Survey of adaptive containerization architectures for HPC

Nina Mujkanovic*
nina.mujkanovic@hpe.com
HPE HPC/AI EMEA Research Lab
Basel, BS, Switzerland

Juan J. Durillo
Nicolay J. Hammer
durillo@lrz.de
hammer@lrz.de
Leibniz Supercomputing Centre
Munich, Bavaria, Germany

Tiziano Müller*
tiziano.mueller@hpe.com
HPE HPC/AI EMEA Research Lab
Basel, BS, Switzerland

ABSTRACT

Containers offer an array of advantages that benefit research reproducibility and portability. As container tools mature, container security improves, and high-performance computing (HPC) and cloud system tools converge, supercomputing centers are increasingly integrating containers into their workflows. Despite this, most research into containers remains focused on cloud environments.

We consider an adaptive containerization architecture approach, in which each component chosen represents the tool best adapted to the given system and site requirements, with a focus on accelerating the deployment of applications and workflows on HPC systems using containers. To this end, we discuss the HPC specific requirements regarding container tools, and analyze the entire containerization stack, including container engines and registries, in-depth. Finally, we consider various orchestrator and HPC workload manager integration scenarios, including Workload Manager (WLM) in Kubernetes, Kubernetes in WLM, and bridged scenarios. We present a proof-of-concept approach to a Kubernetes Agent in a WLM allocation.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → *Operating systems*; • **Security and privacy** → **Operating systems security**.

KEYWORDS

Containers,HPC,High performance computing,Kubernetes,Survey

ACM Reference Format:
Nina Mujkanovic, Juan J. Durillo, Nicolay J. Hammer, and Tiziano Müller. 2023. Survey of adaptive containerization architectures for HPC. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3624062.3624588

*Both authors contributed equally to this research.

SC-W 2023, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0785-8/23/11…$15.00
https://doi.org/10.1145/3624062.3624588

1 INTRODUCTION

The building blocks for containers were laid during the development of Unix V7 in 1979, when chroot - the possibility to change the root directory of a process and its children - was introduced. The design of cgroups or "Control Groups" in 2006 added the possibility for limiting, accounting, and isolating resource usage of a collection of processes. In 2008, using chroot, cgroups, and Linux namespaces, LXC (LinuX Containers) were implemented as the first, most complete version of what we today consider a standard container manager. It wasn't until the development of a container engine - Docker, in 2013 - and cloud computing systems that containers truly exploded in popularity.

Since then, as container tools matured, container security improved, and HPC and cloud system tools converged, supercomputing centers have integrated containers into their workflows [4, 47, 48]. Containers offer an array of advantages, such as greater efficiency than full hardware-level virtualization, support for DevOps, management of on-demand resources, scalability across resources, and improved portability of workflows. All of this in turn enhances the reproducibility of scientific research, as well as the cooperation between researchers.

Research on containers focused thus far mostly on cloud solutions [50], orchestration in cloud environments [2], and benchmarking containers running in cloud systems versus bare metal [23]. With the emergence of more heterogeneous systems, and convergence between HPC, cloud, and edge, more research has been performed on the use and integration of container technologies and orchestrators in HPC environments [15, 49].

In this paper, we perform an in-depth analysis of container technologies with a focus on HPC, integration into an HPC environment, performance, and ease-of-use. Further, we describe integration scenarios taking the limitations of HPC sites into account. Finally, we consider an orchestrator integration scenario, specifically that of merging Kubernetes into an HPC workload management environment, with a proof-of-concept of a possible Kubernetes-in-`Slurm` [26] integration.

We suggest an approach to portable but performant containers in HPC that we term *adaptive containerization*, and which aims to fulfill user, system, and site specific needs by choosing the best adapted tools for each component of the containerization architecture and merging them into an integrated workflow. Adaptive containerization focuses on accelerating the deployment of applications and workflows using containers while distributing some of the maintenance burden and providing additional possibilities to

the research scientist. Essentially, it includes the integration of HPC-centric and specific container engines, registries, and orchestration tools, to deliver full workflow capabilities to an end user.

To our knowledge, while there have been papers with the goal of creating taxonomies over all container applications available for the cloud or HPC/cloud environment [6, 34], empirical analyses of select aspects of container technologies [1], and surveys delving into the similarities and differences between containers deployed on cloud and HPC environments [50], this is the first analysis of the entire containerization, registry, and orchestration stack with a purely HPC-centric focus.

The paper is organized as follows: section 2 discusses the motivation to this work, in section 3 we introduce relevant terminology, as well as HPC specific requirements to containerization. An in-depth analysis of container engines and container registries for HPC is performed in sections 4 and 5, respectively. In section 6 we discuss Kubernetes integration scenarios. Finally, we conclude this paper and give an outlook on future challenges in section 7.

2 MOTIVATION

Building software in a reproducible fashion requires absolute control over either the build environment or the build systems involved. The responsibility to correctly specify and verify dependencies, such as which library to link against, is placed on the user. Ensuring for example consistent use of threaded vs non-threaded libraries becomes crucial for the code to execute correctly. Failure to ensure this consistency can lead to runtime errors that are difficult to track, or may even return adulterated results, despite integration testing.

Correctly applied, containers provide a stable and controlled environment for wrapping software or even entire scientific software stacks into a portable unit. Packaging these portable units in a standardized way makes it possible to write workflows with dependencies on specific containers, rather than specific execution environments. This is in particular exploited by the bioinformatics [46] and data science [33] communities, which use multiple tools with sometimes competing build and runtime environment requirements in complex data processing pipelines.

Thus, within an HPC environment, containers aid in solving the problems of package build systems by:

- enforcing a code-based approach to the build environment and the software packaged by it via the use of container specs such as Dockerfiles or Singularity definitions,
- controlling the build environment such that there is only one library variant available and the software can either build against it or fail at the linker step,
- allowing more lenient package build systems, as often found in scientific software,
- and bundling multiple packages into one consistent directory tree, made relocatable with the help of the container engine.

As applications are not necessarily linked statically and may need additional support files such as parameter sets, configurations, model data, etc., most applications must be run within the container they were built in or, ideally, within a pared-down version of it. These applications are fully decoupled from the system libraries on the host operating system, increasing portability. The clear advantage is that the host operating system can be updated independently, reducing the system administration burden. The drawbacks include the containers not profiting from security, bugfix, or performance updates performed on the host operating system. This mandates the use of Continuous Integration/Continuous Delivery (CI/CD) systems for container update automation, and a service or registry for management and sharing. An efficient formulation of regression tests can for example be done with a software package like ReFrame [8].

An in-depth analysis of the possible container engines, registries, and orchestration tools aids in the selection of the most fitting tools to integrate into a full HPC adaptive containerization architecture based on site requirements.

3 CONCEPTS

The containerization stack consists of multiple components on multiple levels which must be defined. To facilitate the discussion of adaptive containerization, we must further define the set of requirements posed by HPC sites to ensure security and efficiency for all its users.

3.1 Terminology

At the lowest level is the *image*, an immutable file composed of source code, libraries, and dependencies necessary for an application to run. A container image is not a container and cannot be executed. To run an image, it must be unpacked into a container - a runtime instantiation usually isolated using Linux cgroups.

Container images can be stored in a *registry* that is used to manage, push, and pull images. Registries store container *repositories*, collections of related images that may have the same name and are differentiated by their tags or alphanumeric identifiers.

The data inside a container image is usually organized in layers. A *layer* captures changes in the filesystem compared to the previous layer, and is identified by a hash calculated from the data in that layer. Layer deduplication can be employed in registries and locally based on equal hashes (*content-addressable storage*).

At the highest level, we differentiate between the container *engines* and the container *runtime interface* (CRI). Container *engines* include Docker, Podman, Sarus, and many others. They permit the user to make requests regarding container images via a user-facing component. These requests may include image pulls from a registry, signature verification, unpacking of bundles, and ascertaining the availability of required system components. The engine is not a CRI, but is responsible for calling the container runtime. The container *runtime* is a lower-level component that handles image and process management. The runtime sets up the user namespace (`UserNS`), thus starting the container process. The most popular container runtimes include `runc` [20] and `crun` [9].

The interoperability between the various engines and runtimes is standardized under the *Open Container Initiative* (OCI). The OCI defines a standard container image format, the container runtime specification, and a reference runtime implementation `runc`, which was split off from Docker. OCI-compatible runtimes have OCI Runtime Specification compliant CLI options, a configuration interface, and execute OCI hooks, a mechanism that defines entry points to inject code to be run at various phases of the container lifetime.

For building container images, as well as use cases which require more isolation based on virtualization technology, an extra set of tools and terminology is needed. Building container images comes with its own set of challenges, including restrictive site policies disallowing users from building their own; internet access restrictions complicating software builds; the need to choose between hardware optimized versus portable containers; the requirement for software to match between the container and on the host, especially with regard to MPI and accelerators; long build times and large container storage; and the issues regarding host file access from the container. Since we consider these topics beyond the scope of this paper, we hereby refer to the respective literature [5, 13, 35].

3.2 HPC Requirements

Container technologies were initially created to run services, hence the assumption that containers are started by users with root or root-like permissions. Privilege escalation on HPC systems poses a security issue, making alternative container execution models such as rootless a requirement. *Rootless* refers to the ability to start containers as an unprivileged user and avoid running binaries with root permissions in the default or initial namespace in which every process started on a Linux system runs. This execution model provides users the capability for privilege escalation only within the container, and allows access to syscalls affecting the runtime environment, while denying access to resources on the host system to which the user does not have permissions.

While the strict adherence to rootless container execution poses one of the main challenges of deploying container technologies on HPC systems, additional requirements differing from those of non-HPC systems exist.

The advantages of container technologies are achieved by restricting the container process to a subset of resources and thus introducing an interface between the host operating system and the container execution environment. This interface executes a change of the filesystem root via *chroot* or *pivot_root*; separates filesystem mounts, system IDs (uid, gid), process IDs, networks; etc. These restrictions in turn enable the runtime portability of containers.

Container or process isolation enables the running of multiple containers concurrently on the same hardware. Workloads on HPC systems are often spread across multiple nodes. As compute nodes are allocated exclusively, strict container isolation may introduce performance penalties due to increased OS overhead. Note that this penalty is in flux as the use of exclusive node allocation has been shifting for high-density systems with more than two GPUs per node, with enough cores per node, or for workflows that require direct communication between applications in different containers by means of e.g. shared memory.

Beyond the introduced overhead, strict container isolation may break access to HPC hardware such as interconnects or accelerators. Additionally, it may break software with custom inter-node communication such as e.g. workflow-based payloads that require non-MPI communication.

Filesystem access poses another set of issues. Container processes may require access to specific files on the host system, including device libraries, configuration files, license restricted files, etc. When loading host libraries for device drivers, communication, etc., ABI compatibility with the container applications and libraries must be ensured. Failure to do so may lead to errors which are hard to detect and may possibly affect scientific results. This may extend beyond the requirements of the directly involved libraries: if a host library imported into the container requires a newer version of `glibc` than present within the container, it will fail. Overriding most of the container libraries with the host supplied ones may equally introduce compatibility issues.

Interoperability with data on existing shared filesystems is important, whereas large scale container deployments in the cloud often avoid shared data or POSIX filesystems altogether. A container image contains many small files, which may be loaded from shared storage from many compute nodes, and may put strain on the cluster filesystem, slowing down startup time or even execution. This can also affect other users of the shared storage.

Software within containers must be optimized for a target architecture, thus breaking the premise of container portability. While tracking dependencies for security issues across the layers of a container is crucial for cloud service containers, it is less important for HPC containers as the applications usually do not offer a continually run service which can be exploited from arbitrary locations, thus adding a level of isolation. Nonetheless, there are attack scenarios which may require scanning images as due diligence.

Finally, on HPC systems malicious users must be assumed, and privilege escalation avoided. Possible attack vectors if a user is granted elevated privileges include the injection of malicious data in filesystem images to exploit kernel block device driver security bugs. Initial containerization solutions, such as Docker, used persistent background processes to manage images, containers, networks, or storage volumes. Spinning up such daemons on each compute node to control what is most often a single container process is wasteful and may introduce extra jitter, and increases the attack surface in the process.

The various HPC container technologies implemented to date share similar solutions to the above stated requirements:

- HPC container solutions do not require running daemons to start containers, and especially root or root-like daemons must be avoided. Tasks such as registry interactions may use daemons. A monitoring process to give enhanced control over the running process may still be involved, but must run as the same user starting the process.
- Container isolation is weakened, resulting in a setup which offers more isolation than a simple `chroot`, but less than full container isolation. A user `namespace` is created to obtain extra capabilities within the container, which are in turn used to set up separate mounts invisible to everyone beyond the real root of the host system. This enables overlaying parts of the host OS into the container environment, allowing the container processes to access host OS libraries or devices.
- Unused isolations such as `network` or `IPC` namespaces are not set up to reduce complexity and attack surface, or because they may interfere with HPC applications. HPC workload managers such as `Slurm` or the PBS family may enable job sharing on a single node via cgroups support, but are often configured for exclusive node allocation, depending on site policies.

- Host configurations are copied into the container environment, modifying it in the process. User namespacing is limited to a single user to ensure files created by processes in the container have the UID/GID of the user launching the job.
- Container filesystems are (re-)packaged as single-file images to avoid small-file load and latency, potentially providing a speedup against traditional application execution by trading memory and CPU (decompression) for disk IO.
- When a kernel driver for compressed filesystem image reading is used, care is taken that the user can never directly provide such a compressed image. Caching of such filesystem images may thus require a separate service or to be run as root, or a userspace filesystem driver must be used to avoid exposing the kernel to user generated filesystem images.

The containerization technology space is a quickly developing one. Many tools exist that enable the building and running of containers. The following analysis makes no claims to completeness, as new technologies are developed constantly. Nevertheless, great care has been taken to analyze the requirements and the solutions given by containerization technologies with a focus on HPC.

4 CONTAINER ENGINE COMPARISON

This section contains a set of tables giving an overview of features we identified as important for HPC Containers. The chosen criteria are used to evaluate the most prominent (HPC) container solutions which may be deployed without extra management services such as Kubernetes, Docker Swarm, Apache Mesos, OpenShift, Rancher, or Nomad. Additionally, we include Docker as a baseline comparison and for the sake of completeness. Tables 1, 2 and 3 contain the summarized comparison for the following discussion.

4.1 Discussion

In the following, some of the chosen evaluation criteria are defined, examined, and discussed. We refer to projects in shorthand where significant interrelations exist. Singularity, for example, diverged into two correlated projects in 2021, when Sylabs forked the Singularity project to create its community version SingularityCE, covered under the BSD License, and the original Singularity project in turn chose to rebrand itself to Apptainer and move into the Linux Foundation. While initially alignment is expected between Apptainer and SingularityCE, divergence has and will occur. Consequently, where no distinction is required between Apptainer, SingularityCE, and Sylabs' proprietary SingularityPro, we refer to them collectively as *Singularity*. Finally, note that, as Podman-HPC is a wrapper script around the Podman engine that provides additional HPC configuration on top of it, the two are not fully independent.

4.1.1 Champion and Affiliation. In the past, project affiliation was less significant to HPC sites, as they were more isolated, and compatibility with bleeding edge technology played a different role. While all listed projects are Open Source Software (OSS), the entities behind them can retain significant influence. This influence may range from limiting features to paying customers, affecting the development trajectory, up to complete project defunding.

An example of this dynamic is the interaction between Apptainer and SingularityCE. The company behind the Singularity platform, which diverged into Apptainer and SingularityCE, has released SingularityPro with additional features (e.g. Software Bill of Materials (SBOM)) and support contracts, as well as their own maintained registry platform. While it can be advantageous to externalize some of the costs of maintaining base images, it can also be seen as an attempt at the platformization of the HPC container space. Additionally, despite Apptainer incorporating changes made in SingularityCE, a quick comparison shows differences between them, with e.g. Apptainer using `runc` and SingularityCE using `crun` as their default runtimes. As Sylabs develops its business, it is safe to assume that more of the advanced features will be incorporated into SingularityPro and may not be open sourced.

4.1.2 Rootless container and FS implementation. The core principal behind rootlessness is the use of `pivot_root` instead of the classical `chroot` to provide a new root to the processes started in the container. A user gains the capability to `pivot_root` when in their own `UserNS`. Despite the user being able to assume `UID 0` inside of this new namespace, it does not permit mounting block devices or files acting as such via kernel drivers, since kernel drivers are not hardened against maliciously crafted block-device data. Therefore, a SquashFS image can only be mounted by either a setuid-root binary prior to entering the namespace, via a FUSE driver as the FUSE user-kernel interface can be assumed to be audited, or not at all, instead unpacking an image to a directory. When using the setuid-root approach, care must be taken to not only secure the binary and the image itself (e.g. on transparent conversion from an OCI image layer to a SquashFS image at runtime; the resulting image must not be user-writeable), but also to ensure that the user is unable to manipulate it while being mounted, nor inject their own image directly.

One approach that works around the limitations imposed by a shared cluster filesystem is extracting an image to a temporary, node-local storage location. This avoids the need for either user- or kernel-space filesystem drivers, thus reducing the memory and CPU overhead generated by the intermediate image.

An alternative to the namespace-based rootless mechanisms are the `fakeroot` approaches: an `LD_PRELOAD` variant, in which a library intercepting relevant system calls is loaded prior to any executable; or a variant based on the `ptrace` system call, allowing to intercept the system calls of another process. A limitation of the first approach is that it fails with static binaries, and of the second, that it introduces a significant performance penalty and the user requires access to the `CAP_SYS_PTRACE` capability.

We note that benchmarks comparing SquashFUSE and the in-kernel SquashFS show a magnitude lower IOPS for random access and a much higher latency[43]. For many compiled codes this will only be noticeable on start and when loading bundled parameter data, while for projects based on interpreted languages like Python, which consist of many small files, it will have a more noticeable effect. A similar situation can arise with a FUSE-based OverlayFS implementation, where heavy I/O must be absorbed by the CPU.

4.1.3 OCI Container and Hook support. Support for hooks becomes important if extensions such as for additional image modification or accelerator enablement are required. The OCI hooks specification, which is part of the OCI runtime spec, provides a vendor-independent way of installing and running such hooks at defined

Engine	Version	Champion	Affiliation	Runtime	Implem. Language
Docker	v24.0.5 (Jul. 24, 2023)	Docker	Docker	`runc/crun`	Go
Podman [17]	v4.6.1 (Aug. 10, 2023)	RedHat/IBM	Kubernetes	`crun/runc/Crio-O`	Go
Podman-HPC [31]	v1.0.2 (Jun. 15, 2023)	NERSC	-	`crun/runc/Crio-O`	Python, C
Shifter [21]	Git 0784ae5 (Oct. 22, 2022)	NERSC	-	Shifter	C
Sarus [3]	v1.6.0 (May 5, 2023)	CSCS	-	`runc/crun`	C++
Charliecloud [36]	v0.33 (Jun. 9, 2023)	LANL	-	Charliecloud	C
Apptainer [38]	v1.2.2 (Jul. 27, 2023)	LLNL, CIQ	Linux Foundation	`runc/crun`	Go
SingularityCE [19]	v3.11.4 (Jun. 22, 2023)	Sylabs	-	`crun/runc`	Go
ENROOT [11]	v3.4.1 (Feb. 8, 2023)	Nvidia	Nvidia	enroot	C, Bash

Engine	Rootless	Rootless-FS	Container Monitor	OCI Support	
				Hooks	Container
Docker	`UserNS`	fuse-overlayfs	per-machine (dockerd)	yes	yes
Podman	`UserNS`	fuse-overlayfs	per-container (conmon)	yes	yes
Podman-HPC	`UserNS`	SquashFUSE + fuse-overlayfs	per-container (conmon)	yes	yes
Shifter	`UserNS`	suid	no	no	yes (partial)
Sarus	`UserNS`	suid	no	yes	yes (partial)
Charliecloud	`UserNS`	Dir, SquashFUSE	no	no	yes (partial)
Apptainer	`UserNS`, fakeroot[13]	suid, fakeroot, (SquashFUSE)	per-container (conmon)	yes (manually, requires root)	yes (partial)
SingularityCE	`UserNS`, fakeroot	suid, fakeroot, SquashFUSE	per-container (conmon)	yes (manually, requires root)	yes (partial)
ENROOT	`UserNS`	Dir	no	no	yes (partial)

Table 1: Overview of container engines with supported *rootless*-techniques and OCI compatibility.

Engine	Transparent Format Conversion	(Transparent) Native Container Format Caching	Native Format Sharing	Namespacing on Execution	Signature Verification Support	Encrypted Container Support
Docker	-	-	-	full	Notary	no, extensions available
Podman	-	-	-	full	GPG, sigstore	yes
Podman-HPC	yes	yes	no	full/user and mount NS	GPG, sigstore	yes
Shifter	yes	yes	no	user and mount NS	-	no
Sarus	yes	yes	yes	user and mount NS	-	no
Charliecloud	no	-	no	user and mount NS	-	no
Apptainer	yes	yes	yes	user and mount NS, possibly others	GPG (SIF containers)	yes (SIF only, via kernel driver)
SingularityCE	yes	yes	yes	user and mount NS, possibly others	GPG (SIF containers)	yes (SIF only, via kernel driver)
ENROOT	no	-	no	user and mount NS	-	no

Table 2: Continuation of table 1 with focus on supported image formats and features.

points in the lifetime of a container without the need to modify the runtime itself. Most solutions either provide direct support for OCI hooks (in particular when relying on a mainstream runtime like `runc` or `crun`), or a custom hook framework (see plugins in Apptainer). Some container solutions, such as e.g. Shifter, rely on parts of their software being written in a scripting language, therefore making them easily extendable and more flexible, but requiring adaptions of the extensions when the base software is updated.

For OCI compatibility, it is important to note that HPC container solutions may break some of the features a container might expect to be present. The most obvious of these missing features include the lack of an isolated network namespace which permits the binding of services to arbitrary ports, or the availability of different user IDs, as often only a single one mapped directly to the original user ID is made available. Vanilla containers may thus have to be repackaged or modified before running on an HPC container system.

4.1.4 Container Format, Conversion, Caching and Registry. The Singularity Definition file `.def` is similar to the RPM spec file, and all commands to build the container can be placed in a single section, as layering is not available in the flat Singularity Image Format (SIF). The SIF integrates writable overlay data, which may be useful for bundling either models or output data with the code using or generating it. In Dockerfiles, on the other hand, manually grouping commands into layers constitutes an important concept to allow incremental container builds, updates, and deployments. It has to be noted that when integrating Podman and Singularity, as Podman is capable of running SIF containers, Singularity may be solely needed to build the container, leaving Podman to launch it.

In non-HPC containers, the OCI filesystem bundle consists of multiple layers, with each layer tracking a change to previous filesystem layers. These layers are mounted via a union mount filesystem approach - usually the Linux based OverlayFS driver - into a consistent filesystem view with only a new upper layer being writable. Running an OCI container in an HPC setting thus requires all layers to be present on a shared storage. This overlay mount capability may not be enabled on the compute nodes, or may require root privileges depending on the kernel version.

HPC cluster filesystems, or any shared filesystems, are known for not scaling well in cases of random access with many small files, as can be seen with e.g. Python or other interpreters. Even for more static workloads, there may be additional load to the shared filesystem on startup. This load is caused by the containers' base libraries, which have to be loaded into memory, and which may load additional service files that would otherwise already be in memory

due to the host operating system requiring them. For example, `libc` will have to load `/etc/nsswitch.conf` to determine the source of UIDs/GIDs, and from there the respective configuration files; or system libraries may have to load localization specifications, etc. As established earlier, one solution to work around these limitations is to flatten the OCI bundle either to a node-local directory, or to a filesystem image on a shared storage. This conversion can happen either automatically or explicitly. In the automatic case, we want this converted image to be cached to avoid repeated conversion costs (storage and time), and possibly share it between different users.

If an in-kernel driver is used to mount a filesystem image, care must be taken that the user can neither provide nor manipulate the image directly, instead using a userspace driver (SquashFUSE) to mount it, as pointed out in sections 3.2 and 4.1.2. When an in-kernel driver is used, there must be a setuid-root binary doing the conversion, caching, and sharing between multiple users, as in e.g. Sarus. It must be noted that an OverlayFS mount does not suffer from the same risks as a SquashFS mount, since the OverlayFS does not access raw block device data, but acts on the mounted filesystem instead. Both Docker and Podman switched to a FUSE-based OverlayFS `fuse-overlayfs` driver.

Since sharing container images - irrespective of format - via filesystem has several challenges (e.g. managing access via POSIX ACLs or extended ACLs, deduplication, locking), a better approach may be a separate service which can ensure the required access semantics. Sarus and Singularity support sharing their respective *native* HPC container formats, while other solutions do not and sharing always happens prior to conversion to the format, or by manual setup directly between users.

4.1.5 Container Signing and Encryption. While digital signing does not prevent the spread of malware, or protect from well-funded malicious actors, it can help uncover basic attacks in the form of name squatting, breaches of security, and make tracing software provenance possible. Docker and Podman, the industry solutions at the forefront of digital signing, implement different solutions: Docker uses the Notary (v2) tool, while Podman provides similar functionality via GPG signature attachments. Apptainer has built its signing solution on PGP as well, although only for its own SIF container, meaning that signatures for imported OCI containers are not verified. `sigstore` [41], with `cosign` [40] being the implementation for containers, is an independent approach which can be used for general software signing support, and for supporting SBOM or lists of all the open-source and third-party components present. Podman added direct support for `cosign`, but it currently requires extra setup steps.

Running code on or transferring data to external machines always requires a certain amount of trust. Container encryption can limit data access times and access privileges. When combined with secure enclaves, it can make it difficult to access data despite having total control over the hardware the data is stored on. This permits the deployment of workloads which previously had to be run on isolated systems on a supercomputer. Since this feature is still under development for most solutions, we tracked only the simplest form of support: does the runtime, resp. engine, support decryption of encrypted containers.

4.1.6 HPC-specific extensions - GPU and Accelerator Enablement, Library Hookup, WLM Integration. As mentioned in section 3.2, HPC container technologies require additional steps at container startup. This includes granting containers access to host libraries like optimized scientific libraries, device libraries for accelerators, communication, etc. Host library access can be enabled by bind-mounting host directories into the container namespace, providing extra device nodes, or granting extra capabilities to the user process. Most HPC container implementations have solutions built-in for the GPUs of the most common vendors. Other accelerators must be added via hooks or plugins. When a container gains access to host libraries, it requires a matching ABI, as a mismatch may introduce subtle errors. Some solutions like Sarus therefore contain explicit ABI compatibility checks on the libraries.

While it is possible for all container solutions to be invoked explicitly via batch scripts, proper WLM integration may be required. The WLM controls device access rights, which must be passed along to the container engine, and may restrict the capabilities available to the user (like cgroups). A more transparent container execution may be desirable in order to lower the container entrance barrier for users, and to avoid common mistakes when running containers.

As containers introduce an additional layer of indirection, some workflows using interactive access may be broken or require additional steps. This problem could be alleviated via proper WLM integration. Other use cases involving profilers and debuggers may require approaches specifically tailored to container usage in HPC.

4.1.7 Module System Integration. While it is possible to write a wrapper script to transparently start a container in which to run an application, doing so may have unexpected side-effects for the user and require additional work. With the exception of the Singularity Registry for HPC (`shpc` [42]), none of the other projects offer affiliated solutions to automatically integrate containers as modules. Despite `shpc` originating in the Singularity ecosystem, it officially supports other container solutions like `Podman` [17], although they may require additional configuration in the form of wrapper scripts.

4.1.8 Documentation. Grading of the software documentation for each engine is based on three elements - the availability and length of the documentation, the breadth and depth of the topics covered, as well as the clarity of the text. While the grading must by definition be subjective, we hope to somewhat standardize it with the given criteria. Where no documentation (e.g. Podman-HPC) or insufficient documentation was available, the corresponding slot has been marked with N/A. In all other cases, a scale ranging from + for minimal documentation available to +++ for extensive and well-organized documentation was used.

4.1.9 State of Source Code, Contributors, Community. The number of contributors and/or size of the community may serve as an indicator of the future development of a project. A small contributor base originating mostly from a single entity may change the direction of a project drastically, whereas defunding of the project within that entity can bring it to a sudden stop. We see this risk in particular for the Shifter, Sarus, Charliecloud, and Enroot projects, but also to a lesser degree for the Podman-HPC project, which is at the moment at an incubator stage. For the latter it is possible

Engine	GPU-Enablement	Accelerator Support	OS/MPI Library Hookup	WLM Integration	Contains Build Tool
Docker	via OCI hooks	via OCI hooks	via OCI hooks	no	yes
Podman	via OCI hooks	via OCI hooks	via OCI hooks	no	yes
Podman-HPC	yes	via OCI hooks or patch	yes	no	yes
Shifter	no	no	for MPICH	yes / SPANK plugin	no
Sarus	yes	via OCI hooks	yes	partially via OCI hooks	no
Charliecloud	manually	manually	manually	no (no SPANK plugin release)	no
Apptainer	yes	no	manually	no	yes
SingularityCE	yes	no	manually	no	yes
ENROOT	yes, Nvidia only	via custom hooks	via custom hooks	yes / SPANK plugin	no

Engine	Module System Integration	Documentation			# Contributors
		User	Admin	Source	
Docker	via shpc	+++	+	+	486
Podman	via shpc	+	N/A	++	461
Podman-HPC	(via shpc)	N/A	N/A	(+)	3
Shifter	no (shpc announced)	+	+	++	17
Sarus	no (shpc announced)	++	++	+	6
Charliecloud	no	+++	+	++	31
Apptainer	via shpc	++	+	+	148
SingularityCE	via shpc	++	N/A	+	130
ENROOT	no	N/A	N/A	+	9

Table 3: Summary of supported integrations for different container solutions and community analysis.

that required features will be directly ported to `Podman`. For operations, this could mean that system administrators have to take over maintenance at a lower level than initially anticipated. This must be accounted for in the risk assessment.

For an example of this dynamic, we may look at Shifter and Podman-HPC, both developed at NERSC. Based on presentations by NERSC [14], it is possible to assume that Shifter will be replaced by a solution based on `Podman`, with the Podman-HPC development serving as a testing ground for various capabilities such as intercepting layer unpacking to generate a filesystem image, using OCI hooks to setup devices and device libraries, and integrating with the WLM.

It must be stressed that the number of contributors does not paint the whole picture. In particular, while SingularityCE has fewer contributors than Apptainer, the activity in the SingularityCE repository as measured by the number of added and deleted lines for the month of November 2022 was double that of the activity of the Apptainer repository.

4.2 Summary

With regard to HPC requirements, the container technology selection space is essentially tripartite: cloud industry container tools like Docker and Podman, Singularity with their own image format, and projects trying to integrate containers without deviating too much from the cloud industry standards. Industry tools have the advantage of an immense user community with a lot of resources, while HPC-centric tools may be more accommodating to the domain scientists, such as by offering the simpler structure of the Singularity container specification files.

It is difficult to make specific assessments regarding the future-proofness of any single solution. The HPC space has been trending towards using Singularity in previous years. Big cloud providers are pushing into the HPC segment and have started to support SIF containers in their registries to ease the onboarding of scientific compute customers on their platforms. Newer tools like Podman support running containers from SIF, easing the shift from Singularity in case of adoption of Podman for existing deployments. While this Podman support does not indicate that SIF will become a standard used outside of scientific computing, it shows that there is a demand for it, and that there are implementations beyond Singularity capable of executing it. HPC initiatives like Autamus provide both kinds of image formats, anticipating the needs of Singularity alternatives relying solely on OCI containers. Together with NERSC gravitating towards Podman with the help of a small wrapper and support from RedHat, Sylabs improving the support for OCI containers for SingularityCE 4, and Apptainer gaining support for building from Dockerfiles, the longterm prospects of SIF become less certain.

5 CONTAINER REGISTRY AND CI/CD COMPARISON

An internally deployed Container Registry may be required to maintain containers, and to act as an intermediary when moving public or private containers to and from HPC systems, thus avoiding the use of SSH transfers. In this section, we compare existing registry solutions deployable either on-premise or as part of a CI/CD integration workflow. Note that, while some cloud platforms like OpenShift provide built-in registries, evaluating such a cloud platform is out-of-scope for this document, and these registries have been omitted from the comparison. We refer to tables 4 and 5 regarding the included container registry projects and the criteria discussed in the following section.

5.1 Discussion

5.1.1 Champion and Affiliation. While the landscape of container solutions and build tools is diversified, the same can not be said for registry services. Since registries can be provided as-a-Service, most registries have a company sponsored development. This increases the risk of the product taking new or unexpected directions, or of having the development model or license switched. The project

Registry	Version	Champion	Affiliation	Focus	Protocol
Quay [39]	v3.8.10 (Dec. 6 2022)	RedHat/IBM	-	Registry	OCI v2
Harbor [16]	v2.8.3 (Jul. 28, 2023)	VMWare	CNCF	Registry	OCI v2
GitLab	v16.2 (Jul. 22, 2023)	GitLab	-	Git hosting, CI/CD	OCI v2
Gitea	v1.20.2 (Jul. 29, 2023)	(OSS community)	-	Git hosting, CI/CD	OCI v2
shpc [42]	v2.1.0 (Apr. 6, 2023)	vsoch	LLNL	Registry	Library API
Hinkskalle [32]	v4.6.0 (Oct. 18, 2022)	h3kker	University of Vienna	Registry	Library API, OCI v2
zot [51]	v1.4.3 (Nov. 30, 2022)	Cisco	CNCF	Registry	OCI v1

Registry	OCI Artifact Support	Proxying	Repl./Mirroring	Storage Support	Authentication Providers
Quay	Helm charts, cosign, zstd	yes / auto	yes (pull)	FS, S3, GCS, Swift, Ceph	internal, LDAP, Keystone, OIDC, Google, GitHub
Harbor	Helm charts, cosign, user-def.	yes / auto	yes (push + pull)	FS, Azure, GCS, S3, Swift, OSS	internal, LDAP, UAA, OIDC
GitLab	no, separate pkg registries	yes / manual	no	FS, Azure, GCS, S3, Swift, OSS	LDAP
Gitea	Helm, separate pkg registries	no	no	FS, Minio/S3	internal, LDAP, PAM, Kerberos
shpc	-	no	manual (Globus)	Minio, GCS, S3	LDAP, PAM, SAML
Hinkskalle	no	no	no	FS	LDAP
zot	Helm charts, cosign, notation	no	yes (pull)	FS, S3	internal, LDAP

Table 4: List of common container registries with their respective featureset.

documentation often reflects this, as can be seen in the Project Quay documentation for example, which often refers directly to Red Hat services.

Despite both Apptainer and SingularityCE having the capability to push and pull standardized OCI containers, a separate SIF compatible registry adhering to the Library API standard may be advisable for improved integration, especially as pertains to signing, avoiding repackaging, preserving metadata, etc. Apart from the as-a-Service registry provided by Sylabs, which is not part of this comparison, the available Library API registries are maintained by single developers, and are thus likely to receive less scrutiny than the OCI registries.

5.1.2 Focus. Several CI/CD solutions offer package registries with the intention to directly host the generated build artifacts. While these solutions explicitly offer container registries, the included feature sets are limited, and the registries may thus not be suitable if requiring the ability to store, verify, and display signatures, or to accept any other artifacts than containers. We note that a broad support of the OCI standard is crucial for the development of the Adaptive Containerization feature, as it could build on user-defined OCI artifacts.

5.1.3 Proxying Capabilities and Mirroring. The most popular public OCI registry, DockerHub, introduced rate limiting in November 2020, a change quickly affecting any site with a small number of public IP addresses for a large number of clients. While layers originating from public containers can be cached on an internal registry, upstream registries may still be queried regularly when containers are (re)built, or when public containers are used on a system where at least a part of the nodes has unfettered internet access. One possibility to work around this limitation is the use of a proxy server to cache the requests.

Such proxy services can be provided by a registry implementing proxy capabilities via the transparent forwarding and caching of requests in a namespace to an upstream registry. The advantages over a common HTTP(S) proxy include detailed statistics of upstream registry usage, required disk space, image statistics, etc. Additionally, the registry mirroring capabilities can be used to either mirror hosted containers to a public registry, or to preserve remotely provided containers on the local infrastructure. Finally, deploying a site-local registry with proxy capabilities, possibly attached to a clusters' highspeed network, can support public network access scenarios without giving the login or compute nodes full internet access.

5.2 Summary

With the Library API (Singularity) registries being carried by single developers, and the CI/CD system integrated registries supporting only a subset of the possibly required features such as proxying, mirroring, or user-defined OCI artifacts, the remaining candidates for an HPC-centric container setup are Project Quay and Harbor. Harbor, a Cloud Native Computing Framework project, seems to currently enjoy broader support, despite being sponsored by VMWare. A comparison of the number of contributors - around 260 for Harbor versus 60 for Quay - lends credence to this assessment.

We would like to note that, since SIF images can be pushed to OCI registries, there is no technical requirement to deploy a Library API registry when choosing Singularity. This has been demonstrated by the Singularity HPC Library, where SIF container images are hosted on either DockerHub or the Github Container Registry.

6 KUBERNETES INTEGRATION SCENARIOS

In previous sections, we evaluated the integration of container engines with WLMs such as `Slurm`. These WLMs schedule jobs and execute batches of single execution jobs on an HPC cluster based on predefined and flexible rules.

In a cloud system, a similar function may be performed by a cloud orchestrator - an application that can configure, manage, and coordinate multiple containers. The best known of these is `Kubernetes` [25], an open source platform first developed by Google, but now maintained by the Cloud Native Computing Foundation (CNCF). Various distributions of `Kubernetes` exist, including `K3s` [22] (`lightweight Kubernetes`), a fully conformant, pared down version packaged in a single binary and designed for use on e.g. Edge, IoT, Embedded, etc. devices.

Registry	Image Squashing	Image Formats	Multi-Tenancy	Quota	Signing	Deployment	Build Integration
Quay	on-demand	OCI	yes ("Organization")	per-project	yes	Kubernetes Operator	build on Kubernetes, EC2
Harbor	no	OCI	yes ("Project")	per-project	yes	Docker Compose, Helm Chart	via CI/CD
GitLab	no	OCI	yes ("Organization")	minimal solution self-hosted	no	Linux packages, Helm Chart, Kubernetes Operator, Docker, GET	via CI/CD
Gitea	no	OCI	no	no	no	Docker Compose, Binary, Helm Chart	via CI/CD
shpc	-	SIF	no	no	yes	Docker Compose	build on GCC
Hinkskalle	-	SIF, OCI	no	no	yes	Docker Compose	no
zot	no	OCI	no	no	yes	Docker, Helm, Podman	via CI/CD

Table 5: Continuation of table 4, listing supported image formats, deployment techniques, and build integration.

The interest in `Kubernetes` integration scenarios for HPC systems stems from the fact that domains such as bioinformatics and the data sciences have made successful use of traditional cloud resources like the Google Cloud Platform (GCP) for their research pipelines. Subsequently, workflows and workflow systems have been developed which rely on `Kubernetes` as an interface to deploy and run the respective containers and processes. Supercomputing facilities, meanwhile, optimized the use of WLMs on their systems in the past decades. They must thus be able to integrate their WLMs with orchestrators when running such `Kubernetes` workloads. This is particularly crucial in regards to the accounting of used resources, but also for optimized scheduling and the integration of newer features like preemption.

As noted in 3.2, HPC container software may provide only a subset of the isolation techniques cloud-native solutions employ, and some containers may therefore require modification before use. Integrating `Kubernetes`, which supports namespace isolation and makes user mapping possible, could remedy this issue, as it would make it possible to run containers, as well as workflows spanning multiple containers that require these techniques, without alterations.

While container orchestrators are more common in cloud environments, they can and have been used on HPC systems for container deployment, at times in conjunction with classic HPC workload managers. Here we review the possible integration scenarios as found in the literature [2, 27–29, 45], and finally propose a new integration method.

6.1 On-Demand Reallocation of Compute Nodes

We consider an integration scenario where `Kubernetes` and the WLM live adjacent to each other, and one or the other may be in control of node allocation. While Wickberg [45] discusses an explicit `slurm-k8s-bridge` in which `Slurm` is prioritized and schedules both `Slurm` and `Kubernetes` workloads, we are taking a more abstract approach here.

We propose a setup that consists of a minimal, dedicated Kubernetes cluster on separate hardware, or in a suitable virtualized environment with access to the appropriate network segments. As users request nodes to run workloads on Kubernetes, the WLM is automatically instructed to take a corresponding number of nodes offline, which are then reconfigured to run a Kubernetes agent or Kubelet, and connect to the Kubernetes cluster. Kubernetes can then deploy pods on these ephemeral nodes. Idling nodes must be returned automatically to the WLM for management. Being an orchestrator for large scale infrastructure, Kubernetes should be well suited to this kind of dynamic reconfiguration. While the WLM should also be prepared to drain nodes dynamically, this will likely affect the queuing, since nodes requested by a job may not be available anymore.

6.2 WLM in Kubernetes

Automating the different services of a WLM such as `Slurm` on a `Kubernetes` cluster, or any cloud orchestration service, represents the simplest of the integration scenarios described in this paper, and cloud native solutions have been proposed [27]. If the containers receive privileged access to the underlying high-speed network and accelerators, the WLM can be used to schedule HPC jobs as in a classical HPC system setup. This approach does not enable running containerized workloads within the WLM. The WLM merely acts as a classic job scheduler, avoiding the need to rewrite workflows to directly use e.g. `mpirun` and the WLM facilities to provision resources.

For compute centers, this setup may provide a flexible multi-tenancy architecture, but its implementation has some caveats. Recent WLM versions can run within unprivileged containers, but setting up resources like accelerators or HPC network devices requires extended privileges, possibly even to the fabric. Great care must thus be applied when scheduling the Pods of different tenants on the same compute nodes, or when allowing customers to obtain elevated privileges within the Pods or the Kubernetes instance itself. This applies when Kubernetes is used by the operator to provide segmentation of a cluster, and even more so if customers are permitted to directly deploy Pods on the Kubernetes cluster expecting a self-deployed WLM to obtain privileged access. Virtualization may alleviate some of the security concerns if properly integrated with Kubernetes and the fabric, but any possible performance penalties incurred by the additional layer introduced must be verified.

6.3 Kubernetes in WLM

Initial Kubernetes cluster in WLM allocation setups were evaluated on VEGA with the `CloudHypervisor` [18] serving as a Virtual Machine Monitor (VMM). With the advent of rootless `Kubernetes` setups, it is possible to run `Kubernetes` as a non-root user with the same technology used for running containers. This can be exploited to not only run Kubelets, but to also launch minimal Kubernetes `K3s` and Minikube clusters within a WLM allocation.

In this WLM allocation, the first node runs a minimal Kubernetes instance, with the other nodes running the Kubelets connecting back to this first node. The network is fully managed by the

WLM, and should not require additional configuration. While this approach permits perfect isolation between Kubernetes clusters started by different users, it can introduce considerable startup overhead. Until the Kubernetes cluster is ready, scheduling Pods or running workflows is not possible. Additional difficulties include the integration of this setup into existing user workflows, as well as the question of how users specify workloads to run.

6.4 Bridged Kubernetes and WLM

We identified two modalities on how to "bridge" `Kubernetes` and HPC WLMs. The first one is via `Kubernetes` *Operators*[28], allowing `Kubernetes` to schedule external resources. By providing such a bridge operator, users of `Kubernetes` can use the same resource description model as for the rest of the their workloads to explicitly schedule processes on a WLM like LFS, `Slurm`, or Torque. The `Kubernetes` *Batch* Working Group is working on extending the current description framework to close the gap to the detailed resource model provided by HPC WLMs. While this can be integrated with common workflow managers like `Kubeflow` [24], the drawback of this approach is the required explicit formulation in the resource description.

A more elegant approach, named KNoC, and recently published by Maliaroudakis et al. [29], is the implementation of a virtual Kubernetes agent or Kubelet. In contrast to the first, external resource management modality outlined above, a separate service acts as a regular Kubelet. It schedules Pods as jobs by starting containers using e.g. `Apptainer` [38] within WLM allocations, then tracks their execution and reports back. This execution happens in an almost transparent way to the user of the Kubernetes cluster and to the operators of the HPC cluster, who only have to provide the capability to run containers within their WLM.

6.5 Kubernetes Agents in WLM Allocation

Similar to the previous scenario, this relies on a dedicated Kubernetes cluster. Rather than reallocate complete compute nodes, Kubernetes agents (Kubelets) are started as part of a WLM allocation (e.g. one Kubelet on each node). These agents then have to be able to connect back to the Kubernetes cluster to receive instructions on which containers to run. This scenario relies on the rootless-approach to run Kubernetes, and requires a compatible configuration between the WLM and Kubernetes. This includes enabling version 2 of the Linux cgroups framework, cgroup delegations, and setting a suitable network configuration.

As with the KNoC approach described above, the advantage of this approach is that all the accounting information is available within the WLM. The main difference to KNoC is that the vanilla `K3s` distribution can be used on the Kubelet side, with any extensions required for scheduling being added to the main Kubernetes cluster. KNoC's design relies on the `VirtualKubelet` [44] project and an underlying implementation for scheduling the Pod via the WLM. The Pod environment produced by this implementation may undesirably deviate from the standard of Kubernetes.

Our Kubernetes agent in WLM allocation solution caters to the needs formulated at the beginning of section 6:

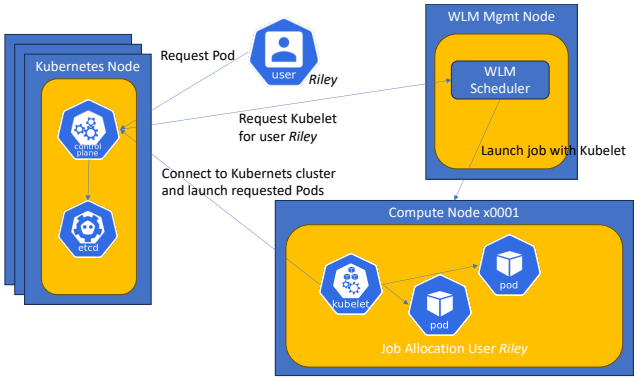

Figure 1: Principle of running Kubernetes Kubelets dynamically within a WLM job allocation. [7]

- We want a continuously run Kubernetes cluster to schedule workflows without requiring the user to start a full Kubernetes first via the WLM.
- Workloads or Kubernetes Pods should be scheduled on compute nodes *within* a Slurm allocation, so as to use Slurms accounting and compute resources.
- Pods should run transparently on compute nodes (no changes to existing workflows).

A proof-of-concept of this approach has been implemented to show the feasibility of building a Kubernetes cluster across the high-speed network of a compute cluster using Slingshot, and is shown in figure 1. A user requests a Pod from the Kubernetes cluster, which generates a request for the WLM to start a Kubelet on a compute node. The Kubelet connects to the Kubernetes cluster, which then has the resources to schedule the requested Pod.

6.6 Summary

Static partitioning leads to reduced utilisation and/or a load imbalance, while dynamic partitioning, including dynamically draining or undraining nodes from `Slurm` or `Kubernetes`, is cumbersome, slow and introduces disturbances to the system which may be difficult to monitor. Accounting of CPU time has to be monitored and consolidated separately. Running the WLM in `Kubernetes` does not provide accounting for `Kubernetes` jobs via the WLM, and potentially introduces performance bottlenecks. Forwarding hardware resources, in particular vendor interfaces, in a secure way is non-trivial in this setting. Conversely, running all of `Kubernetes` within a WLM allocation leads to long startup times and requires allocations to be able to submit jobs to Kubernetes. Finally, using a Kubernetes-WLM Operator requires a change in workflow scripts.

The only solutions satisfying the requirements are therefore the ones mentioned in section 6.5 and the second part of 6.4. Yet for both approaches, secure multi-tenancy and transparent scheduling of multi-node Pods remain challenging, with the latter likely requiring support of the workflow manager used for scheduling.

7 CONCLUSION AND OUTLOOK

We have highlighted the specific needs of high-performance compute systems and sites, and provided an in-depth analysis of underlying operating system mechanisms. We categorized the most prominent cloud and, especially, HPC container solutions according to the evaluated system mechanisms, providing a decision document for supercomputer operation centers. A special focus was given to security-relevant decision criteria.

From the given analysis, we infer that the `Apptainer` and Singularity family of container solutions has a unique feature set, permitting their application to a wide range of deployment scenarios, if one is willing to compromise on security. Beyond this, the HPC-specific solutions closely match what cloud-focused solutions like `Podman` provide, except for flattened filesystem images, namely unpriviled `UserNS` without setuid, and using FUSE-based drivers. With registries like `Quay` [39]or `Dragonfly` [12] providing `eStargz` or `EroFS` images, which can be either generated on-the-fly or uploaded in addition to the OCI compatible layers, we assume it will not be long until these formats are evaluated, and possibly adopted, for HPC usage as an alternative to SIF. And while Singularity has pioneered the support of encryption and signing, registry-supported solutions for both are being introduced in the cloud compute ecosystem via the `Notary` [37], `sigstore`, and `ocicrypt` [10] projects. Given that cloud computing is the driving force behind container development, we expect HPC to follow suite.

In section 6, we discussed why integration with Kubernetes may become relevant for supercomputing centers, and reviewed several solutions, finally proposing a new one. Two approaches were identified to address HPC needs, with their differences lying in the granularity of the scheduling, and their capability to provide a standard Kubernetes execution environment. For both approaches, multi-tenancy and security aspects must still be worked out. We did not, however, discuss the practicalities of data locality and movement. For container solutions, we highlighted the aspect of UID/GID mapping back to the original filesystem, indicating a manually managed and likely non-federated storage such as a cluster filesystem. The biggest advantage of API-based job scheduling via Kubernetes would be its neutrality towards the execution engine and placement, making it possible to have workflow engines place the execution of processes based on different criteria. This leaves the remaining challenge of data movement in HPC, as cloud environments often focus on object storage, while HPC uses cluster filesystems.

Our analysis provides the required data points to solve both containerization and abstract scheduling of applications aspects on HPC systems. What remains beyond this in regards to adaptive containerization within HPC is the challenge of optimizing containers, selecting the most fitting optimized container, and generating optimal runtime parameters for the respective target hardware in an automated fashion, as well as scalable registry architectures for the optimized storage and retrieval of containers. We previously worked on the issue of container build optimization by using input from the user in combination with performance modeling, then mapping the optimal application parameters to a target infrastructure and building an optimized container [30]. Challenges faced while working on this project included the costliness of running benchmarks on each given hardware and for each container to create the baselines for the performance modeling; the multitude and complexity of available performance providing components such as compilers, libraries, etc.; and the storage, matching, and retrieval methods for the thus created containers.

As containers may be stored in local or public registries, overlap of tags used to label each container may complicate accurate identification and retrieval of optimal containers, thus making clear guidelines for the labeling of containers to improve metadata-aware container indexing, lookup, and selection, a necessity. Internet access restrictions make building new or reusing existing containers cumbersome, and built containers must often be shared via the filesystem, an issue which may be solved by implementing site-local registries as a proxy for third-party registries, and to directly host filesystem images such as SIF or SquashFS. Finally, the user must be guided through the selection of the best matching container and the optimal runtime parameters, a process that currently can not be accomplished fully automatically due to the multitude of applications, container sources, and metadata application styles, which is additionally complicated by the hardware heterogeneity of modern HPC systems. Security policies of sites often prohibit users from building their own containers, necessitating new guidelines regarding the roles of users and system administrators in the process of adding applications on user behest.

ACKNOWLEDGMENTS

The authors would like to thank Alfio Lazzaro, HPE HPC/AI EMEA Research Lab, for the very valuable discussions and early feedback on the draft. The authors would also like to thank the Gauss Centre for Supercomputing e.V. (GCS) (www.gauss-centre.eu) for funding this project as part of an innovation partnership aimed at a next-generation GCS supercomputer at the Leibniz Supercomputing Centre (www.lrz.de). We thank the European Commission for continued funding of research on this topic under the Horizon project OpenCUBE (GA-101092984).

REFERENCES

[1] Subil Abraham, Arnab K Paul, Redwan Ibne Seraj Khan, and Ali R Butt. 2020. On the use of containers in high performance computing environments. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE, Virtual, 284–293.

[2] Angel M. Beltre, Pankaj Saha, Madhusudhan Govindaraju, Andrew Younge, and Ryan E. Grant. 2019. Enabling HPC Workloads on Cloud Infrastructure Using Kubernetes Container Orchestration Mechanisms. In *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. IEEE, Denver, 11–20. https://doi.org/10.1109/CANOPIE-HPC49598.2019.00007

[3] Lucas Benedicic, Felipe A. Cruz, Alberto Madonna, and Kean Mariotti. 2019. Sarus: Highly Scalable Docker Containers for HPC Systems. In *High Performance Computing (Lecture Notes in Computer Science)*, Michèle Weiland, Guido Juckeland, Sadaf Alam, and Heike Jagode (Eds.). Springer International Publishing, Cham, 46–60. https://doi.org/10.1007/978-3-030-34356-9_5

[4] Lucas Benedicic, Miguel Gila, Sadaf Alam, and T Schulthess. 2016. Opportunities for container environments on Cray XC30 with GPU devices. In *Cray Users Group Conference (CUG16)*. Cray User Group, London, 1–11.

[5] Ouafa Bentaleb, Adam S. Z. Belloum, Abderrazak Sebaa, and Aouaouche El-Maouhab. 2022. Containerization Technologies: Taxonomies, Applications and Challenges. *J Supercomput* 78, 1 (Jan. 2022), 1144–1181. https://doi.org/10.1007/s11227-021-03914-1

[6] Emiliano Casalicchio and Stefano Iannucci. 2020. The state-of-the-art in container technologies: Application, orchestration and security. *Concurrency and Computation: Practice and Experience* 32, 17 (2020), e5668.

[7] Kubernetes Community. 2023. Community/Icons at Master · Kubernetes/Community. https://github.com/kubernetes/community/tree/master/icons. Icons provided by the Kubernetes Community under CC-BY-4.0 license..

[8] ReFrame HPC community. 2023. ReFrame. https://github.com/reframe-hpc/reframe
[9] Containers. 2023. Containers/Crun. https://github.com/containers/crun
[10] Containers. 2023. OCIcrypt Library. https://github.com/containers/ocicrypt
[11] NVIDIA Corporation. 2023. ENROOT. https://github.com/NVIDIA/enroot
[12] dragonflyoss. 2023. Dragonfly. https://github.com/dragonflyoss/Dragonfly2
[13] Dave Dykstra. 2022. Apptainer Without Setuid. https://doi.org/10.48550/arXiv.2208.12106 arXiv:2208.12106 [cs]
[14] Daniel Fulton. 2022. Containers for HPC: Shifter and Podman. https://www.nersc.gov/assets/Uploads/06-Containers-for-HPC-Shifter-and-Podman.pdf
[15] Yiannis Georgiou, Naweiluo Zhou, Li Zhong, Dennis Hoppe, Marcin Pospieszny, Nikela Papadopoulou, Kostis Nikas, Orestis Lagkas Nikolos, Pavlos Kranas, Sophia Karagiorgou, et al. 2020. Converging HPC, Big Data and Cloud technologies for precision agriculture data analytics on supercomputers. In *High Performance Computing: ISC High Performance 2020 International Workshops, Frankfurt, Germany, June 21–25, 2020, Revised Selected Papers 35*. Springer, Frankfurt, 368–379.
[16] Harbor. 2023. Harbor. https://github.com/goharbor/harbor
[17] Red Hat. 2023. Containers/Podman: Podman: A Tool for Managing OCI Containers and Pods. https://github.com/containers/podman
[18] Cloud Hypervisor. 2023. Cloud Hypervisor. https://github.com/cloud-hypervisor/cloud-hypervisor
[19] Sylabs Inc. 2023. SingularityCE. https://github.com/sylabs/singularity
[20] Open Container Initiative. 2023. Runc. https://github.com/opencontainers/runc
[21] Douglas M Jacobsen and Richard Shane Canon. 2015. Contain This, Unleashing Docker for HPC. In *CUG2015*. Cray User Group, Chicago, 33–49.
[22] k3s io. 2023. K3s - Lightweight Kubernetes. https://github.com/k3s-io/k3s
[23] Rafael Keller Tesser and Edson Borin. 2022. Containers in HPC: A Survey. *J. Supercomput.* 79, 5 (Oct. 2022), 5759–5827. https://doi.org/10.1007/s11227-022-04848-y
[24] Kubeflow. 2023. Kubeflow/Kubeflow. https://github.com/kubeflow/kubeflow
[25] Kubernetes. 2023. Kubernetes (K8s). https://github.com/kubernetes/kubernetes
[26] SchedMD LLC et al. 2023. Slurm. Slurm Development and Support. https://github.com/SchedMD/slurm
[27] Sergio López-Huguet, J. Damià Segrelles, Marek Kasztelnik, Marian Bubak, and Ignacio Blanquer. 2020. Seamlessly Managing HPC Workloads Through Kubernetes. In *High Performance Computing (Lecture Notes in Computer Science)*, Heike Jagode, Hartwig Anzt, Guido Juckeland, and Hatem Ltaief (Eds.). Springer International Publishing, Cham, 310–320. https://doi.org/10.1007/978-3-030-59851-8_20
[28] Boris Lublinsky, Elise Jennings, and Viktória Spišaková. 2022. A Kubernetes 'Bridge' Operator between Cloud and External Resources. https://doi.org/10.48550/arXiv.2207.02531 arXiv:2207.02531 [cs]
[29] Evangelos Maliaroudakis, Antony Chazapis, Alexandros Kanterakis, Manolis Marazakis, and Angelos Bilas. 2022. Interactive, Cloud-Native Workflows on HPC Using KNoC. In *High Performance Computing. ISC High Performance 2022 International Workshops (Lecture Notes in Computer Science)*, Hartwig Anzt, Amanda Bienz, Piotr Luszczek, and Marc Baboulin (Eds.). Springer International Publishing, Cham, 221–232. https://doi.org/10.1007/978-3-031-23220-6_15
[30] Nina Mujkanovic, Karthee Sivalingam, and Alfio Lazzaro. 2020. Optimising AI Training Deployments using Graph Compilers and Containers. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Virtual, 1–8. https://doi.org/10.1109/HPEC43674.2020.9286153
[31] National Energy Research Scientific Computing Center (NERSC). 2023. Podman-HPC. https://github.com/NERSC/podman-hpc
[32] VBCF NGS. 2023. Hinkskalle. https://github.com/csf-ngs/hinkskalle
[33] Daniel Nüst, Vanessa Sochat, Ben Marwick, Stephen J Eglen, Tim Head, Tony Hirst, and Benjamin D Evans. 2020. Ten simple rules for writing Dockerfiles for reproducible data science. , e1008316 pages.
[34] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. 2019. Cloud Container Technologies: A State-of-the-Art Review. *IEEE Transactions on Cloud Computing* 7, 3 (2019), 677–692. https://doi.org/10.1109/TCC.2017.2702586
[35] Reid Priedhorsky, R. Shane Canon, Timothy Randles, and Andrew J. Younge. 2021. Minimizing Privilege for Building HPC Containers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3458817.3476187
[36] Reid Priedhorsky and Tim Randles. 2017. Charliecloud: Unprivileged Containers for User-Defined Software Stacks in HPC. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3126908.3126925
[37] Notary Project. 2023. Notary Project Specifications. https://github.com/notaryproject/specifications
[38] The Apptainer Container Project. 2023. Apptainer. https://github.com/apptainer/apptainer
[39] QUAY. 2023. Project Quay. https://github.com/quay/quay
[40] Sigstore. 2023. Cosign. https://github.com/sigstore/cosign
[41] Sigstore. 2023. Sigstore. https://www.sigstore.dev/
[42] Vanessa Sochat and Alec Scott. 2021. Collaborative Container Modules with Singularity Registry HPC. *JOSS* 6, 63 (July 2021), 3311. https://doi.org/10.21105/joss.03311
[43] Harmen Stoppels, Simon Pintarelli, and Ben Cumming. 2023. Squashfs-Mount. Swiss National Supercomputing Center (CSCS). https://github.com/eth-cscs/squashfs-mount
[44] virtual kubelet. 2023. Virtual Kubelet. https://github.com/virtual-kubelet/virtual-kubelet
[45] Tim Wickberg. 2022. Slurm and/or/vs Kubernetes.
[46] Laura Wratten, Andreas Wilm, and Jonathan Göke. 2021. Reproducible, scalable, and shareable analysis pipelines with bioinformatics workflow managers. *Nature methods* 18, 10 (2021), 1161–1168.
[47] Junqi Yin, Shubhankar Gahlot, Nouamane Laanait, Ketan Maheshwari, Jack Morrison, Sajal Dash, and Mallikarjun Shankar. 2019. Strategies to Deploy and Scale Deep Learning on the Summit Supercomputer. In *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*. IEEE, Denver, 84–94. https://doi.org/10.1109/DLS49591.2019.00016
[48] Andrew Younge. 2021. *Constructing Containers for Exascale Computing*. Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).
[49] Andrew J. Younge, Kevin Pedretti, Ryan E. Grant, and Ron Brightwell. 2017. A Tale of Two Systems: Using Containers to Deploy HPC Applications on Supercomputers and Clouds. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, Hong Kong, 74–81. https://doi.org/10.1109/CloudCom.2017.40
[50] Naweiluo Zhou, Huan Zhou, and Dennis Hoppe. 2023. Containerisation for High Performance Computing Systems: Survey and Prospects. *IIEEE Trans. Software Eng.* 49, 4 (April 2023), 2722–2740. https://doi.org/10.1109/TSE.2022.3229221 arXiv:2212.08717 [cs]
[51] The zot Project. 2023. Zot. https://github.com/project-zot/zot