Containers Everywhere: Towards a Fully Containerized HPC Platform

Dan Fulton
NERSC
Lawrence Berkeley National Laboratory
Berkeley, USA
0000-0002-7562-8308

Laurie Stephey
NERSC
Lawrence Berkeley National Laboratory
Berkeley, USA
0000-0003-3868-6178

R. Shane Canon
NERSC
Lawrence Berkeley National Laboratory
Berkeley, USA
0000-0002-8440-738X

Brandon Cook
NERSC
Lawrence Berkeley National Laboratory
Berkeley, USA
0000-0002-4203-4079

Adam Lavely
NERSC
Lawrence Berkeley National Laboratory
Berkeley, USA
0000-0002-2994-9004

Abstract—**Although containers provide many operational advantages including flexibility, portability and reproducibility, a fully containerized ecosystem for HPC systems does not yet exist. To date, containers in HPC typically require both substantial user expertise and additional container and job configuration. In this paper, we argue that a fully containerized HPC platform is compelling for both HPC administrators and users, offer ideas for what this platform might look like, and identify gaps that must be addressed to move from current state of the art to this *containers everywhere* approach. Additionally, we will discuss enabling core functionality, including communicating with the Slurm scheduler, using custom user-designed images, and using tracing/debuggers inside containers. We argue that to achieve the greatest benefit for both HPC administrators and users a model is needed that will enable both novice users, who have not yet adopted container technologies, as well as expert users who have already embraced containers. The aspiration of this work is to move towards a model in which all users can reap the benefits of working in a containerized environment without being an expert in containers or without even knowing that they are inside of one.**

Index Terms—**Containers, High-Performance Computing**

I. Introduction and Background

The advent of Linux containers has resulted in a major paradigm shift in software development and deployment architectures. They have also made inroads into the High Performance Computing (HPC) community. Today, use of containers positively impacts many aspects of HPC systems, on both the administrator and user side. For example, at NERSC, containers improve operational efficiency and are used to deploy critical services on the Perlmutter supercomputer [1], including the Slurm batch scheduler, system and network monitoring, and an image gateway for user container images. For users, container runtimes are available and used at many research computing facilities, including Singularity [2] at OLCF, Singularity, Charliecloud, and Podman at Sandia [3], Charliecloud at LANL [4], Sarus at ETH Zurich [5], and Shifter [6] and Podman-HPC [7] at NERSC. It is well-demonstrated that containerized HPC workloads achieve good performance with virtually no overhead [3], [8]. For certain types of workloads, containers can even outperform bare-metal jobs by eliminating metadata contention when using interpreted languages with libraries on shared filesystems [6]. Users also take advantage of containers for increased productivity by bundling and sharing the runtime dependencies of a shared HPC workload with collaborators and by insulating their workloads from system software changes. Despite the many benefits already realized, we believe that containers can provide even more flexibility and advantages for HPC when given a more central role in system design and functionality.

For those used to working with bare-metal systems, thinking in a container-centric way can feel somewhat alien. At first glance, using containers seems to add substantial complexity for little payoff. However, containers have powerful properties including process isolation and portability. When used as the fundamental system building block, containers can create resilient and flexible systems, as has been demonstrated by the commercial cloud over the last decade. Although there has been growing interest in containers within the HPC community, we have been relatively slow to embrace a container-based system.

The idea of deeper container integration is not unique in the HPC community- other work is ongoing in this area. The HPC system integrator Hewlett Packard Enterprise (HPE) is envisioning container-based approaches, both on the system administrator side with their User Access Instances (UAIs) [9] and the user side via their EX Urika Capsules framework [10]. HPE has also begun considering how they can package and distribute their programming environment via container, including how to handle licenses for proprietary software included in their current system stack.

In this paper we will argue for a *containers everywhere* system strategy in which containers are the fundamental building block of an HPC system. We examine the benefits such a system could bring to HPC administrators, system staff, users with existing container expertise, and novice users. We take a realistic look at the requirements such a system would have and discuss the technical work needed to enable this vision. We project several milestones for implementing these ideas, starting from a minimum viable product and ending at the so-called ”kitchen sink” model which includes all possible functionality. Finally, we will discuss notable technical challenges and open questions related to achieving a *containers everywhere* vision.

II. Design Requirements

A *containers everywhere* strategy for operating an HPC system would necessarily touch many aspects of the system and have many stakeholders. In this section we develop design requirements for such a scheme. First, stakeholders are identified. Next, we outline potential motivations for each category of stakeholder. Third, we outline several example use cases for a *containers everywhere* approach. Last, we arrive at a list of several attributes and develop example milestone targets that we would expect a *containers everywhere* road map to have.

A. Stakeholders

We believe that the impacts of containers broadly fall into one of two stakeholder bins: HPC operators and HPC users. Within the HPC operator category, we distinguish between administrators with rootful privileges and center staff who provide software or services but do not have root access to the system. It is additionally useful to distinguish between HPC users who have experience with containers and containerized development and HPC users who are relative novices with containers. We are left with four primary stakeholder categories:

- *HPC Admin* - An HPC administrator
- *HPC Staff Member* - An HPC systems engineer providing services, software, or other support
- *Experienced Container User* - An HPC user who is experienced with the use of containers
- *Novice Container User* - An HPC User who is a novice with the use of containers

In the remainder of this paper, and especially in this requirements section, we consider these perspectives.

B. Motivations

Several motivations are present to adopt a *containers everywhere* strategy for operating an HPC center. In their current state of adoption, containers already bring significant advantages to the HPC space, but we argue that many benefits are not being fully realized without the full integration of container orchestration as a primary operational tool. Below we highlight several areas in which containers have unrealized potential for HPC operation.

1) Motivations for HPC Users: A *containers everywhere* approach creates advantages for users experienced with containers and for container novices alike. Users are already realizing benefits of using containers within batch jobs at many HPC centers. At NERSC in 2022, 857 unique users launched Shifter containers out of 5292 unique users who ran a batch job. This number is trending up from previous years.

a) User Control: Although using containers allows users control over the software runtime of their workloads, bringing custom packages into the development environment can continue to be challenging for users. The programming environment in HPC centers is often a collaborative effort managed by a menagerie of tools, including package managers like `conda` and `spack`, modules, and `PATH` management. Packaging the programming environment in containers would not only allow HPC administrators to decouple packaging from deployment, but also allows users more customization via user provided images. Beyond custom packages, users can also run processes inside containers as root, which may enable use of some software which is otherwise difficult to provide on a multiuser system. Users who wish to isolate themselves from system software updates can use containers to provide a stable development and runtime environment.

b) Enhanced User Collaborations: Enabling users to build custom programming and development environments provides opportunities for small teams or large collaborations to easily share software and custom tools, overcoming a major barrier of providing their teams with a uniform software environment. It also enables developers of common software packages like *cp2k* [11] to prioritize distributing official containers rather than trying to curate installation instructions for a wide array of HPC systems.

c) Containers Democratized: A more integrated *containers everywhere* approach has the ability to present the containerized environment in a way that “feels like” a traditional bare-metal HPC environment. This would provide some of the advantages of containers, such as process isolation and environment reproducibility to users who are novices in container technology, while experienced container developers could leverage substantially more flexibility. Some novice users might not even realize they are using containers.

2) Motivations for HPC Operators: From the perspective of a systems administrator or staff member at an HPC center, many of the common advantages of containers and container orchestration apply and are discussed below.

a) Decoupling System Management and User Environment Management: One chief benefit of moving towards a *containers everywhere* approach is that it allows the decoupling of system management from the user environments. Additional staff and even users can help to manage and curate the user environment by providing container images. Container versions can move at their own natural cadence and no longer need to be tethered to major system updates that may require outages. This also means critical updates to the base system OS that may be needed for security or stability reasons can move more nimbly with less risk to disrupting the user environment. This would greatly reduce the burden placed on the system staff and allow responsibilities for maintaining the user environment to be more distributed.

b) System resilience and robustness: Container orchestration provides significant advantages to system resilience and operational robustness. Using declarative orchestration for critical services has the potential to improve up-time by automatically rescheduling failed instances based on programmatic health checks. In particular applications which have been designed as a collection of microservices, individual microservice components can fail or be restarted without bringing down the entire service. Having a container orchestration layer would also provide a means of removing hardware and having services automatically rescheduled onto other cluster resources, making rolling maintenances easier to carry out.

c) Testing and deployment: Containerizing software at both the system level and in the programming environment makes it easier to do integration tests before committing to upgrade the entire HPC system to new software versions. Rather than relying on filesystem paths and per-application configurations to choose software versions for testing, a small subset of the supercomputer can test new versions in containers, and easily revert to the latest stable versions after testing is complete. Dynamically allocating resources to test versions in containers this way would reduce the need for standalone development and staging hardware, and allow more resources to be allocated to the main user system. Common integral components of an HPC system such as batch schedulers, monitoring, and global configurations can be handled with this treatment.

d) Versioning software and services: Beyond just integration testing, using containers has advantages for packaging and versioning software and scripts provided by HPC operators. Using containers and container orchestration for versioning allows packaging not just the software version but also the configuration details, runtime environment, and deployment details into static images and markdown language files for easy reproducibility and tracking. Complex software developed or customized at an HPC site, such as Jupyter, Globus, or web-portals, stand to gain the most from leveraging containers for versioning and deployment in this way.

e) Synergy with cloud infrastructure: A significant advantage of using a container orchestration framework, such as Kubernetes, underneath system software and services would be synergy with cloud container platforms. This would enable us to leverage a great deal of existing software, infrastructure, practices, and standards without needing to reinvent the wheel. Possibilities include being able to burst to the cloud, and deploying HPC-like clusters in the cloud for prototyping and testing new software and architecture.

C. Use Cases

To help provide direction and test different design concerts we have outlined several use cases that should be supported by the *containers everywhere* approach. The use cases specifically reference the stakeholders outlined in Section II-A.

Many of the use cases presented would generally be achievable with any HPC system today, but are still listed here as a container-based system must also satisfy them. Other use cases detailed here are aspirational- they are most likely possible to achieve using containers but would be not be feasible without signficant effort using existing tools. Underlying these use cases is the principle that users should be able to choose from a spectrum of options that requires little to no experience with containers, but enables users to unlock flexibility and control as they gain expertise.

1) Interactive Login: A *Novice Container User* wants to easily gain access to the HPC center by logging in to an interactive session. In this case, the user will need a behind-the-scenes mechanism which decides what environment to place them in, and what aspects of their environment are ephemeral or persistent. Typical examples of interactive login would be via SSH, a remote desktop client, or a web portal like Jupyter.

2) Routine File Editing: Once in an interactive session a *Novice Container User* wants to edit some simulation source code files. They must have access to persistent storage or shared filesystem containing their files, as well as text editing tools.

3) Code Building: After editing source code, a *Novice Container User* wants to compile their HPC application using traditional compilers and approaches. They require access to a compiler and appropriate libraries for their programming model. This could require contact with a license server for proprietary compilers.

4) Running a Compiled Application: After compiling their application, a *Novice Container User* wants to run it and get results. They may want to run interactively or launch a batch job. The *Novice Container User* will need a way to interact with a scheduler and provide the parameters of their job in a simple script or specification, as well as specifying or inheriting sensible defaults for the runtime environment.

5) Debugging and Performance Analysis: After a job runs, a *Novice Container User* sees an error in the results or undesirable performance. They would like to carry out traditional runtime inspection techniques, such as attached debugging, instrumented performance analysis, or stack tracing, to improve their code.

6) Running a Metadata-intensive Application: A *Novice Container User* has an analysis routine written in large Python package like AstroPy or PyTorch, and would like to run it in parallel at potentially very large scales in an efficient way. They will need a mechanism to ensure that the required subpackage metadata are available to all the workers in the job without filesystem contention being a significant bottleneck.

7) Sharing Data Collaboratively: A *Novice Container User* wants to share simulation results or other data with colleagues. Their containerized session must have some means to share filesystem access and user identity with their collaborators.

8) Composing A Development Environment: A *Novice Container User* wants to recompile their code using a different compiler. Progressively more advanced users should have the ability to compose environments that contain precisely the versions of tools and libraries they desire. Ideally this should not require learning advanced container development methods.

9) Sharing A Composed Development Environment: An *Experienced Container User* who has composed their own

TABLE I
EXAMPLE MILESTONE TARGETS FOR *containers everywhere*

Minimum Viable	**Simple, Loveable, Complete**	**Kitchen Sink**
Minimal impact to *Novice Container User*	Users may customize login env containers	User specified resource access control
Secure: all user containers in user namespaces	Containers-in-containers	Kubernetes-in-Kubernetes
Default login and development env in container	User env/system software containers decoupled	Fully containerized system software
Compute jobs in containers	Service Plane (Staff Accessible)	Service Plane (User Accessible)
Session management		

tailored development environment wants to formalize using the same set of tools across a research collaboration. They require the ability to version and save the set of tools they are using, and to distribute it in a way that is accessible to colleagues, and would like to do so in a container.

10) Persistent Complex Services: An *Experienced Container User* wants to run a custom, complex, multi-process service to help with a data processing pipeline, or automate HPC work on the receipt of new data. They require a framework for easily launching this service, monitoring it once its running, and stopping or restarting it when necessary, and they would like to use Helm so they can easily redeploy the service in other places, including the public cloud.

11) Custom-building a HPC Runtime Environment: An *Experienced Container User* has very specific performance requirements and wants to install specific versions of HPC libraries, such as MPI and libfabric. They need access to a performant filesystem and a way to distribute these versions to the workers in their batch job. They would like to install their specific library versions in a container, so they can use it reliably when needed and revert back to the default HPC center environment otherwise.

12) Advanced Resource and Access Control: An *Experienced Container User* wants to build complex access control tooling into their collaborative environment. The user collaboration has irreplaceable data, and only specific jobs should have write access to this filesystem area while other arbitrary jobs may have read only access. The user would like to restrict write access to only automatically triggered jobs using pre-tested routines to avoid any accidental loss of data. Similarly, the *Experienced Container User* may want to use network-based controls to ensure that any services or APIs are protected from external access or can only be accessed by properly authorized clients.

13) High-Availability During Cluster Maintenance: An *HPC Admin* wants to restart or update a core system service with a rolling deploy, with the goal of having improved availability on the system by avoiding a system-wide maintenance. This requires a way to control and orchestrate deployment of the service, such that it can pushed to nodes in the system as they become empty or available.

14) Sandboxing System Update Tests: An *HPC Admin* wants to test updated system software on the production system without using a dedicated staging system or a dedicated maintenance window for the production system. Their goal is to avoid downtime and to provide as much hardware as possible to the production system when tests are not being carried out.

15) Sandboxing Non-Administrative Staff: A busy *HPC Admin* wants to sandbox a section of the production system for an *HPC Staff Member* to safely experiment with untested system software. The time of the *HPC Admin* is often in demand, and they would like to share responsibilities with a much larger pool of staff members without risking the integrity of the production system by granting full administrative privileges.

16) Update a Staff Provided Service or Software Package: An *HPC Staff Member* wants to provide a new library or service level software in the user development environment, without any possible disruptions to existing user sessions, and without deprecating prior versions. Examples of such software or services might include a new Python base image, a new database client base image, or a new compiler base image. Ideally, this build and version release process could be done at any time, completely separate from its deployment and from system maintenance.

17) Troubleshoot a User Session: An *HPC Staff Member* has received a ticket from a *Novice Container User* and wants to help troubleshoot the user's problem. The *HPC Staff Member* requires access to the user's environment and current session to inspect what is going on.

D. Design Milestone Targets

The use cases outlined in subsection II-C motivate specific design requirements for a *containers everywhere* HPC strategy. We group these design features by priority into three sequential milestone targets: (1) the Minimum Viable Product (MVP), which is the minimum implementation that would technically function, (2) the Simple, Loveable, Complete (SLC) product, which is the minimum implementation beyond the MVP, which users would actually use and adopt, and (3) the "Kitchen Sink" (KS) target, which goes beyond the SLC to include nice-to-have features and other aspirations. Design requirements are shown, sorted by milestone target, in Table I.

Despite anticipated growing pains with the operational refactor represented by a *containers everywhere* approach, achieving functional parity is a basic requirement of what we propose. We recognize the need to avoid disrupting the current functioning workflows of HPC users, and so minimizing negative functional impact is a key constraint on our design. Similarly, any changes in the HPC control plane must provide operational feature parity. Another core concern is that the *containers everywhere* approach cannot introduce any significant new security risks. To achieve this, we anticipate all user containers will be operated in user namespaces, which disallow administrative kernel capabilities and grant equivalent filesystem and execution permissions to what a user would have in the bare-metal multiuser system.

When considering use cases from the four stakeholder roles, we are assuming different functionality depending on context in the HPC system. It is useful to consider four spaces:

1) HPC System Administrative Control Plane
2) User-facing Login and Development Environment
3) Compute Allocations
4) Non-privileged Service/Workflow Plane

Some additional design targets emerge when considering these spaces. For example, the decision point to use containers in the Administrative Control Plane is separate from the other spaces, since the control plane is already isolated in most HPC deployments. The idea of a non-privileged service/workflow plane naturally emerges from Kubernetes, but is optional, and not something typically integrated with a production HPC system today. A containerized login environment means that the user may not be able to infer the host they are on, and strongly implies the need for user sessions and session management. The necessity to manipulate compute jobs or service deployments from the login environment implies a containers-in-containers scheme if all spaces are containerized. Having conceptual separation of these spaces is important for identifying appropriate progressive steps in an implementation road map.

In the next section, we discuss possible implementation schemes to approach these design target milestones.

III. Implementation

In the previous section we laid out some of the attributes and use cases a *containers everywhere* should support. In this section, we describe some of the components, tools, and software artifacts that are needed to support this vision.

A. Curated User Development Environment

A fully-containerized user experience requires that there are container images that encapsulate the user environment, providing the tools and libraries for the user to interact with the system. We expect that some of these images would mimic existing non-containerized user environments so that early adopters could easily transition to the new model. NERSC is already exploring managing user environments with containers. There is also prior work and tools that could play a role, for example, the E4S SDK provides one sample environment [12]. HPE has also developed prototype containerized environments (Capsules) but has not yet made these a full product [10].

B. Composable Images

In addition to curated environments, we envision tools that allow users to easily compose images. Existing software package managers such as Conda/Mamba, Spack and Easybuild could play a role. Ideally, users should be able to easily express which tools, software and versions they require and the compose tools should help to instantiate an appropriate environment. Mechanisms could be through manifests, GUIs or wizards.

This is also a pragmatic concern, as very large container images are more difficult to move, store, and load, so it is undesirable to have a large monolithic image with several software versions. In additional to image composition, in some cases packages could be dynamically loaded into the container at run time. Allowing composability at run time would also allow injecting proprietary vendor software as binaries, without concerns of users redistributing it in images. There are a number of ways run time injection could be achieved, via OCI hooks, through container volumes, filesystem mounts, or even importing software stored in other containers.

C. Image and Session Management

A key component of this architecture is a service or set of services that assist in managing the list of available images and defining and managing sessions. The definition of a session would include aspects such as the image, accessible volumes, network access, environment variable settings, injection of libraries for accelerators or interconnects, etc. These could be defined through a variety of methods including simple YAML files, web-based GUIs integrated into a user management system (e.g. NERSC's Iris system), and command-line based tools. HPC staff would provide default templates that cover the most common scenarios and patterns. A session manager would also assist in instantiating sessions and assist proxy services in routing connections to the appropriate session. See Figure 1 for how the session manager fits in the proposed architecture.

D. Session Proxy Service to Containers

Another key component is a session proxy, or potentially multiple instances, that manage routing connections to the appropriate back-end container. Currently, SSH is an extremely common method of accessing the system and would be the first client to consume the Session Proxy, however it could also interface with other login vehicles like remote desktop sessions, or web portals such as Jupyter.

For connecting SSH sessions, running an SSH daemon in every user environment container would be impractical and requires administrative system capabilities. Instead, a session proxy can catch incoming SSH connections and then request that a user session be started, and route the incoming user into the session via a container exec. This is similar to the model used in ContainerSSH [13] and this could serve as an initial prototype. But ContainerSSH lacks several important features. For example, it doesn't currently support reconnecting to already running containers. Therefore, this is one area where we anticipate some development is required. See Figure 1 for how the session proxy fits in the proposed architecture.

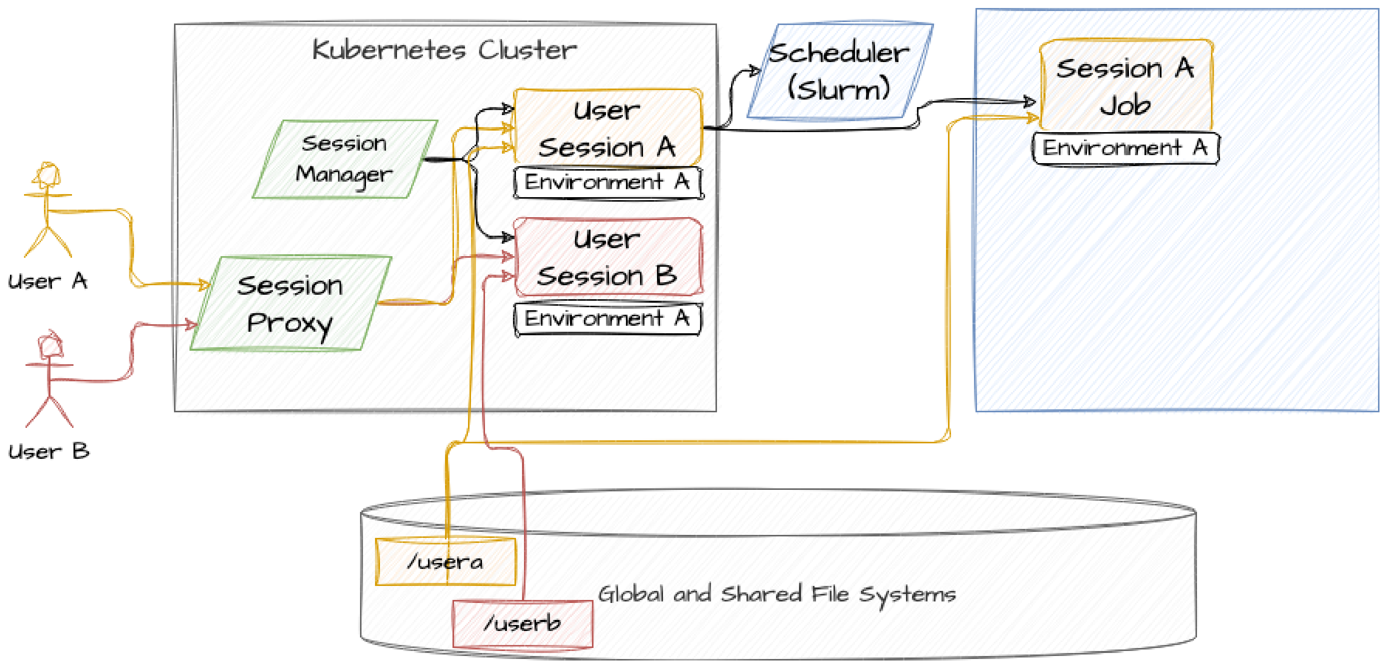

Fig. 1. Diagram of the proposed Session Proxy architecture

E. Jupyter

Recently more users have accessed NERSC via Jupyter than via SSH. We expect this trend will continue and anticipate needing to integrate Jupyter with our *containers everywhere* vision.

Today, Jupyter users land in a JupyterLab session that is run on a bare-metal login node, and can run custom Python kernels, which may or may not be containerized. Using a containerized python kernel does require some manual configuration by the user by manually write a kernelspec file which specifies their desired image. The image may be custom or off the shelf. The kernelspec will supply the container name, the kernel name, and location of the Python binary to Jupyter. In a containers-first future, it would ideal if these kernelspecs were automatically generated based on all images in a designated NERSC registry. This would require some introspection to locate the Python binary and choose a sensible kernel name.

By integrating the JupyterHub manager with the proposed Session Proxy, the outer JupyterLab service could also be run in a container. This would allow users to bring their custom user environment tools into the Jupyter context. Sample architecture showing JupyterHub integration with Session Proxy is shown in Figure 2.

The JupyterHub service is currently being run in NERSC's Rancher 2 Kubernetes cluster and is well-suited as a containerized service.

F. Batch Integration

Batch integration with containers spans multiple areas and use cases. One aspect of integration is to provide a seamless user experience when submitting and running jobs from a containerized environment. Another is how scheduling and placement of containerized services are controlled by the scheduler to ensure optimal resource utilization vs availability. We will discuss each of these.

1) Seamless User Experience: One major shortcoming today is that NERSC container users are unable to directly interface with our Slurm scheduler from inside their Shifter or Podman-HPC containers. This is because the Munge perimeter needed for secure authentication with the scheduler does not extend into the running container.

Over the past year or so, SchedMD has been developing more direct support for containers within Slurm jobs [14]. Using Slurm container support would enable processes inside the running container to issue `srun`, `sbatch`, and similar commands since Slurm's OCI `scrun` solution runs within the Munge perimeter. One shortcoming of this solution is that Slurm container support is in its infancy, and does not support many of the more advanced requirements of HPC users- e.g. running multiple processes per container in an MPI application, or being able to use multiple containers per job.

An alternative to relying on container support within the scheduler itself is working to enable scheduler support within container runtimes like Podman-HPC. One current way this can be accomplished at NERSC is with a workaround that sets up a sidecar container that is meant to proxy the requests to and from the scheduler [15]. This could be enabled via a Podman-HPC module, but since in a *containers everywhere* future users will land in a container when they log in to the system, one could argue that this capability to interact with the scheduler should be always on by default.

Further batch system integration would be required to make the experience feel seamless, and to allow users to select a development login environment. The batch executor would

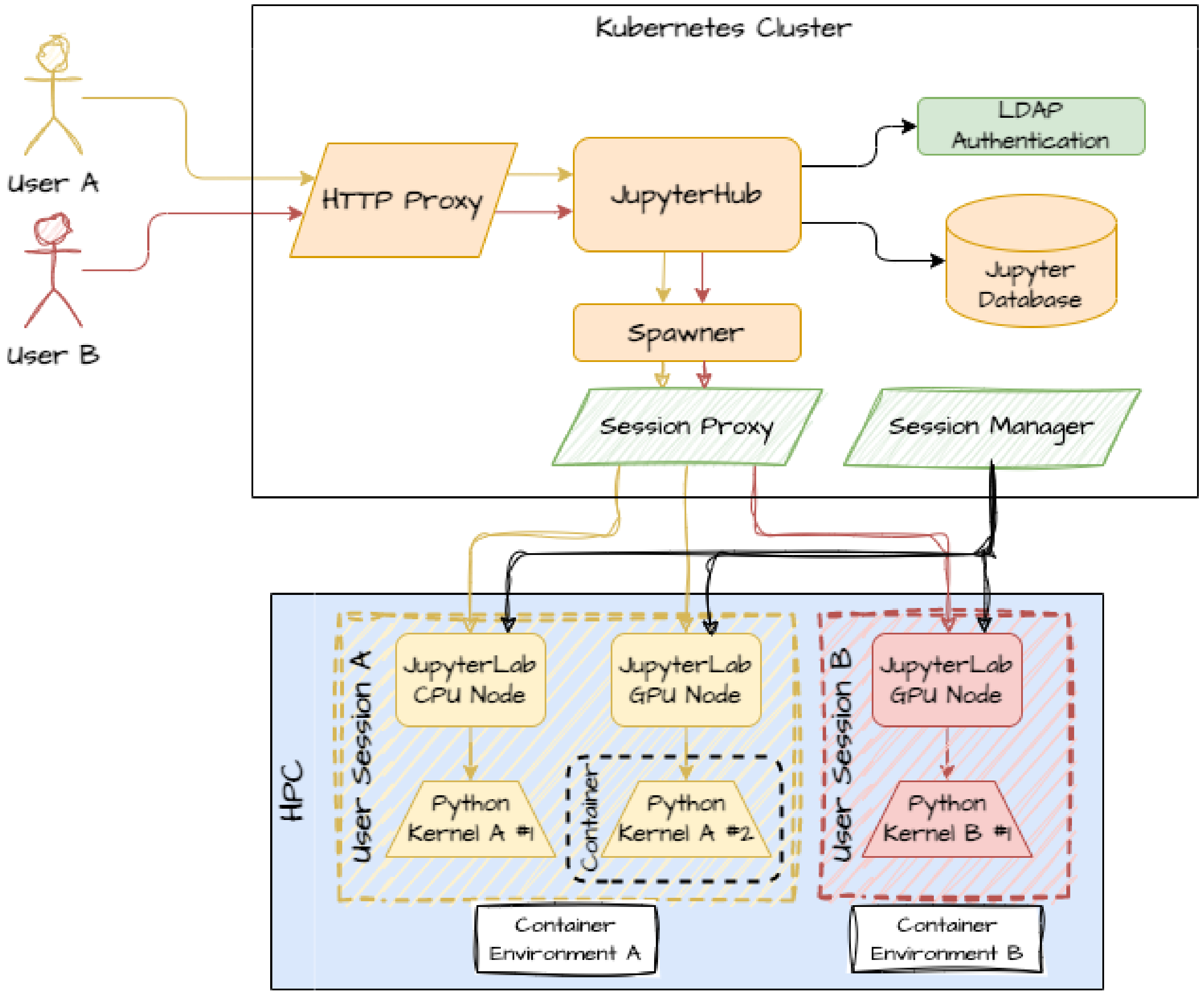

Fig. 2. Diagram of integration of JupyterHub with the proposed Session Proxy architecture

need to track the users session container, and, if no other image is specified to use, import the current session as the default batch environment.

2) Orchestration Integration: If using container orchestration in a user facing way, there is also a broader question of how to resolve the HPC scheduling paradigm with Kubernetes [16]. Most batch systems address the problem of resource scarcity, and attempt to achieve high hardware utilization. This often results in jobs waiting in a queue. By contrast, Kubernetes anticipates adequate resources, but is optimized for deployment, uptime, and resiliency. It expects administrators to provide appropriate resources for the amount of work at hand.

G. Containers in Containers

Allowing users to specify custom container images for use as development or compute environments demands that we provide container development tools to users. Fortunately, container-in-container is a well established paradigm. Using the host kernel from inside a container usually requires exposing a socket inside the container, as well as some configuration in the outer container.

```
podman run --cap-add=sys_admin,mknod \
    --device=/dev/fuse \
    --security-opt label=disable \
    -it --rm quay.io/podman/stable /bin/bash
[root /]# podman pull ubuntu
[root /]# podman run ubuntu:latest echo 'hello!'
hello!
```

This and any other necessary changes can easily be accomplished with a container runtime hook that can be triggered by the session manager any time it launches an interactive environment for a user.

H. Kubernetes in Kubernetes

More advanced scenarios could require complex hierarchical orchestration. One example would be allowing users to run services in a rootless environment like Usernetes, which has resources allocated to it by a staff or administrator level Kubernetes cluster. Similarly, an administrative Kubernetes cluster could control dynamic allocation of resources to production and staging instances, while a staff cluster could deploy software versions within an instance. Such orchestrator-in-orchestrator tricks might also be useful to resolve scheduling issues mentioned in III-F.

I. `ptrace` inside a container

Profiling, tracing, and debugging applications can be challenging in containers since the container is running in a different namespace and with some default security restrictions. Depending on the container runtime, it may need to be granted additional capabilities to enable these kinds of operations. Let's take `strace` as an example, which like `gdb`, relies on `ptrace`. There are two main cases here- (1) using `strace` inside of a container and (2) using `strace` outside of the container to get information about an application running inside of the container.

The first case is more straightforward than the second, although it does require that `strace` is installed in the image. This suggests that any NERSC base image will need to come preinstalled with basic profiling, debugging, and tracing tools like `strace`. Today with the Shifter container runtime and also the Podman-HPC container runtime, using `strace` to trace an application running inside a container works without any additional configuration required. Users can additionally use `podman-hpc exec` to exec into a running container, and also issue an `strace` command. There don't appear to be any major technical hurdles in this case. Below is an example of `strace` in Shifter and Podman-HPC.

```
shifter strace \
     -e trace=file -f \
    -o /tmp/shifter-myapp-trace \
    python3 -m myapp $(date +%s)
podman-hpc run --rm \
    --volume /tmp:/tmp \
    docker.io/myapp:v1.0 \
    strace -e trace=file -f \
    -o /tmp/podman-myapp-trace \
    python3 -m myapp $(date +%s)
```

The second case will apply in the "container-in-a-container" scenario and is more challenging. In this case, `strace` is used to profile an application running inside another container. For Podman-HPC, the `strace` container needs to be started with the `SYS_PTRACE` and `SYS_ADMIN` capabilities and also with `SECCOMP` turned off to enable low-level tracing capabilities in the container. Further, we must start our `strace` container using the same pid namespace and network namespace as our application container.

Starting the application container:

```
podman-hpc run --rm --name=test3 \
    docker.io/myapp:v1.0 \
    watch -n1 -x echo "hello"
```

Running the `strace` container:

```
podman-hpc run --rm --volume /tmp:/tmp \
  --pid=container:test3 \
  --net=container:test3 \
  --cap-add sys_admin --cap-add sys_ptrace \
  --security-opt=seccomp:unconfined \
  docker.io/mystrace:v1.0 \
  strace -e trace=file -f \
  -o /tmp/podman-test3 -p 1
strace: Process 1 attached
```

Users will either have to understand how to attach to a running application container, which requires some expert-level container knowledge (i.e. namespaces), or we will need to provide some resources like a wrapper script that automatically detects a running container to help make this process more accessible to novice users. If feasible such a wrapper could also inject debugging and profiling tools, too.

J. Accessing a license server in a container

A *containers everywhere* future will need to provide a more straightforward way to support products that require a license server running in the container. One common example of this is the use of Intel compilers in a container. This currently requires working from a pre-created NERSC base image and creating an SSH connection to the NERSC license server during the build (when running outside of NERSC). Now that Podman-HPC is available on NERSC systems configuring this license server tunnel isn't required, but it still requires the user to set up the configuration. An alternative is a multi-stage build, although novice users may struggle with this relatively advanced concept.

To help address this, one solution could be to provide a special `--intel` module for Podman-HPC that would allow users to automatically connect to the license server both during the build and while the container is running. A larger and more complex solution might be to provide a container building web-portal where the complexities of connecting to the license server are hidden from the user. This could potentially be integrated into CI pipelines.

In addition to Intel compilers, other services that NERSC that require license servers are Matlab, IDL, and VASP. A workable solution to enable license managers to communicate with containers at NERSC is a strict requirement for these applications.

IV. Discussion and Future Steps

In the previous sections we have outlined our vision for a *containers everywhere* HPC environment. Here we will attempt to synthesize these ideas into a roadmap. Fortunately this vision doesn't require an all-or-nothing implementation; in many of the use cases we have outlined, progress can be made incrementally. Unfortunately the progression will favor the *Experienced Container User* over the *Novice Container User* at the beginning of this effort, but supporting these users is still a central tenet of this vision and we will continue to work with this goal in mind.

In Table II, we outline the use cases we discussed in Section III, with the exception of "Containers-in-containers", since there are no technical hurdles in this area. We split these implementation goals according to the example milestones we discussed in Section II with the goal of clarifying the road map.

There are several notable gaps that are present between what exists today and what is required in our vision. We will discuss the most major gaps below.

A. *Control Plane Devices*

Although a lot of system software can benefit from having orchestrated deployment, Kubernetes can make it difficult for software to track and have affinity to specific devices. This could be problematic for using orchestration with HPC high speed networks. Other unseen problems could emerge related to Kubernetes and device management. Kubernetes has the ability to support additional vendor devices via a Device Plugin system [17], but this is largely unexplored from the HPC perspective at NERSC and demands more attention.

B. *Sharing Across User Namespaces*

User namespaces are a key component of secure user-permission containers running on a multi-user system. But while isolating users from each other is beneficial in many situations, it also means users no longer see other users and filesystem groups, creating a barrier to sharing files on a common filesystem. A workaround or alternative means of user data sharing will need to be developed to overcome this.

C. *Session Management and Proxying*

Services to support session management and routing are one of the key gaps. While there are some existing implementations that could be leveraged or extended, these lack the full set of required features and capabilities.

D. *Slurm and Kubernetes integration*

The issue of how traditional HPC schedulers like Slurm will integrate with more cloud-like schedulers like Kubernetes is still very much an open question. Even SchedMD, the parent company of Slurm, recently gave a talk where they are exploring several possibilities [16]. This is a question that will likely not be solved by NERSC alone but by the larger community of both vendors and HPC stakeholders.

E. *Composable images*

Another major hurdle to container adoption among users is the need to learn a new way of building and and installing software environments (i.e. containers). This barrier may be especially high for novice users who are not familiar with configuring low-level libraries. Providing a web-GUI or yaml-based composable framework to assist these users in constructing images would likely be extremely helpful for this class of *Novice Container User*. Developing the infrastructure to support this vision though will require technical investment. Similar tools like NVIDIA's HPC container-maker exist [18], although are very vendor-specific and don't provide any web interface. Coupling a framework with automatic container build and push service, perhaps to some official registry, will also require some development.

V. Conclusion

The vision expressed in this paper is admittedly ambitious and a great deal of technical work will be required to realize the vision for the *containers everywhere* blueprint laid out.

It is our hope that by identifying the different types of stakeholders, identifying the core functionality that must be supported for each of these stakeholders in a *containers everywhere* paradigm, and beginning to explore some of the technical hurdles that must be addressed to make these ideas possible, this set of implementationm suggestions and thought experiments will help concretize the work that will need to be done in the future. We have sketched out a blueprint to realize this vision in Table II.

Those skeptical of containers may feel that this paper describes an awful lot of additional work to achieve an environment that in some cases is meant to look and feel like a bare-metal environment. It is our belief that the additional innovation in pursuit of a *containers everywhere* future can help usher in a new, more cloud-like era of HPC, where we can reap the benefits of more robust systems and user applications, more easily and broadly-managed HPC systems, more portable and shareable user environments, and a more productive HPC experience for both staff and users alike. Users and administrators of the commercial cloud have already embraced this containers-first approach- it seems it's time for the HPC community to shift our mindset to do the same.

Acknowledgment

This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. We would like to express our gratitude to Christopher Samuel for his help in installing Slurm 23.02 on a test system for initial container support evaluation. We would also like to gratefully acknowledge Adam Lavely for review and contributions to this manuscript.

TABLE II
A PROGRESSIVE ROAD MAP FOR ACHIEVING A *containers everywhere* SYSTEM PARADIGM.

Feature	**Minimum Viable**	**Simple, Loveable, Complete**	**Kitchen Sink**
Curated User Development Environment	Provide basic NERSC environment	Provide NERSC environment with profiling, debugging tools	Provide NERSC environment with advanced tools, including creating and managing services
Composable Images	Provide set of base images users can FROM	Full web-based container warehouse	Full web-based container warehouse including custom build/upload capability
Image and Session Management	Users can leave and return to a session, but with manual configuration	Special wrappers/utilities allow users to seamlessly move between containers	Web-based tools allow users to toggle between sessions
SSH Sessions to Containers	Users can SSH into a specific container, but with manual configuration	Session proxy service helps route user sessions	
Jupyter	Users can manually configure a stock or custom image	Users automatically land in their chosen image	
Batch Integration	Users can interact with scheduler with manual configuration	Seamless access to scheduler	Integration of both Slurm and Kubernetes-like scheduling
Kubernetes in Kubernetes	Possible for *Experienced Container User*	Wrapper scripts/other utilities make workflow more accessible	Can be started and managed with web-based service interface
`ptrace` inside a container	Tracing and debugging is possible but requires expert knowledge	Tracing and debugging accessible to *Novice Container User* facilitated by wrapper script or other utilities	
Accessing a license server in a container	License server works but requires expert-level user build and/or configuration	License server accessible within the container with minimal user intervention (e.g. using a Podman-hpc module)	All license servers accessible without user intervention

REFERENCES

[1] "NERSC Perlmutter Architecture." [Online]. Available: https://docs.nersc.gov/systems/perlmutter/architecture/

[2] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PloS one*, vol. 12, no. 5, p. e0177459, 2017, publisher: Public Library of Science San Francisco, CA USA.

[3] K. Pedretti, A. J. Younge, S. D. Hammond, J. H. Laros III, M. L. Curry, M. J. Aguilar, R. J. Hoekstra, and R. Brightwell, "Chronicles of Astra: Challenges and Lessons from the First Petascale Arm Supercomputer," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2020, pp. 1–14, journal Abbreviation: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis.

[4] R. Priedhorsky and T. Randles, "Charliecloud: Unprivileged Containers for User-Defined Software Stacks in HPC," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. Association for Computing Machinery, 2017, event-place: New York, NY, USA. [Online]. Available: https://doi.org/10.1145/3126908.3126925

[5] L. Benedicic, F. A. Cruz, A. Madonna, and K. Mariotti, "Sarus: Highly Scalable Docker Containers for HPC Systems," in *High Performance Computing*, M. Weiland, G. Juckeland, S. Alam, and H. Jagode, Eds. Cham: Springer International Publishing, 2019, pp. 46–60.

[6] D. M. Jacobsen and R. S. Canon, "Contain this, unleashing docker for hpc," *Proceedings of the Cray User Group*, pp. 33–49, 2015.

[7] L. Stephey, S. Canon, A. Gaur, D. Fulton, and A. J. Younge, "Scaling Podman on Perlmutter: Embracing a community-supported container ecosystem." IEEE, 2022, pp. 25–35.

[8] A. Torrez, T. Randles, and R. Priedhorsky, "HPC Container Runtimes have Minimal or No Performance Impact," in *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, Nov. 2019, pp. 37–42, journal Abbreviation: 2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC).

[9] E. Lund, "UAIs Come of Age: Hosting Multiple Custom Interactive Login Experiences Without Dedicated Hardware," 2022.

[10] H. P. Enterprise, "HPE Cray EX Urika Analytic Applications Guide (1.4) (S-8006)," Tech. Rep. [Online]. Available: https://support.hpe.com/hpesc/public/docDisplay?docId=a00115103en_us&page=Urika_Manager.html

[11] "cp2k." [Online]. Available: https://github.com/cp2k/cp2k

[12] M. Heroux, J. Willenbring, S. Shende, C. Coti, W. Spear, J. Peyralans, J. Skutnik, and E. Keever, "E4S: Extreme-scale Scientific Software Stack," 2020.

[13] "ContainerSSH." [Online]. Available: https://github.com/ContainerSSH/ContainerSSH

[14] "Slurm containers." [Online]. Available: https://slurm.schedmd.com/containers.html

[15] "NERSC container-proxy." [Online]. Available: https://github.com/scanon/container_proxy

[16] "Slurm and or vs Kubernetes." [Online]. Available: https://slurm.schedmd.com/SC22/Slurm-and-or-vs-Kubernetes.pdf

[17] "Kubernetes Device Plugins." [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins

[18] "NVIDIA hpc-container-maker." [Online]. Available: https://github.com/NVIDIA/hpc-container-maker