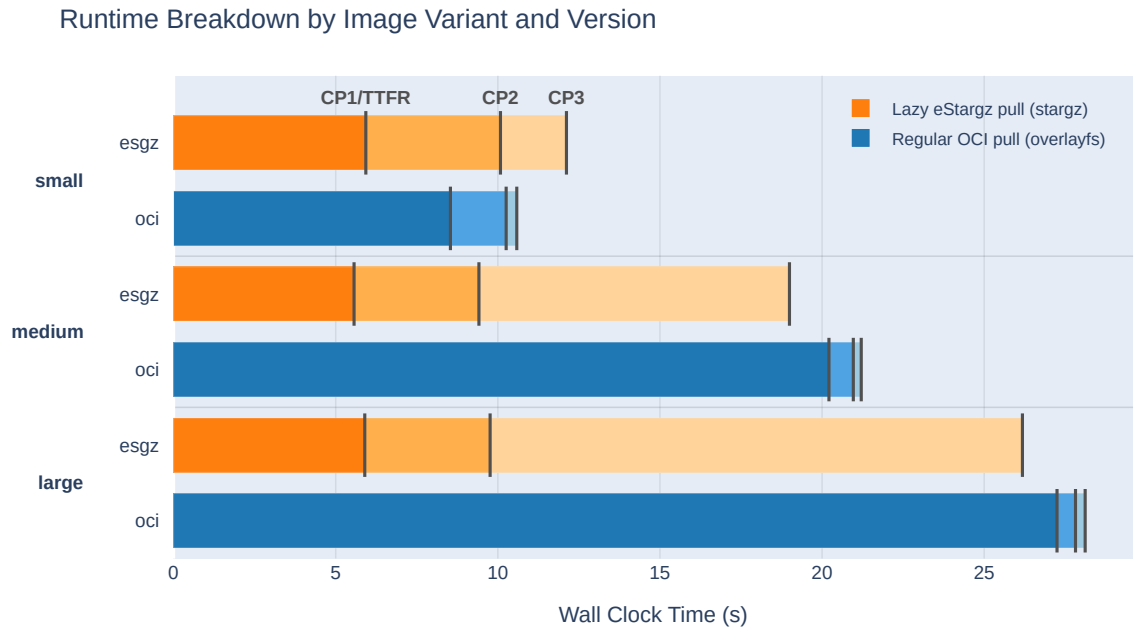
5.2 Benchmark Results

The benchmark results in Figure 3 show the wall-clock times to reach the three checkpoints for each image tier and pull mode. Checkpoint times are reported relative to CP0.

Across all three tiers, the eStargz variant reaches the first checkpoint (CP1 / time to first result) earlier than the regular OCI variant. This effect becomes most pronounced as image size increases. For the `large` tier, CP1 is reached after around 6 s and CP2 after roughly 10 s with eStargz, while the OCI baseline reaches CP1 only after 27 s (and CP2 shortly after). The `medium` tier shows the same pattern, with eStargz reaching CP1 and CP2 substantially earlier than OCI. In addition to earlier intermediate checkpoints, the end-to-end time to CP3 is also lower for eStargz in the `medium` and `large` tiers.

For the `small` tier, the picture is less clear: While eStargz still reaches CP1 earlier, the total time to CP3 is slightly higher than for OCI. At the same time, the `small` OCI results exhibit a substantially higher variability across repetitions (SD = 6.48 s, averaged over checkpoints) compared to the other configurations (see caption), which indicates,

that the `small` tier is noticeably noisier in this measurement setup.



SD ($n = 3$, averaged over checkpoints per image version): eStargz small 0.82, medium 0.29, large 0.31; OCI small 6.48, medium 0.92, large 0.25.

Figure 3: Benchmark results comparing lazy pulling and full pulling.

5.3 Benchmark Discussion

In Figure 3, the main separation between the two pulling methods appears at the early checkpoints (CP1/CP2) and grows with image size. This behavior matches the lazy pulling mechanism described in Section 2.3. With the default `overlayfs` snapshotter, containerd must complete a full pull and unpack of the image layers before the container can start. This means that CP1 and CP2 are delayed by the full image delivery cost. In contrast, with the stargz snapshotter and eStargz layers, the container can start after fetching only a small initial working set (including the eStargz index and prefetch region), while the remaining layer contents are retrieved on demand as files are accessed. As a result, the workload can reach CP1 and CP2 earlier because the first stages touch only a subset of the image filesystem, while the reference artifact, that defines the tier, is only needed later.

This means, that under lazy pulling, the large artifact remains mostly "cold" until CP3, so its transfer can be deferred and partially overlapped with computation. This also explains why, for `medium` and `large`, the eStargz variant also reduces the time to CP3. Parts of the image delivery happen in parallel with the earlier pipeline stages, instead of strictly before execution.

For the `small` tier, the results are less clear, and the effect at CP3 is even inverted. A possible explanation is, that the fixed overhead of the snapshotter-based pulling (additional indirections, index handling, and on-demand fetching) becomes relatively more

significant once the image payload is small. In addition, the `small` OCI configuration has a substantially higher run-to-run variability of 6.48 s (see the SD in the figure caption). This suggests, that the observed differences at this tier are influenced by noise in the measurement setup, rather than only by the pulling mechanism. A more fine-grained benchmark (with more tracked stages within the process) would be needed to explain the variance reliably.

Overall, the benchmark supports the conclusion that lazy pulling via snapshotters can significantly improve the time to first result for analytical workloads, when large parts of the image are not needed in early stages. At the same time, lazy pulling introduces additional moving parts and potential overhead, which can become visible for small images. It further requires maintaining a compatible snapshotter deployment. For the stargz approach used here, an additional image preparation step to publish an eStargz-formatted variant is required.

5.4 Recent Developments: Lazy Pulling in Scientific Context

A notable example of snapshotter-based lazy pulling in scientific computing is the CERN ecosystem around CernVM File System (CVMFS). CVMFS is a distributed, read-only software delivery filesystem. In a container context, a CVMFS snapshotter can realize lazy pulling at the file level. Instead of downloading and unpacking complete image layers upfront, clients fetch only the files that are actually accessed and cache them via the CVMFS/Squid infrastructure [MLB21]. Compared to stargz, this is more fine-grained because files are fetched individually rather than via chunked range reads from compressed layer blobs. Operationally, this can make large-scale, many-parallel-job launches more feasible without saturating shared network links, particularly for workloads where only a small working set of files is needed early [MLB21].

In addition to individual approaches, recent work has also compared different snapshotter mechanisms side-by-side in scientific workloads. Fatouros et al. evaluate multiple containerd snapshotters (`overlayfs` as baseline, CVMFS, stargz, and SOCI) and report that for workloads that touch only a small subset of files, the transferred data can drop to a small fraction of a full pull ((roughly 0.6% to 37%, depending on workload and snapshotter)) [Fat+25]. However, it should be noted that these results represent a best-case scenario for lazy pulling. The benchmarked workloads, like starting a shell or printing from Python intentionally access only a small subset of the image [Fat+25]. From an adoption perspective, the approaches also have different operational requirements. In practice, stargz requires publishing an eStargz-converted image variant as used in our benchmark. SOCI keeps the OCI image unchanged, but relies on an additional index artifact. CVMFS builds on a file-distribution and caching infrastructure that is typically available in the CERN ecosystem, but not necessarily in generic deployments [Fat+25].

6 Cross-Topic Synthesis

6.1 Answers to the Research Questions

This work addressed two main research questions: (RQ1) What overhead the OCI→SIF conversion workflow introduces compared to native OCI handling in containerized HPDA environments, and (RQ2) under which conditions lazy pulling can reduce the time to first result for OCI-containerized analytical workloads.

For RQ1, the Patch Tuesday benchmark in Sections 4.1–4.3 shows that the main penalty of the Slurm/Apptainer path lies in image preparation rather than in workload execution itself. While native OCI handling in K3s/containerd benefits from layer reuse and fast distribution of small image changes, the OCI→SIF workflow always requires rebuilding a new SIF artifact. This also introduces an additional artifact that must be managed operationally. Thus, in the evaluated short-job and frequent-update scenario, the SIF-based workflow causes a noticeable turnaround-time disadvantage. At the same time, the recent developments discussed in Section 4.4 further support the broader trend toward more native OCI handling in Slurm-based HPC environments.

For RQ2, the lazy-pulling benchmark in Sections 5.1–5.3 shows, that snapshotter-based lazy pulling improves startup behavior by allowing execution to begin before the full image has been transferred. The benefit is most visible for the `medium` and `large` image tiers, whereas for small images the advantage is limited. Taken together with the academic-context discussion in Section 5.4, the results indicate that the gains of lazy pulling can be very large, but depend strongly on constructing workloads that match the lazy pulling mechanics. This means that, ideally, only a small subset of the container image should be accessed early.

6.2 Comparative Synthesis

Both experiments show that container-image handling becomes especially relevant when the workload runtime is short relative to the image preparation or delivery process. In such cases, end-to-end turnaround time is shaped less by the analytical computation itself and more by how quickly container contents can be made available. The OCI→SIF benchmark in Section 4 highlights this effect at the update and preparation stage, where repeated conversion introduces overhead before execution can start. The lazy pulling benchmark in Section 5 highlights the same general issue at the startup stage, where parts of the image-delivery cost can be deferred until the data is actually accessed. Taken together, the results show that both topics address the same broader bottleneck, which is that container logistics can dominate short and iterative HPDA workflows.

6.3 Limitations and Future Work

A key limitation of both experiments is, that the observed effects are most relevant for short-running workloads, where image handling contributes substantially to the total turnaround time. For long-running, compute-dominated jobs, these overheads are likely to be amortized. In addition, all benchmarks were conducted on OpenStack-based virtual machines and were constructed to highlight the respective mechanisms of interest, so future work should examine larger-scale and more representative HPC workloads to assess the practical relevance of these optimizations.

7 Conclusion

This report examined two current trends in containers for HPC, both centered on the question of how container-image handling affects practical workload execution. The first part compared the HPC-related OCI→SIF workflow with native OCI handling, while the second part investigated snapshotter-based lazy pulling as a way to reduce startup delays. Across both topics, the results show that container performance in HPDA settings is determined not only by application runtime, but also by how images are prepared, transferred, and made available at execution time.

For the OCI→SIF comparison, the benchmarks showed that the main disadvantage of the Slurm/Apptainer path arises before the workload itself starts: Repeated conversion and redistribution create additional turnaround-time overhead, especially in short-job and frequent-update scenarios. In contrast, native OCI handling benefits from layer reuse and more incremental image distribution. For lazy pulling, the results showed that substantial improvements in time to first result are possible when workloads are constructed in a way that aligns with the mechanism. That is, when only a small subset of image contents is required early. At the same time, these findings are mainly relevant in scenarios where the image-handling time is of the same order of magnitude as the workload runtime itself.

Overall, the report suggests that an important current trend in containers for HPC is a shift toward more OCI-native, incremental, and demand-driven image handling. Rather than treating container execution artifacts as fully materialized and static, newer approaches increasingly aim to reduce unnecessary upfront work. However, the practical value of these developments depends on workload characteristics and deployment context. For this reason, future work should further investigate these mechanisms in larger-scale and more representative HPC environments.

References

[CY19] 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.

[Fat+25] Max Fatouros, Derek Feichtinger, Clemens Lange, et al. "Efficient and fast container execution using image snapshotters". In: *EPJ Web of Conferences*. Vol. 337. EDP Sciences. 2025, p. 01197.

[KB22] Rafael Keller Tesser and Edson Borin. "Containers in HPC: a survey". In: *J. Supercomput.* 79.5 (Oct. 2022), pp. 5759–5827. ISSN: 0920-8542. DOI: 10.1007/s11227-022-04848-y. URL: https://doi.org/10.1007/s11227-022-04848-y.

[MLB21] 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.

[ZZH23] Naweiluo Zhou, Huan Zhou, and Dennis Hoppe. "Containerization for High Performance Computing Systems: Survey and Prospects". In: *IEEE Transactions on Software Engineering* 49.4 (Apr. 2023), pp. 2722–2740. ISSN: 2326-3881. DOI: 10.1109/tse.2022.3229221. URL: http://dx.doi.org/10.1109/TSE.2022.3229221.