Containers-first user environments on HPE Cray EX

Felipe A. Cruz
Swiss National Supercomputing Centre
Lugano, Switzerland
felipe.cruz@cscs.ch

Alberto Madonna
Swiss National Supercomputing Centre
Lugano, Switzerland
alberto.madonna@cscs.ch

***Abstract*—In High-Performance Computing (HPC), managing the user environment is a critical and complex task. It involves composing a mix of software that includes compilers, libraries, tools, environment settings, and their respective versions, all of which depend on each other in intricate ways. Traditional approaches to managing user environments often struggle with finding a balance between stability and flexibility, especially in large systems serving diverse user needs.**

This work introduces a containers-first approach for HPC, enhancing stability and flexibility in user environments by seamlessly integrating container technologies on an HPE Cray EX system. This approach evolves the user environment management and delivery, enabling customized, fast, and transparent deployment through containers. It aims to isolate the user environment from system software, maintaining consistent workflows even during maintenance and upgrades. Furthermore, it increases environment flexibility, adaptability to specific user needs, and accelerates deployment of updates and fixes, while minimizing system dependency.

The containers-first approach is a step forward towards improved testing capabilities, with increased portability and reproducibility of scientific workflows. This paper outlines the principles and implementation of seamless integration of container technologies into HPC infrastructure, which can directly benefit operational efficiency, user productivity, and the overall management of HPC systems.

***Index Terms*—User environments, Linux containers, user services, system operations, workload manager**

I. Introduction

In the current context of High-Performance Computing (HPC), Linux containers are improving how we deploy and run HPC workloads. The usage of containers is a response to the challenges of traditional HPC system management, which suffers from the complex integration of interdependent software components, management of software conflicts, and the fine-tuning of systems for peak performance, all of which are formidable challenges. Moreover, the often sequential and compartmentalized nature of traditional system engineering work has made matters more challenging, often resulting in slow and error-prone processes. As we strive towards more flexible and efficient delivery of HPC solutions, Linux containers appear as an innovative and practical alternative. Container technology presents itself as a controlled and managed abstraction for HPC workflows, enhancing productivity, collaboration, and ease of management. Notably, containers have been adopted as the medium of choice to deploy many AI workflows, and the increasing support by workflow managers (AiiDA [1], Nextflow [2]) will only accelerate this process.

An extensive usage of containers can bring major benefits to HPC users and providers; and could address relevant problems encountered by HPC users and providers alike:

- Decouple user environment from system components: more freedom and flexibility in planning and executing system maintenances, improved robustness and consistency of the user environment as they do not change across maintenance, more informative and meaningful analysis of regressions as user-side components do not change and are reproducible.
- Isolation: These environments allow for isolation, where specific software versions and dependencies can be installed without depending on shared global system installations. This feature provides scalable management of dependencies that are tailored to the specific needs of projects and users, ensuring each has access to the specific versions of libraries and tools it requires, and preventing conflicts between different projects.
- Reproducibility and consistency: By utilizing containerization, projects become more reproducible and consistent. Enabling other scientists to replicate the exact environment, complete with identical software and library versions. This effectively resolves the "works in my machine" issue while maintaining consistency across development, testing, and production environments.
- Efficient deployability: Scientists can readily utilize tools from the container ecosystem that simplify the software management of a project. Capabilities like Dockerfiles in conjunction with package managers can leverage base images and layer caching alongside automation (CI/CD and DevOps), greatly facilitating consistent and efficient deployments.
- Extended compatibility: Containerization freezes the specific set of all installed software, while system dependencies can be addressed via system-specific hooks. This approach simplifies complexity and allows scientists to seamlessly deploy the same environment across systems quickly.
- Flexibility for experimentation: Scientists can use containerized user environments to rapidly experiment and test different software versions without affecting their primary environment. This flexibility is crucial for quickly evaluating new libraries and ensuring compatibility, without risks.

Nevertheless, while Linux containers offer a streamlined method for distributing software and creating isolated execution environments, their use within HPC, particularly for scientific users developing high-performance applications, presents challenges. The simplicity of importing container images strongly contrasts with the complexity of configuring these images into fully functional HPC environments. Often, HPC users struggle with multiple complex command-line operations to customize many properties, including bind mounts, environment variables, container working directory, container entrypoint, and the correct activation of hooks and plugins.

This complexity primarily arises from the lack of integration between the container tools and the broader HPC infrastructure. Often viewed as accessory utilities for isolated use, traditional HPC container tools pose a significant challenge to achieving cohesive and efficient workflows within the HPC ecosystem. This challenge has a significant impact on the iterative process of developing and debugging complex scientific software stacks. Debugging workflows with containers forces developers into a time-consuming and manual effort of iteratively rebuilding, uploading, and re-importing images, a process that not only slows down progress but also introduces additional layers of complexity through the need for careful coordination between various tools and systems.

The overall effect of these challenges is substantial, as they introduce friction toward the adoption and effective use of containers for HPC, causing many HPC users to miss out on the benefits that containers bring. Overcoming these obstacles is the next step towards leveraging the full potential of container technology, ensuring container-based workflows that are feature-rich, efficient, and user-friendly for the HPC user community.

We propose an approach to user environments in HPC designed around making containers first-class elements of the system, focusing on integrating container technologies directly into HPC infrastructures. This approach, dubbed *containers-first* for short, is intended to make software management and execution more efficient while preserving the familiar user experience offered by traditional HPC systems. It directly tackles the complexities of integrating containers into HPC environments, ensuring a seamless and transparent usage of HPC systems. This strategy also enhances productivity for users and administrators by streamlining operations.

The rest of this document is structured as follows. Section II states the principles that characterize the approach of containers as first-class HPC objects and outlines a solution founded on such principles. Section III describes a pathfinder implementation for the proposed solution. Section IV discusses how the pathfinder is deployed and configured to suit an HPE Cray EX system. Section V presents a number of real-world use cases and highlights the advantages delivered by a containers-first approach for each of them. Finally, Section VI concludes the discussion by summarizing the main contributions of this work.

II. Principles of containers as first-class HPC elements

The containers-first approach introduces several principles to improve user experience and operational efficiency in HPC systems:

(P1) Ease of Use. The system is intentionally designed to be user-friendly and intuitive. We prioritize interfaces that simplify the workflow for users and developers, making the process straightforward and efficient.

(P2) Removal of Complexity. Although deploying containers into functional HPC environments can be complex, our approach ensures that this complexity is hidden from users. The system manages all the details internally, presenting users with a simplified and clean interface. This allows scientists to focus on their work without the need to concern themselves with the technical aspects of container management.

(P3) Predictable Operations. Users can expect reliable and consistent results each time they use the system. A containers-first approach aims to provide a stable user environment, maintaining consistency through infrastructure updates and maintenance without unexpected changes.

(P4) Seamless Integration. The system orchestrates the operations of the various container toolset components, thereby simplifying the user's workflows. As such, users do not need to manually manage the interactions between different container tools to ensure smooth operations.

(P5) Minimal Learning Curve. Designed with familiarity in mind, the containers-first system is easily accessible to users familiar with traditional HPC systems. It offers straightforward core functionalities and enables users to become proficient with minimal additional training quickly.

Through these principles, the containers-first approach aims to modernize HPC systems, making them more accessible, efficient, flexible, and easier to operate without sacrificing the performance and familiarity that HPC users expect.

In this work, we have put into practice the containers-first principles, implementing a container engine that seamlessly integrates with the HPC infrastructure. The starting step was to evolve the integration between the Workload Manager (WLM) and HPC container tools. We achieved this via a tightly integrated WLM scheduler plugin (as pioneered by Shifter [3] and Pyxis [4]), introducing simplicity and efficiency in workflow management and making the process intuitive for users and developers (P1).

Central to our approach is the introduction of the Environment Definition File (EDF), a comprehensive blueprint detailing all the properties required to instantiate a functional container-based HPC environment. It encapsulates the container customizations in a single, manageable unit, simplifying the tasks of handling, distributing, and building upon them. This design choice removes the underlying complexity of containers (P2), allowing scientists to concentrate on their

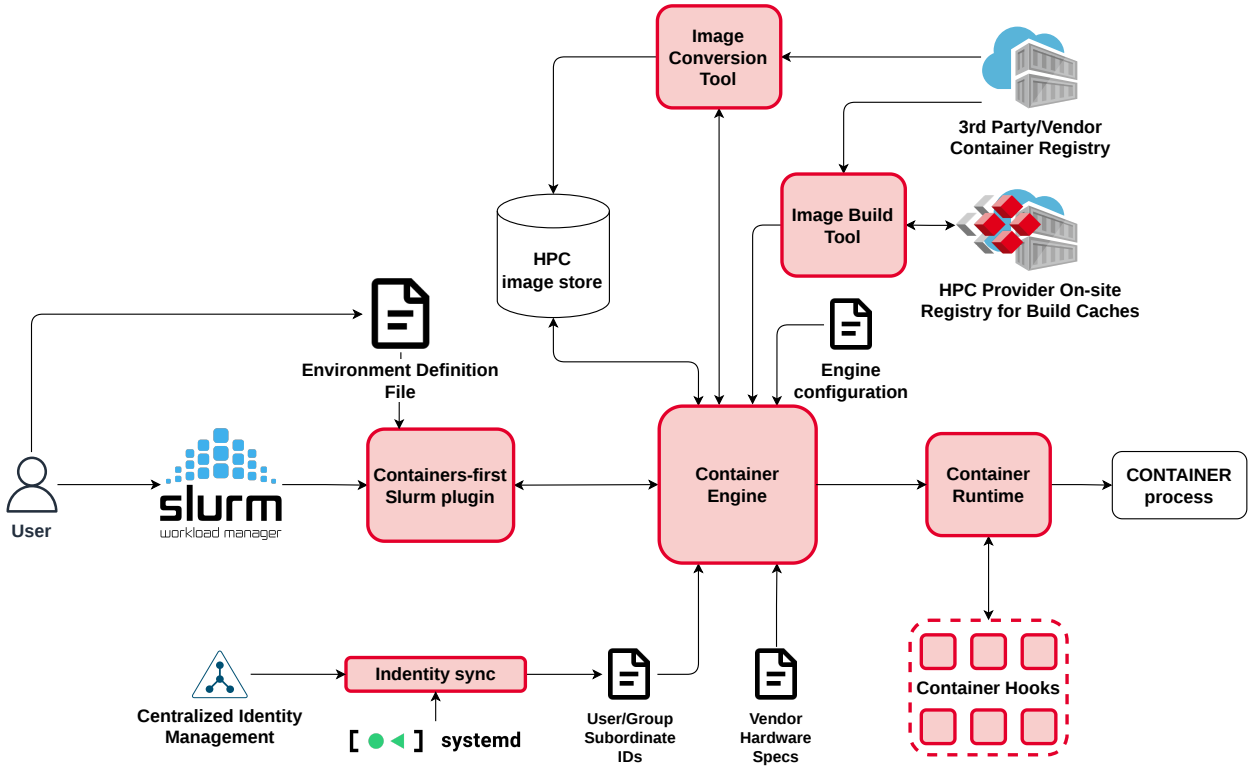

Fig. 1: Design diagram of a containers-first implementation on an HPC system with Slurm.

research without concerns about the technicalities of container management.

The integration of the EDF with the WLM further leverages container technologies to interface with its host capabilities, ensuring Predictable Operations (P3). As such, users can conveniently submit the EDF as a single option within the WLM command or batch script, guaranteeing a stable and consistent user environment. This stability is maintained throughout infrastructure updates and maintenance by employing container hooks that interface with host capabilities. In effect, hooks implement on-demand access to system features and hardware, enabling HPC performance without requiring a strict coupling between container images and host resources. Thus, changes in driver libraries or low-level middleware maintained by the HPC provider are prevented from affecting user software stacks, ensuring reliability in results.

Furthermore, through the WLM integration, we achieve Seamless Integration (P4). The WLM plugin orchestrates the multiple container toolset components during the instantiation of the user environment, eliminating the need to manage these components manually, simplifying user workflows, and ensuring smooth operations. As such, even complex use cases can be abstracted to the point where end users do not directly interact with any container tool, effectively running their workflows as if they were operating natively on a traditional HPC system.

Lastly, the system has been tailored to ensure a Minimal Learning Curve (P5). Users familiar with WLM on traditional HPC systems will find the container-first enhancements easily accessible. The EDF's straightforward nature and the system's seamless integration of core functionalities allow users to become proficient quickly, with minimal additional training required.

By implementing the principles of containers as first-class elements, we present a solution where container instantiation becomes entirely transparent for the end user. This advancement allows all HPC workflows to be executed within containers, bringing to full fruition the benefits of user environment customization, flexibility, and decoupling. Containers are thus transformed into fundamental, first-class components of the HPC ecosystem, evolving how HPC platforms are prepared, deployed, operated, and maintained.

III. Pathfinder Implementation

A pathfinder version of the described solution was developed by combining established file formats, modified open source tools, and modular components built for industry standard containers in HPC.

A. The Environment Definition File format

The Environment Definition File (EDF) is an artifact intended to represent in a declarative and prescriptive fashion the particular details which transform a container image into a usable HPC environment on a given computing system. It

must support the most popular and useful container features, so that it can substitute the extensive use of container-related command line options, which is frequent in real-world HPC use cases and results in long, elaborate, and unwieldy command lines. It must be accessible and human-friendly, so that users can quickly understand and modify it to their needs. It must be composable and extensible, so that new environments can be built upon established ones for a given system or project without repeating or duplicating settings.

To implement the concept of the EDF, the TOML [5] file format was chosen. TOML was selected for its ease of both reading and writing, for its support of comments, for the simplicity of integrating a parser in the source code, and for its previous adoption by other container tools (for example Podman [6] and Singularity [7]) as a format for configuration files.

With reference to Listing 1, the current implementation of the EDF supports the following features:

- defining one or more base environments to conveniently create variants of existing EDFs (line 1), for example adding debug settings and tools - settings from base environments are imported incrementally according to the order given in the value;
- selecting the container image to use for the environment (line 5) - images can be specified either as registry references or as absolute paths to a Squashfs file;
- providing a recipe file to build the image (line 6) - this feature is used in the case the image parameter (line 5) is a filesystem path to a non-existing file;
- setting the initial working directory for the container (line 7);
- choosing whether to execute the container image entrypoint (line 8);
- setting the container's filesystem as writable or read-only (line 9);
- defining a list of bind mounts to be performed within the container (line 10);
- setting, modifying, or unsetting environment variables in the container (line 16);
- defining OCI-like annotations[1] for the container, which can be used for various purposes, including activation and control of hook programs (line 21). The EDF supports annotations through a TOML table, so it's possible to improve readability and scoping of related annotations by declaring TOML sub-tables.

To improve the convenience of writing, managing, and sharing Environment Definition Files, two more general features are supported. First, parameter values can contain relative filesystem paths, so that EDFs can be stored alongside application code, both on a computing cluster and on Source Code Management (SCM) systems. Second, host environment variables can be referenced and expanded within EDFs using the dollar sign and curly braces syntax, making files quicker to adapt when changing systems or users.

[1]In the context of the OCI specifications [8], annotations are key-value pairs which represent arbitrary metadata for different entities, including containers; it is recommended that annotation keys adopt a reverse domain name notation.

```
1  base_environment = [
2      "parent_env0",
3      "research_group_env"
4  ]
5  image = "nvcr.io/nvidia/nvhpc:23.11-devel-
       cuda_multi-ubuntu22.04"
6  containerfile = "./Containerfile.devtools"
7  workdir = "/workdir"
8  entrypoint = false
9  writable = true
10 mounts = [
11     "/user/experiment:/workdir",
12     "/configs:/workdir/configs",
13     "${SCRATCH}/data:/workdir/data"
14 ]
15
16 [env]
17 MELLANOX_VISIBLE_DEVICES = "none"
18 CONFDIR = "/workdir/configs"
19 UNSET_BEFORE_RUNNING = ""
20
21 [annotations]
22 com.hooks.cxi.enable = true
23 com.hooks.ofi_nccl.version = "cuda12"
```

Listing 1: EDF example. This is meant only to represent the currently supported parameters and syntax, and does not necessarily reflect an actual EDF use case.

B. Workload Manager plugin and container runtime

Workload manager integration and container management are implemented by extending the open source software Pyxis and Enroot [9], which are respectively a SPANK[2] plugin for the Slurm workload manager [10] and an HPC-focused container runtime. This combination is chosen because Pyxis already provides many of the desired features (and therefore the possibility to expedite development), however it is only intended to work in conjunction with the Enroot runtime.

Pyxis is extended by adding the following features:

- the `--environment` option for the `srun` command, through which the user can specify an EDF, and therefore a containerized environment in which to execute a job;
- support for the `EDF_PATH` environment variable, to define a custom search path for EDFs and facilitate referencing frequently used environments;
- a plugin configuration option to define a search path for system-wide EDFs, thus enabling the possibility to provide default environments for the system;
- a TOML parser to decode the EDF file specified by the `--environment` option and map its entries to container features;
- support for starting containers directly from images in the Squashfs [11] file format - the motivations behind this feature are discussed in Section IV.

[2]SPANK stands for Slurm Plug-in Architecture for Node and job [K]control

In turn, Enroot is modified in the following aspects:

- when pulling images from remote Docker registries, layers are saved directly if they are compressed using the Zstandard [12] (zstd) format: normally Enroot extracts downloaded layers and re-compresses them using zstd before finally saving them to its layer cache - this is done to leverage zstd's decompression speed when extracting cached layers ahead of flattening and squashing them, and it's advantageous if layers are originally compressed in a format that is slower to decompress (e.g. gzip [13]); however, the extraction and re-compression are redundant if the registry already sends a zstd-compressed layer, and can be skipped;
- when importing images from local Podman repositories, the Squashfs image is generated directly from a Podman container instead of exporting the container's contents as a tar archive and extracting them again - a more detailed description of this leaner, faster procedure is provided in Section V-D;
- specific configuration options for squashfuse [14] and fuse-overlayfs [15] programs were added to have more fine-grained control of the binaries used; this allows to tailor the installation better to the characteristics or limits of the host operating system. One such example is provided in Section IV.

C. Running containerized environments

In accordance with the principles stated in Section II, our implementation is intended to be familiar, intuitive, and seamless to use compared to a traditional HPC system.

A single option to Slurm commands (`--environment`) is required to request a job to be run in a containerized environment. The option takes one mandatory argument, which is the absolute path to the EDF describing the desired environment, for example:

```
$ srun --environment=$SCRATCH/edf/debian.toml
    cat /etc/os-release
```

The EDF is looked up and parsed by the modified Pyxis WLM plugin, which also implements a search path feature. By default the search path for user EDFs is initialized to `$HOME/.edf`, but can be controlled through the `EDF_PATH` environment variable, which must be a colon-separated list of absolute paths to directories, similarly to the `PATH` and `LD_LIBRARY_PATH` variables. The complete search path is obtained by appending the system-wide search path to the user-specific one, if the former is defined in the WLM plugin configuration. If a file is located in the EDF search path, the argument to the command line option can be just the environment name, that is the name of the file without the `.toml` extension, for example:

```
$ ls -l $SCRATCH/example-project/
fedora-env.toml
$ export EDF_PATH=$SCRATCH/example-project/
$ srun --environment=fedora-env cat /etc/os-
    release
```

```
1  $ cat example.sbatch
2  #!/bin/bash -l
3  #SBATCH --job-name=edf-example
4  #SBATCH --environment=debian
5  #SBATCH --time=0:01:00
6  #SBATCH --nodes=1
7  #SBATCH --output=slurm-%x.out
8
9  # Run job step
10 srun cat /etc/os-release
```

Listing 2: Example of a Slurm batch script using `--environment` as an `sbatch` option to select an EDF.

The `--environment` command line option can also be used within batch scripts as an `#SBATCH` option, as shown in Listing 2. It is important to note that in such a case all the contents of the script are executed within the containerized environment, therefore Slurm commands like `srun` or `scontrol` must be available in the container. This can be achieved through a container hook like the one described in Section III-D1.

A current limitation in the support for batch scripts is that when an EDF is selected via an sbatch option, all subsequent commands and job steps will be executed within that specific environment, without the possibility to choose a different one. If a batch use case requires switching between multiple container environments or if it's not possible to setup Slurm commands within containers, the `--environment` option can still be used explicitly in `srun` command lines when starting job steps.

D. Container Hooks

A number of container hooks work in synergy with the WLM plugin to expose the full spectrum of system features into containers and deliver a consistent and transparent user experience. Hooks are accessory, standalone programs which are called by container runtimes at various points during the lifetime of a container to execute arbitrary actions, often related to container customization, like performing additional bind mounts. In this regard, compatibility with the hook interface defined by the Open Containers Initiative (OCI) [8] Runtime Specification [16] is highly desirable, since complying with an open industry standard allows hook programs to be reusable across different tools in a straightforward way.

Enroot supports custom 'pre-start' hooks in the form of Bash scripts, which are executed right before the containerized user application starts and must employ specific syntax elements. However, Enroot does not natively support OCI hooks due to its non-compliance with OCI specifications. To overcome this limitation, we developed a custom Enroot hook that acts as an "adapter" for OCI hook executables. This adapter simulates an OCI-compliant bundle and container state, enabling OCI hooks to function as if they were being executed by an OCI-compliant runtime. This solution allows the integration of OCI hooks such as those developed in the scope of the Sarus [17] project. As a result, the presented

container engine actually utilizes both regular Enroot hooks and OCI hooks.

We now outline the specific hooks implemented in our pathfinder implementation to enable the containers-first approach. These hooks use the adapter mechanism described above to extend functionality, ensuring robust and versatile container operations. This integration is crucial for enhancing the flexibility, efficiency, and usability of our containerized user environments, and aligning with the strategic objectives of a containers-first approach.

1) Slurm hook: Mounts Slurm binaries, directories, configurations and dependencies into containers, so that it is possible to use Slurm CLI commands within interactive jobs or in batch scripts, whose content is executed within containers when using the `--environment` option and the EDF.

2) Libfabric hook for Slingshot 11: Enables connectivity through the HPE Slingshot 11 high-speed network within containers. This is achieved by mounting the custom libfabric [18] installation provided by HPE in Cray EX systems (which includes the user-space portion of Slingshot in the form of the CXI libfabric provider), alongside the related dependencies. The hook also creates a drop-in configuration file for the `ldconfig` utility, refreshes the dynamic linker cache in the container to include the newly-mounted shared libraries, and can optionally set environment variables to customize settings for libfabric or the CXI provider.

3) NVIDIA GPU hook: Provides access to NVIDIA CUDA GPUs inside containers by using components from the NVIDIA Container Toolkit [19]. This hook is bundled with the Enroot source code and is reused without any modification.

4) OFI NCCL plugin hook: The NVIDIA Collective Communication Library (NCCL) [20] is widely used to implement optimized communication between NVIDIA GPUs, particularly within Machine Learning frameworks and applications. Despite its broad adoption, NCCL's design and APIs cannot interface directly with libfabric. This is a problem on Slingshot 11-based systems, where the network software stack is exposed to user applications through libfabric. Nevertheless, NCCL supports external network plugins which allow the library to leverage arbitrary or custom network technologies. Amazon Web Services has developed one such plugin, called "AWS OFI NCCL"[3] and available as open source software [21]. The plugin allows NCCL to utilize libfabric as a backend to access network hardware, thereby extending support to interconnects exclusively accessible through libfabric.

The "OFI NCCL plugin hook" bind mounts the plugin binary into the container in a standard linker path and sets an environment variable to instruct NCCL to load that specific plugin, since a default NCCL may be already installed in container images.

The container hook supports selection of different plugin variants through an annotation provided in the EDF, so that users can match the version of CUDA in their containerized environment with the corresponding build of the plugin.

5) SSH hook: The SSH hook bundled with the Sarus [17] container engine is an OCI hook capable of enabling SSH connections inside containers. It implements several functionalities, such as generating dedicated SSH keys to not expose actual user keys from the host system, mounting SSH binaries into containers and starting up an SSH daemon at container-creation time, and ensuring that incoming connections are provided with an appropriate and consistent execution context (for example in terms of environment variables and working directory). The Sarus SSH hook is intended to augment containers which are not originally equipped with SSH utilities, and therefore does not pose any requirement on the container: the SSH software consists in a statically linked version of Dropbear, a minimal SSH server and client implementation.

Among its features, the hook can authorize connections using an arbitrary public key provided by the user, taking advantage of Dropbear's compatibility with OpenSSH public key authentication. This is useful to support not just direct connections from user workstations, but also remote debugging use cases with tools like Microsoft's Visual Studio Code [22]. An example is described in Section V-B.

Despite the Sarus SSH hook being an OCI-compliant hook, we are nevertheless able to call it from Enroot thanks to a suitable adapter hook, which also manages the cleanup of the SSH daemon process once the container terminates.

6) MPI replacement hook: Through the use of an Enroot adapter hook, our pathfinder container engine implementation can also utilize the Sarus MPI hook, which performs on-the-fly complete replacement of an MPI installation in the container with an ABI-compatible host counterpart. This operation transparently provides container applications with an MPI stack tuned for the system where the container is running. When ABI compatibility between container and host MPI stacks is satisfied, the Sarus MPI hook enables near-native communication performance for generic, portable containers and advanced features like RDMA, which may not be available in the original MPI from the container image.

7) Slurm PMIx hook: Provides support for the Slurm integration of the PMIx interface [23]. When Slurm uses PMIx to coordinate multi-rank jobs, this hook automatically bind mounts into the container the PMIx directories created by the WLM and ensures that related container environment variables are set. This hook is bundled with the Enroot source code and is reused without any modification.

IV. Deployment and configuration for an HPE Cray EX system

In this section we discuss the deployment of the pathfinder container solution introduced in Section III on an HPE Cray EX system at CSCS, and how the configuration is tailored to the characteristics of said system. The cluster features the HPE Slingshot 11 [24] high-speed network, which connects the compute nodes between themselves and to an HPE ClusterStor E1000D Lustre-based parallel filesystem.

[3]OFI stands for Open Fabrics Interfaces, which in this context is another way to reference the libfabric framework.

The binaries for the modified versions of Pyxis and Enroot, and the Sarus OCI hooks are installed via RPM packages, while configuration files and the Enroot hook scripts (Section III-D) are generated at installation time from templates managed through an Infrastructure as Code (IaC) approach.

For container image pulls, Enroot is configured to cache downloaded image layers in individual user folders on the Lustre filesystem, and to create Squashfs images using the Zstandard compression algorithm [12] with compression level 1, without compressing data blocks. Zstd offers an excellent balance between compression speed and compression ratio, allowing to noticeably reduce the time spent importing images while producing comparable image sizes to the default compressor based on zlib [25].

The Pyxis plugin is configured to start containers directly from Squashfs images, instead of its typical mode of operation which uses plain directories to serve as the root filesystem (*rootfs*) for the container. This setting is made possible by one of the new features we implemented in Pyxis (Section III), and is chosen for three reasons. First, mounting the rootfs from an image in the Squashfs format happens almost instantaneously, while a plain directory must be extracted from the image before the mount operation can take place; therefore, the direct mount of the Squashfs image significantly reduces the container startup time and enhances the implementation of the Seamless Integration principle (P4). Second, when the data are located on a parallel filesystem, a filesystem loop mounted from a Squashfs image achieves superior I/O performance compared to a plain directory [3], especially in the case of random access patterns, such as those generated by large Python application stacks. Third, if a plain directory is extracted to a node-local, in-memory filesystem in order to prevent the I/O performance degradation, this would consume a portion of the RAM available to user applications; hence, the direct mount of a Squashfs image also preserves the amount of memory usable by applications.

To perform the Squashfs and overlay mounts required to setup the container, Enroot is configured to use specific binaries. On the one hand, `squashfuse_ll` 0.5.1 is chosen because, compared to the regular `squashfuse` program, its low-level FUSE API implementation and multithreading support enable the container filesystem to obtain higher performance [26]. On the other hand, fuse-overlayfs 1.13 is in use after we verified that employing version 1.1.0 of the utility, as provided by the base operating system, was causing program hangs when accessing specific symlink patterns.

V. Example use cases

This section describes examples of EDFs and workflow elements taken from real-world use cases by early access users and staff on CSCS systems.

A. Training an ML/AI model

FourCastNet [27] is a global data-driven weather forecasting model based on a deep learning approach and the use of Adaptive Fourier Neural Operators (AFNO). Compared to traditional Numerical Weather Prediction (NWP) models based on the solution of differential equations, FourCastNet generates final results in a fraction of the time, with comparable (or, for some parameters, higher) forecasting accuracy.

Despite the complexity of the application, an environment to train FourCastNet can be expressed with a fairly simple EDF, as shown by the example in Listing 3. The container image provides the source code of the model [28] on top of an NVIDIA NGC [29] PyTorch [30] base image, and is entered in the EDF as a pre-pulled Squashfs file path to skip the remote pull normally performed by Pyxis and start the container sooner. An array of bind mounts is defined taking advantage of TOML syntax and entering one array element per line to improve readability. An important aspect to consider when setting up FourCastNet for training is how to structure the input files and directories, which are dictated by configuration settings of the given model. In this context, a container is convenient to use because the model configuration can remain unaltered and different datasets can be swapped in by changing the mount source paths from the host. The EDF also includes annotations that activate specific hooks detailed in Section III-D. These hooks enable the integration of the libfabric framework with the Slingshot 11 interconnect via the CXI provider. They also ensure that the AWS OFI NCCL plugin, which is compatible with the version of the CUDA runtime found in the container, is properly mounted.

This setup exemplifies how the containers-first concept streamlines the deployment of complex HPC applications and usage workflows. It separates them from other system components, treating each as an independent, self-contained unit. This isolation is achieved through a combination of the container, the EDF, and necessary hooks.

One of the main benefits of this approach is that it hides the internal workings and complex build stack of the application from the system software stack. On the other hand, it also hides the majority of the system software from the application itself. This prevents the application from depending on specific system components, allowing it to interact only with the elements and interfaces exposed into the container. These components are then explicitly tracked and controlled within the EDF. As a result, this method supports modularity, encapsulation, and abstraction, which enhance overall deployment robustness and manageability.

B. Remote debugging with Visual Studio Code

Debugging code on HPC systems can be a complex task, and even more so if the code runs within a container. The workflow is additionally challenged by the preference of some users, often coming from outside a traditional HPC background, to develop and debug code remotely from their laptops, employing graphically rich tools, like Microsoft's Visual Studio Code [22].

Our containers-first solution can facilitate this use case in two ways: by using an EDF to modularize the debugging setup and by making remote connections more convenient through a container hook.

```
image = "${SCRATCH}/fcn-collaboration/ml+
    fourcastnet+22.12.sqsh"

mounts = [
    "${SCRATCH}/fcn-collaboration:/workdir",
    "${SCRATCH}/fcn-results/:/output",
    "${SCRATCH}/init_dir_fcn:/init_dir",
    "${SCRATCH}/fcn-collaboration/
    profiler_output:/profiler_output",
    "${SCRATCH}/m01/p_levels_raw:/input"
]

workdir = "/workdir"

[annotations]
com.hooks.cxi.enable = true
com.hooks.ofi_nccl.version = "cuda11"
```

Listing 3: Example of an EDF used to train FourCastNet on a CSCS cluster.

Regarding the first aspect, consider a containerized environment based on an NGC image for PyTorch, which could be used to develop and run Python codes. An example EDF for such an environment is represented in Listing 4. To carry out remote debugging, additional tools and environment features are required; similarly, a working directory dedicated to debugging might be preferred to the one used for regular runs. EDFs can designate other TOML files as bases, and therefore compose environments incrementally from different files. This capability is useful to abstract the debugging elements from the base development environment, leaving the latter unaltered and making the definition of the debugging environment akin to an "add-on module" to the base (Listing 5). Instead of creating a nearly identical environment which might have to be maintained separately, the debugging setup is defined by modular composition and seamlessly inherits changes from the base environment.

Regarding the second aspect, a combination of EDF and SSH hook from Section III-D5 can be used to minimize the complexity of setting up remote connections. Before starting the container, the user copies to the HPC system the public portion of the key they intend to use for connecting. Then, they enter in the debug EDF the annotations to drive the execution of the SSH hook, as shown in Listing 5. In particular, the `authorize_ssh_key` annotation indicates the path to the user public key file on the host system. When the container is created, the hook introduces a statically linked SSH server into the container, automates its launch as a daemon and, as instructed by the `authorize_ssh_key` annotation, adds the user public key to those accepted for incoming connections. Afterwards, the user adds an OpenSSH configuration entry in their remote workstation, indicating the address of the cluster node where the container is running, and matching the port and private key with those entered in the EDF. Finally, the user opens the Visual Studio Code Remote SSH extension and uses the newly created OpenSSH configuration to connect directly to the remote container. At

```
image = "nvcr.io/nvidia/pytorch:24.01-py3"

mounts = [
    "${SCRATCH}$:/scratch"
]

workdir = "/scratch"
writable = true
```

Listing 4: Example of an EDF defining a development environment with PyTorch, CUDA and optimized NVIDIA ML dependencies.

```
base_environment = "pytorch-24.01"

workdir = "/scratch/ml/demo-Debug"
writable = true

[annotations.com.hooks.ssh]
enabled = "true"
port = "51234"
authorize_ssh_key = "${HOME}/.ssh/cscs-key.
    pub"
```

Listing 5: Example of an EDF adding elements useful for remote debugging to a base environment. Notice the definition of a base environment in line 1 and the annotations to activate the SSH container hook starting on line 6

this point, the container environment running in the Cray EX system can be controlled remotely by Visual Studio Code's terminal. Likewise, VS Code's editor and debugger can be used to edit and run code, break execution and inspect the program state on the compute node.

In summary, the EDF offers an intuitive format to define composable settings for debugging, while the SSH hook enables a simplified connection procedure. The latter is achieved without bundling SSH software into a container image and without handling the startup of the SSH daemon in a container entrypoint script. The approach outlined in this section can also be applied to Jupyter [31] notebooks: the Jupyter kernel is run on the HPC system while the notebook web interface and the Visual Studio Code debugger are executed remotely on the user's workstation.

C. Adopting system-specific defaults

From a user perspective, a great deal of the EDF's value lies in defining environments tailored around applications and use cases, and being able to deploy them in a convenient, flexible, and reliable way. Yet, while the focus of many execution environments is the application, system-specific settings are often needed to get the most out of a given platform. Cluster owners, on their side, are interested in providing default settings to make the user experience more convenient and to facilitate access to system features.

For example, Listing 6 represents an EDF with administrator-defined defaults for a CSCS cluster assigned to Machine Learning projects. It enables fundamental system features, like the user's scratch directory and the hook for the libfabric-based connectivity to the Slingshot network

```
mounts = [
    "${HOME}/.bash_history:/.bash_history",
    "${SCRATCH}:${SCRATCH}",
    "/tmp:/tmp"
]

[env]
HISTFILE = "/.bash_history"

[annotations]
com.hooks.cxi.enable = true
com.hooks.ofi_nccl.version = "cuda12"
```

Listing 6: Example of an EDF providing system settings to be reused as defaults.

```
base_environment = "cscs-mlp"
image = "nvcr.io/nvidia/pytorch:24.01-py3"
```

Listing 7: Example of an use-case specific EDF using a base environment to inherit system defaults on a CSCS cluster for Machine Learning.

(Section III-D2). It specifies domain-specific settings, like mounting the AWS OFI NCCL plugin for CUDA 12 using the related hook from Section III-D4. Finally, it defines convenience features to increase container transparency, like mounting the Bash history and the `/tmp` directory from the host system. The latter mount is especially useful when different containerized processes on the same physical node (whether running concurrently or in subsequent batch job steps) expect to consistently use the contents of the temporary filesystem directory. This EDF is stored in a location which is configured as the path for system-wide environment definitions in the WLM plugin. Notice that in Listing 6 no container image is defined, and therefore the environment definition is incomplete. A single EDF does not need to be deployable on its own if it is combined with others to obtain a fully valid definition.

The composability of EDFs allows to transparently inherit system-default settings into more portable, use case-specific environment definitions. Assuming that the file from Listing 6 was saved with the name `cscs-mlp.toml`, Listing 7 represents a user EDF adopting it as base to acquire all its parameters. Because of its location in the configured search path for system EDFs, the base environment can be identified just with the `"cscs-mlp"` string. The user EDF only specifies the details which are related to the desired application, in this case just a PyTorch image. As such, the user EDF can easily switch between different contexts, or different platforms, by adjusting the base environment parameter or overriding the minimum required amount of settings. This is convenient for users who prefer storing EDFs in SCM tools alongside their project's source code.

D. Building customized image variants

Due to the popularity of containers, vendors and open source communities offer a wealth software stacks in the form of container images. It is not uncommon that users only require to make small modifications to an already available image to meet their needs. When doing so, the best practices advise to build a new, customized image from a recipe file, in order to preserve the reproducibility and prescriptiveness characteristics brought by containers. However, building a custom image to be used on an HPC system is inconvenient for some users. There are several possible reasons for this, including but not limited to the following:

- users might not have access to a computer where to build the image with the same microarchitecture as the target HPC system;
- users might not have access to a container registry where to upload their image variant;
- building on a separate system, uploading to a registry, and pulling is time consuming and not worth for experimenting with small changes;
- direct interaction with tools to build and import images is undesirable or confusing for inexperienced users.

Our containerized environments implementation provides a way to build and deploy custom container images directly on the HPC system with minimal user intervention. As an example, consider the following case: a user wishes to augment a PyTorch image from the NVIDIA GPU Cloud with the Python package to connect to the Weights & Biases platform, used by some ML practitioners to assist in model development. This can be achieved in two Containerfile[4] instructions, as shown in Listing 8.

```
FROM nvcr.io/nvidia/pytorch:24.01-py3
RUN pip install wandb
```

Listing 8: Example of a Containerfile adding a Python package to a base image with `pip`.

```
image = "${SCRATCH}/my_images/pytorch-wandb.sqsh"
containerfile = "${SCRATCH}/containerfiles/Containerfile.pytorch-24.01-wandb"
```

Listing 9: Example of an EDF which creates a custom image variant from a Containerfile.

The `containerfile` parameter in the EDF is used to associate a Containerfile to a containerized environment. When an EDF image is specified as a filesystem path, the WLM plugin checks if it corresponds to an existing file. If no file exists at the given path, an image is built according to the provided Containerfile using a container build tool like Buildah or Podman. The new image is then imported by Enroot (converting it to a Squashfs-based format suitable for HPC) and saved at the path indicated by the `image` EDF parameter. Listing 9 provides an example of such a use case.

[4]"Containerfile" is a vendor-neutral nomenclature to refer to an image recipe for an OCI-compliant build tool. In practice, the term can be considered equivalent to "Dockerfile".

If Podman is used as build tool, we implemented an optimized import procedure in Enroot to skip explicit layer flattening or exporting the image contents from Podman as a tar archive. Instead, a Podman container is created to have all the layers merged by an OverlayFS filesystem with negligible overhead. The root filesystem (*rootfs*) of said container is then mounted to a location which can be accessed from the host, and the `mksquashfs` utility is invoked directly on the rootfs directory. This procedure reduces the time spent in importing an image to only the time required for conversion into Squashfs. The approach is implemented by combining Podman's `create`, `unshare`, and `mount` commands, and is inspired by the procedure followed by Enroot when pulling images from remote repositories, which also leverages an OverlayFS to skip image layer flattening.

The end result of the `containerfile` parameter in the EDF is that building container images is transparently integrated into the same command used to run applications. The feature streamlines the process of introducing modifications of arbitrary complexity to reference images, facilitating the creation of radically different images as well.

The ability to quickly and easily generate image variants from Containerfiles removes the need to resort to startup or entrypoint scripts in order to make on-the-fly customizations to containers.

VI. Conclusions and Future Work

In this paper, we introduce a containers-first strategy for user environments on HPE Cray EX systems that enhances stability, flexibility, and efficiency. The containers-first approach improves the management and deployment of HPC user environments while leveraging, enhancing, and integrating multiple container tools to address the challenges traditionally associated with integrating containers into HPC to provide frictionless containerized user environments.

The presented approach isolates user environments from the system software stack, enabling reproducibility and streamlining deployment processes. By integrating containers into HPC environments, we alleviate common usage friction, simplify configuration, and facilitate the deployment of complex scientific software stacks. The modular and adaptable nature of the containers-first approach allows users to maintain consistent environments across various systems, improving the reproducibility of experiments and sharing of results. Moreover, the operational improvements make the system more convenient to use, reducing the time to solution for complex applications and workflows.

In conclusion, the containers-first approach represents a paradigm shift in managing user environments for HPC systems. It addresses the current needs of HPC management and makes HPC more accessible, modular, immediate, and adaptable. Looking ahead, we plan to further refine and expand this approach. Our focus will include improved integration with container build tools and registries to better support the increasingly complex, fast-paced, innovative, collaborative, and reproducible workflows on traditional HPC and emerging fields such as AI and ML.

Acknowledgments

The authors would like to acknowledge the contributions of the following colleagues at the Swiss National Supercomputing Centre: Dr. Fawzi Mohamed for collaborating on the development and deployment of the pathfinder container engine implementation and the Environment Definition File format; Dr. Michele Brambilla for contributing to the development of the container hooks; Dr. Theofilos Manitaras for providing the use case material for the FourCastNet ML/AI model example; Stefano Schuppli for sharing material used in the remote debugging use case example.

References

[1] S. P. Huber, S. Zoupanos, M. Uhrin, L. Talirz, L. Kahle, R. Häuselmann, D. Gresch, T. Müller, A. V. Yakutovich, C. W. Andersen *et al.*, "AiiDA 1.0, a scalable computational infrastructure for automated reproducible workflows and data provenance," *Scientific data*, vol. 7, no. 1, p. 300, 2020.

[2] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame, "Nextflow enables reproducible computational workflows," *Nature biotechnology*, vol. 35, no. 4, pp. 316–319, 2017.

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

[4] NVIDIA Corporation. (2019) Pyxis: Container plugin for the Slurm Workload Manager. [Online]. Available: https://github.com/NVIDIA/pyxis

[5] Tom Preston-Werner, Pradyun Gedam, et al. (2013) TOML: Tom's Obvious, Minimal Language. [Online]. Available: https://toml.io

[6] The Podman authors. (2017) Podman: A tool for managing OCI containers and pods. [Online]. Available: https://github.com/containers/podman

[7] 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.

[8] Open Container Initiative. (2022) Open Container Initiative home page. [Online]. Available: https://www.opencontainers.org/

[9] NVIDIA Corporation. (2018) Enroot: A simple yet powerful tool to turn traditional container/OS images into unprivileged sandboxes. [Online]. Available: https://github.com/NVIDIA/enroot

[10] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple Linux Utility for Resource Management," in *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60.

[11] The kernel development community. (2023) Squashfs 4.0 Filesystem. [Online]. Available: https://www.kernel.org/doc/html/v6.8/filesystems/squashfs.html

[12] Y. Collet and M. Kucherawy, "Zstandard compression and the 'application/zstd' media type," RFC 8878, USA, 2021. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8878

[13] P. Deutsch, "Gzip file format specification version 4.3," RFC 1952, USA, 1996. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1952

[14] Dave Vasilevsky and Phillip Lougher. (2012) squashfuse - Mount SquashFS archives using FUSE. [Online]. Available: https://github.com/vasi/squashfuse

[15] The fuse-overlayfs authors. (2018) fuse-overlayfs: FUSE implementation for overlayfs. [Online]. Available: https://github.com/containers/fuse-overlayfs

[16] Open Container Initiative. (2015) Open Container Initiative Runtime Specification. [Online]. Available: https://github.com/opencontainers/runtime-spec

[17] 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.

[18] OpenFabrics Interfaces Working Group. (2022) Libfabric home page. [Online]. Available: https://ofiwg.github.io/libfabric/

[19] NVIDIA Corporation. (2019) NVIDIA Container Toolkit: Build and run containers leveraging NVIDIA GPUs. [Online]. Available: https://github.com/NVIDIA/nvidia-container-toolkit
[20] NVIDIA Corporation. (2024) NVIDIA Collective Communications Library (NCCL). [Online]. Available: https://developer.nvidia.com/nccl
[21] Amazon Web Services, Inc. (2018) AWS OFI NCCL. [Online]. Available: https://github.com/aws/aws-ofi-nccl
[22] Microsoft Corporation. (2015) Visual Studio Code. [Online]. Available: https://code.visualstudio.com/
[23] R. H. Castain, J. Hursey, A. Bouteiller, and D. Solt, "PMIx: Process management for exascale environments," *Parallel Computing*, vol. 79, pp. 9–29, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819118302424
[24] Hewlett Packard Enterprise Company. (2022) HPE Slingshot Interconnect. [Online]. Available: https://www.hpe.com/us/en/compute/hpc/slingshot-interconnect.html
[25] Greg Roelofs and Jean-loup Gailly and Mark Adler. (1996) zlib compression library home page. [Online]. Available: https://www.zlib.net/
[26] D. Dykstra, "Apptainer Without Setuid," *arXiv preprint arXiv:2208.12106*, 2022.
[27] J. Pathak, S. Subramanian, P. Harrington, S. Raja, A. Chattopadhyay, M. Mardani, T. Kurth, D. Hall, Z. Li, K. Azizzadenesheli, P. Hassanzadeh, K. Kashinath, and A. Anandkumar, "Fourcastnet: A global data-driven high-resolution weather model using adaptive fourier neural operators," *arXiv preprint arXiv:2202.11214*, 2022.
[28] Pathak, Jaideep and Subramanian, Shashank and Harrington, Peter and Raja, Sanjeev and Chattopadhyay, Ashesh and Mardani, Morteza and Kurth, Thorsten and Hall, David and Li, Zongyi and Azizzadenesheli, Kamyar and Hassanzadeh, Pedram and Kashinath, Karthik and Anandkumar, Animashree. (2022) FourCastNet source code, data, and model weights. [Online]. Available: https://github.com/NVlabs/FourCastNet
[29] NVIDIA Corporation. (2024) NVIDIA NGC. [Online]. Available: https://www.nvidia.com/en-us/gpu-cloud/
[30] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, G. Chauhan, A. Chourdia, W. Constable, A. Desmaison, Z. DeVito, E. Ellison, W. Feng, J. Gong, M. Gschwind, B. Hirsh, S. Huang, K. Kalambarkar, L. Kirsch, M. Lazos, M. Lezcano, Y. Liang, J. Liang, Y. Lu, C. Luk, B. Maher, Y. Pan, C. Puhrsch, M. Reso, M. Saroufim, M. Y. Siraichi, H. Suk, M. Suo, P. Tillet, E. Wang, X. Wang, W. Wen, S. Zhang, X. Zhao, K. Zhou, R. Zou, A. Mathews, G. Chanan, P. Wu, and S. Chintala, "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation," in *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, Apr. 2024. [Online]. Available: https://pytorch.org/assets/pytorch2-2.pdf
[31] B. E. Granger and F. Pérez, "Jupyter: Thinking and storytelling with code and data," *Computing in Science & Engineering*, vol. 23, no. 2, pp. 7–14, 2021.