Modern Software Deployment on a Multi-Tenant Cray-EX System

John Biddiscombe
ETH Zürich, Swiss National Supercomputing Center (CSCS)
Lugano, Switzerland
john.biddiscombe@cscs.ch

Andreas Fink
ETH Zürich, Swiss National Supercomputing Center (CSCS)
Lugano, Switzerland
andreas.findk@cscs.ch

Ben Cumming
ETH Zürich, Swiss National Supercomputing Center (CSCS)
Lugano, Switzerland
bcumming@cscs.ch

Simon Pintarelli
ETH Zürich, Swiss National Supercomputing Center (CSCS)
Lugano, Switzerland
simon.pintarelli@cscs.ch

***Abstract*—User-facing software – libraries, tools, applications and programming environments tuned for the node and network architecture – is a key part of HPC centers' service offering. Teams that maintain and support this software face challenges: providing a stable software platform for users with long running projects while also providing the latest versions of software for developers; giving full responsibility to build, modify and deploy the whole software stack to staff who do not have root access; and reproducible deployment based on GitOps practices.**

CSCS addresses these challenges on Alps by using small independent software environments called uenv, which deploy from text-file recipes without requiring installation of the Cray Programming Environment. This paper discusses installing communication libraries from HPE and NVIDIA with Slingshot support; the CI/CD pipeline that builds uenv and deploys them in a container registry; and the command line tools and SLURM plugin that interface users with the software environments. We demonstrate diverse use cases such as JupyterHub, summarize the user and support team experience, and document how to build and deploy CPE containers.

***Index Terms*—CPE, spack, squashfs, slurm, gitops**

I. Introduction

A differentiating feature of Alps compared to other HPE Cray-EX deployments is how multi-tenancy is implemented and supported. Instead of deploying a single large SLURM cluster, the system is partitioned into non-overlapping SLURM and Kubernetes clusters, each customised for specific use cases and community requirements. The combination of tenants and use cases requires distributing responsibility for providing and supporting software environments to many individuals and teams inside CSCS.

Ideally software support staff would have end to end responsibility: the ability to install all software as users without interrupting the operation of the system, using automated CI/CD pipelines. Furthermore, to continue scaling support to more use cases and systems, the tooling and methods used need to:

- reduce the dependence of software stacks on the base OS image installed on each tenant
- decouple (and simplify) software environments
- automate deployment
- and, support users requiring custom software stacks.

CSCS has found that there is no silver bullet that meets all of these requirements. For example, HPC containers allow users to bring software stacks built elsewhere, they offer excellent isolation between environments, and container runtimes can inject optimized communication libraries making containerised software resilient to changes to the underlying system. But supporting containers alone is not sufficient – we also need to provide software optimised for the target platform, and support users building their own optimized containers. In other words, while providing a robust container runtime with SLURM integration reduces dependency on the base OS and decouples software stacks, .

At CUG 2023 CSCS presented Stackinator [2], a tool for building isolated and reproducible software environments as SquashFS images. The software stacks have no dependency on the Cray Programming Environment (CPE), and use custom Spack packages for installing the Cray-MPICH implementation of MPI and its dependencies as stand alone packages. Over the last two years CSCS have continued to develop Stackinator, and the user-facing tools for interacting with the software stacks, which we have dubbed *uenv*. This paper focusses on the full end to end process and tooling that CSCS has developed to deploy uenv, how users interact with them, and their impact on support staff.

In Section II the requirements discussed above will be enumerated as objectives with measurable aims. Then the tools and processes used to achieve our objectives will be introduced in three stages: building environments; deploying environments; and the user experience for users of the deployed environments in Section IV. How CSCS continues to provide CPE as a container, with recipes and lessons learnt

for building CPE in a container will be covered in Section III. Finally we will describe some interesting use cases for uenv in Section VI, and discuss the maintenance overheads and issues with our approach in Section VII.

We note that while much of the implementation is CSCS-specific, all of the software and pipelines discussed in this paper are in permissibly licensed open source repositories that can be used as is, or used for tips and guidance in developing solutions at other sites.

II. Objectives

We first establish concrete objectives that define the requirements for deploying software on HPC systems. These objectives can be used to evaluate the effectiveness of our proposed solutions. The objectives span technical, operational, and organizational aspects of software deployment.

A. Provide optimized software

Libraries that implement inter-node communication, for example MPI and NCCL, need to be optimized for the Slingshot 11 network in HPE Cray-EX systems. The method used by Stackinator to install Cray-MPICH using Spack [5] outside the CPE was demonstrated in [2]

To support more applications, including ML/AI and commercial applications , similarly robust methods are required for installing packages like OpenMPI, NCCL, NVSHMEM, cuFFTMp and cuSOLVERMp and with libfabric support:

- **aim**: we can deploy OpenMPI with Slingshot 11 support;
- **aim**: build libfabric and CXI from source;
- **aim**: we can deploy key NVIDIA communication libraries on Slingshot 11.

Note that the NVIDIA libraries are not discussed in this paper, because we ran out of time to finish test before publication.

B. Gitops deployment

With GitOps, infrastructure and environments are defined as code in version control systems like Git. This enables automated, reproducible deployments through CI/CD pipelines, while providing transparency, rollback capabilities, and a single source of truth for the entire system configuration.

We define the following criteria for a GitOps workflow for deploying software on a HPC system:

- **aim**: all software stacks are defined as text file recipes in a git repository;
- **aim**: a CI/CD pipeline builds, tests and deploys uenv images using triggers on the git repository;
- **aim**: each pipeline build deploys unique artifacts;
- **aim**: artifacts can be rebuilt from recipe.

C. Decouple environments

Many use cases on Alps require stable environments that do not change over the duration of multi-year projects, while others require updated versions of the same environments, or frequent rolling release style deployment to keep pace with rapidly changing software stacks (for example ML/AI and development projects). To support all use cases on the same system, we set the following requirements:

- **aim**: existing workflows based are unaffected by releasing new versions of the uenv;
- **aim**: minimise the chance that uenv are "broken" by updates to the underlying system, while being able to redeploy them quickly if there is a breaking change;
- **aim**: uenv that provide the latest versions of software (e.g. with pre-releases of cray-mpich or the latest PyTorch version) can be deployed immediately after their release.

D. End to end responsibility for software support teams.

The team responsible for supporting software environments should have full responsibility for their deployment. Building and deploying software should not require root privileges, modification to the system itself or the involvement of system administrators:

- **aim**: staff who support applications and programming environments should have end to end responsibility;
- **aim**: user communities can manage their own software deployments using the same tools;
- **aim**: individual users can build and share their own uenv images.

III. The Cray Programming Environment (CPE)

The Cray Programming Environment (CPE) provides a rich software stack: compilers, communication libraries, commonly used scientific libraries and tools. The CPE is provided by HPE as RPMs, that CSCS installed using Ansible and deployed to the nodes using the data virtualization service (DVS) [7].

To provide additional software to users, HPC centers with HPE Cray-EX systems typically deploy the CPE, then build additional software using the components it provides – primarily the compiler wrappers, cray-mpich, CUDA/ROCM and widely-used libraries like FFTw and netcdf. Centers have developed workflows that use on HPC package management tools like Spack and EasyBuild to build the software, store the configuration in git repositories, and deploy using pipelines with varying levels of automation [4], [6]. This additional software is deployed to a shared file system, where it is available for users to use alongside the CPE, as illustrated in Fig. 1.

The downside of this approach is that installing a new version of the CPE, which is released every 6 months, can be very disruptive:

- changes in the CPE often require reconfiguring and rebuilding centre-provided software – the task of updating Spack and EasyBuild to use new locations and versions of CPE software is particularly time consuming;
- users who have built upon this foundation might similarly have to rebuild, or be affected by new bugs in the new version of CPE;
- installing the new CPE requires building a new OS image, typically on a test and deploymebnt (TDS) system;
- deployment requires rebooting the system.

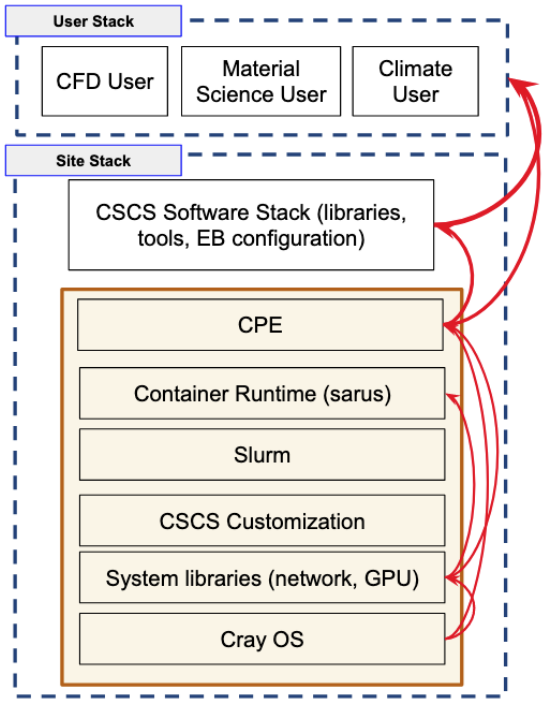

Fig. 1: The "standard" HPE-EX software stack, with the Cray OS, drivers, CPE and site-specific software in the system image, site-provided software installed on a shared file system. User-installed software depends on the software layers underneath. The red arrows indicate where changes to one layer have a knock on effect on other software layers, requiring rebuilds or reconfiguration.

These disruptions go against all of the objectives for software deployment outlined in Section II. This was the main motivation behind CSCS developing the software deployment methods described in this paper.

We note that as of the time of writing, CPE is still available through a `cray` module on the Daint cluster on Alps, for users who require it, or are still in the process of transitioning to uenv and containers. The `cray` module is the only module available to users on Alps when they first log in, and loading it loads the `PrgEnv-cray` meta-module and populates the full set of available modules. In Section V we describe how CSCS will continue providing CPE in a container, in order to improve how CPE is deployed, and help users make a more informed decision about their software environment.

IV. Methods

This section presents the components of CSCS' software deployment workflow, following the complete lifecycle from initial software compilation through to end-user interaction. We examine three key stages: the building of software components with their specific dependencies and optimizations, the deployment mechanisms across diverse computational environments, and the CLI and SLURM tools that provide these environments to users.

A. Building Software: Stackinator

The Stackinator tool, used to generate squashfs images from a YAML-based recipe, was described in detail at CUG 2023 [2]. The implementation details of Stackinator were covered in that paper, and have not changed materially. The method used by Stackinator to install Cray-MPICH using Spack [5] outside the CPE was demonstrated was shown in [2] This approach is used by Stackinator, however it can also be used directly with Spack – CSCS staff use it to install the most recent versions of Cray-MPICH on LUMI.

Stackinator has not needed fundamental modification since the original paper. There have been quality of life improvements, and the main new feature is improved support for generating *views*: flexible descriptions of how to modify (add, update or remove) environment variables when a uenv is loaded.

After two years of using Stackinator, we have observed that specifying version of Spack to use when building a uenv is defined in the recipe has been beneficial. The alternative approach is to fix the version of Spack at a site-level, build a Spack repository of site-specific and customised Spack packages, and install all software using that version of Spack. Our approach reduces the overhead of upgrading the version of Spack: each recipe upgrades when it is ready, and maintains a small set of custom packages (it any at all). There is a small centrally-managed Spack repository (`cray-mpich`, its dependencies and customisations of `nvhpc` and `cuda`). The approach is not perfect – Stackinator has to detect which version of Spack is being used, and customise the configuration to work around breaking changes introduced in each version of Spack. Furthermore, the next version of Spack will be 1.0, which introduces larger breaking changes, which will require changing the recipe format, which in turn more tightly couples the version of Stackinator with the recipe.

One of the large changes introduced to Spack is builtin packages for the recently open-sourced HPE Slingshot libraries (`libcxi`, `cxi-drivers`, `cassini-headers`), which are required to build our own network stack and properly support. We had hoped to show mature support for OpenMPI and NVIDIA communication libraries in the final version of this paper, however back-porting these packages to the most recent v0.23 release of Spack wasn't practical, so the results here are not deployed in the production pipeline in Section IV-B.

1) Libfabric and OpenMPI: Although support for Cray-MPICH is mature and well tested, it relies on an external package installed via RPM, namely libfabric. The recent release of source codes for Slingshot libraries (`libcxi`, `cxi-drivers`, `cassini-headers`) and their associated libfabric CXI provider gives us the incentive to provide more flexibility in networking layer customization, including:

- linking Cray-MPICH to non default (compiled from source) versions of Slingshot dependencies;
- supporting non-vendor provided MPI implementations such as OpenMPI;

- allowing custom versions/feature branches and options for both top level (MPI) and dependent libraries (`libfabric`, `cxi`, etc).

To achieve this, we have extended the `mpi` specification of the Stackinator YAML recipe so that users can override the defaults setup by cluster managers. The following extract (fig 2) shows how one can redefine the networking stack using the familiar spack syntax (note that yaml tags shown may change prior to next release). When combined with Stackinator's existing `package` customizations (where `package_attributes` such as `git` repository/location may be set), the new options give total control over the network configuration on a per cluster basis.

```
mpi:
  spec: openmpi@git.feature-5.0.6=main
  depends:
  - libfabric@git.lnx_refactor_main=main +debug
    fabrics=lnx,cxi,rxm,tcp,shm dev_path=/scratch/
    libfabric
  - libcxi@main
  - cxi-driver@main
  - cassini-headers@main
  xspec: +cuda +debug +internal-pmix fabrics=cma,ofi
    ,xpmem schedulers=slurm +cray-xpmem
```

Fig. 2: Specifying a custom developer version of OpenMPI

By default Cray-MPICH is installed, and this new feaature makes it possible to override the `depends` section to change drivers, or override the `mpi` spec entirely as well as providing custom versions of any or all of the dependencies

Currently the Slingshot sources must be tested carefully for compatibility since official release versions are not regularly made, so git SHA refs are used to identify versions used in Stackinator defaults. As the recipes become more widely used and tested, concrete versions will be chosen for stable uenv deployment.

B. Deployment: CI/CD

Alps has five node node types described in Table I, on top of which CSCS partitions the system into vClusters[1] [1], where each is customised for a tenant. As such, the version of SLURM, mounted file systems and software installed in the OS image (including libfabric) can vary between vClusters.

The deployment of uenv in this environment means managing the versions of uenv available to users on vClusters of Alps. We use a CI/CD pipeline that builds, tests and stores uenv images in a self-hosted container registry. The design of the deployment pipeline and tools had to accommodate the following requirements:

Not all uenv are provided for every micro-architecture on every cluster. Instead, uenv are "deployed" to specific cluster+micro-architecture combinations where they are required by the tenant and can be supported by the appropriate team at CSCS or a partner organisation. For example, the ICON uenv – for building and running the ICON atmospheric model – is deployed on only two systems where it is supported: the *gh200* nodes on Santis (the climate and weather platform system) and the *a100* nodes on Balfrin (the Swiss weather service test and development cluster).

Uenv images deployed to a vCluster are built on the target vCluster. In their first two years in production, uenv have proven to be portable between vClusters with the same microarchitecture, and we continue to further reduce the dependence on the underlying OS image. However, to deploy a uenv to a vCluster we still require that it be built on a compute node of the target vCluster to avoid the need for cross compilation, and to ensure that dependencies like libfabric and SLURM are correctly handled.

Uenv are versioned, each version can be have multiple minor updates, and all versions/update combinations are available. Each uenv has a name and version, for example, the `prgenv-gnu` uenv is versioned using the `YY.MM` format: `prgenv-gnu/24.11` was released in November 2024. Minor updates and fixes can be applied, and are tracked using `tags`. Using the `prgenv-gnu` example, the initial release had the tag `v1`, and the `v2` release provided exactly the same software, with the addition of the `netcdf` packages.

Our aim is to provide these uenv to users in a way that lets them upgrade and roll back when it is appropriate for them. This requires deploying unique uenv artifacts for each combination of system, micro-architecture, name, version and tag. This information is captured in the full description of a uenv – the uenv *label* – of the form `name/version:tag@system%uarch`.

The remaionder of this section will first describe the three infrastructure components – the GitHub repository for uenv recipes, the CI/CD service, and the container registry – followed by a more detailed description of the pipeline.

INFRASTRUCTURE: GITHUB RECIPE REPOSITORY

The uenv recipes and information required to configure the pipeline to build and test them are managed in a GitHub repository[2]. Recipes for the uenv are managed in sub-directories as illustrated in Fig. 3. Note how the `prgenv-gnu` uenv provides two versions (`24.7` and `24.11`), with microarchitecture specific recipes, while the `linaro-forge` uenv has a single recipe for each version because the same recipe can be used to deploy on any microarchitecture.

Information required to build and test uenv recipes on target system/uarch combinations is stored in a YAML file in the root, as illustrated in Fig. 4. Each system is described, with an entry for each microarchitecture provided by the system that describes which SLURM partition to use, and additional environment variables used by SLURM and FirecREST [3]. For each uenv version, the recipe to use for each microarchitecture is provided – note how the same recipe can be used to target multiple architectures.

INFRASTRUCTURE: CI/CD MIDDLEWARE

[1] Versatile software defined cluster

[2] github.com/eth-cscs/alps-uenv

uarch	type	blades	nodes	CPU sockets	GPU devices
gh200	NVIDIA GH200	1,344	2,688	10,752	10,752
zen2	AMD Rome	256	1,024	2,048	–
a100	NVIDIA A100	72	144	144	576
mi300	AMD MI300A	64	128	512	512
mi200	AMD MI250x	12	24	24	96
	TOTAL	1,748	3,880	13,480	11,936

tenant	vCluster	uarch
ML	Clariden	gh200
ML	Bristen	a100
HPC	Daint	gh200
HPC	Eiger	zen2
Climate	Santis	gh200
MetoSwiss	Balfrin	a100 and zen2

TABLE I: Alps node types and their specifications (left), and examples of vClusters provided to tenants (right).

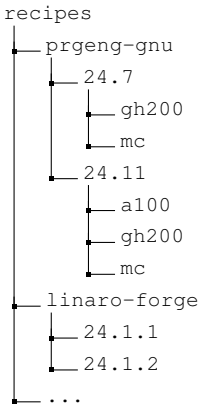

Fig. 3: The directory layout of uenv recipes in the GitHub repository. The `prgenv-gnu` recipes are specialized on target node type, while the `linaro-forge` has a single recipe for each version.

CSCS developed a CI/CD service for software projects hosted on GitHub, GitLab and BitBucket. The service provides a simple interface for CSCS users to register a new project at CSCS, configure a build pipeline using a YAML file in the repository, build and test on all Alps vClusters, and display the results. The aim was to provide access to HPC hardware and schedulers to test scientific applications on HPC systems, while integrating with GitHub similarly to existing services like Travis and Circle CI.

The CI/CD service is used to implement the pipeline for building and deploying uenv images. A new CI/CD project was created, and configured so that:

- staff responsible for uenv are white-listed to start build and test pipelines;
- the project was granted access to all systems where CSCS deploys uenv.

INFRASTRUCTURE: CONTAINER REGISTRY

Artifacts generated by the pipeline need to be stored so that they can be queried and accessed by users. To these ends, we use a container registry provided by a self-hosted JFrog[3] that is visible on the CSCS network. There are two artifacts generated by every pipeline that are managed in the registry:

- **uenv SquashFS images**: in the range of 300 MB – 10 GB in size.

[3] jfrog.com

```yaml
clusters:
  daint:
    targets:
      -
        uarch: 'gh200'
        partition: 'normal'
        variables:
          F7T_URL: "https://api.cscs.ch/hpc/
    firecrest/v1"
    runner: f7t
  eiger:
    targets:
      -
        uarch: 'zen2'
        partition: 'normal'
        variables:
          SLURM_CONSTRAINT: 'mc'
          F7T_URL: "https://api.cscs.ch/hpc/
    firecrest/v1"
uenv:
  prgenv-gnu:
    "24.7":
      recipes:
        zen2: 24.7/mc
        zen3: 24.7/mc
        gh200: 24.7/gh200
    "24.11":
      recipes:
        zen2: 24.11/mc
        zen3: 24.11/mc
        gh200: 24.11/gh200
        a100: 24.11/a100
  linaro-forge:
    "24.1.2":
      recipes:
        zen2: "24.1.2"
        gh200: "24.1.2"
      mount: "/user-tools"
    "24.1.1":
      recipes:
        zen2: "24.1.1"
        gh200: "24.1.1"
      mount: "/user-tools"
```

Fig. 4: Part of the configuration in the GitHub uenv recipe repository. The `clusters` field describes the target clusters, including which microarchitectures are available on each cluster, and how to access them The `uenv` field describes the uenv in the repository, specifically the uenv, their versions, and which microarchitectures they can be deployed to.

- **meta data**: a single tar ball with information that includes ReFrame tests, the input recipe, and build information. Less than 10 KB in size.

Oras[4] is a command line tool for managing file types in OCI-compliant container registries. The pipeline and command line tool (see Section IV-C) use Oras to query, push and pull uenv and meta data with the registry, see Fig. 5.

THE DEPLOYMENT PIPELINE

The pipeline is triggered when a whitelisted user makes a formatted comment on a pull request, for example:

```
cscs-ci run alps;system=daint;uarch=gh200;uenv=cp2k
    :2025.1
```

The `cscs-ci run alps;` command triggers the pipeline, and the remaining part of the argument is a list of variables that describe which image to build and where, that is forwarded to the first stage of the pipeline.

The trigger specifies which uenv to build, because automatically detecting what to build is impractical, and we need to only build specific images that are affected by the pull request. In the first stage, the variables and information in the configuration file in Fig. 4 are used to dynamically generate the specific pipeline that will build and test the uenv on the target system.

```
repo=jfrog.svc.cscs.ch/uenv
outpath_path=build/clariden/gh200/prgenv-gnu
    /24.11:1716820670
# push the squashfs image to the container registry
oras push $repo/$output_path \
    --artifact-type application/x-squashfs \
    store.squashfs
# attach the metadat image to the image
oras attach $repo/$output_path \
    --artifact-type uenv/meta \
    ./meta
```

Fig. 5: Examples of the Oras commands at the end of the build stage that push a SquashFS image and attach the meta data path to the image. The image is pushed to the `build` namespace, and is tagged with the CI job id `1716820670`.

The scripts that perform the build and test stages are in a separate repository[5], that is used by the uenv build service (see below), and the by user-communities to create their own pipelines (see Section VII-C)

If the build stage succeeds, the SquashFS image and the meta data are pushed to the local registry `jfrog.svc.cscs.ch/uenv`. The images are pushed to a `build` namespace, namely `jfrog.svc.cscs.ch/uenv/build/$label` where `label` has the format `system/uarch/name/version:tag` and the `tag` is is the unique id of the CI/CD job that built it. Examples of the Oras commands used to perform this are illustrated in Fig. 5. Similar commands are used to download meta data and SquashFS images in the test stage, and also by the uenv CLI tool in Section IV-C.

[4] `oras.land`

[5] `github.com/eth-cscs/uenv-pipeline`

When an image is ready for deployment, it is copied from the `build` namespace to the `deploy` namespace using the uenv CLI tool:

```
uenv image copy build::vasp:1631426005 \
                deploy::vasp/v6.5.0:v1@daint
```

Note that the deployment step is manual because CSCS staff often want to perform additional tests, or give the image to key users to test, before making the final decision to deploy.

THE UENV BUILD SERVICE

Some advanced users started building uenv images for themselves or their research groups, and CSCS staff were also providing one-off uenv images in response to individual user requests. We wanted to make it easy to build and provide "unsupported" uenv for these use cases.

A build command was added to the uenv CLI, that will build an image using the pipeline from a user-provided recipe from anywhere on the CSCS network:

```
> uenv build myapp/v1.2@daint%gh200 ./myapp-recipe
Log         : https://cicd-ext-mw.cscs.ch/ci/uenv/
build?image=cu3upvpoag1s73eo3n80-3690753405420143
Status      : submitted

Destination
Registry    : jfrog.svc.cscs.ch/uenv
Namespace   : service
Label       : myapp/v1.2:1626811672@daint%gh200
```

where `./myapp-recipe` is a path containing the recipe.

The "Log" URL links to a web site with live output of the build process. Once the image has been built, the image can be pulled from the container registry:

```
uenv image pull service::myapp/v1.2:1626811672@daint
```

Note that the images are pushed to the `service` namespace, which is separate from the `build` and `deploy` namespaces used by the main deployment pipeline. Images in the service namespace are available to all users (for ease of sharing), and are subject to more aggressive cleanup policies.

C. User Experience: the uenv CLI

A command line tool called *uenv* is the main interface for uenv users, alongside the SLURM plugin which is documented in Section IV-D.

The uenv tool and SLURM plugin are written in C++, with a library that is used by both, in an open-source GitHub repository[6]. The uenv command line tool is a statically-linked executable, and is bundled with the SLURM plugin in a single RPM, for ease of installation.

IMAGE MANAGEMENT

In Section IV-B the CI/CD pipeline that deploys uenv images to a container registry was described. Users need to download images to local storage before they can use them on a cluster. The uenv CLI provides a suite of functionality through the `uenv image` command for querying, downloading, and other image management tasks, illustrated in Fig. 7.

[6] `github.com/eth-cscs/uenv2`

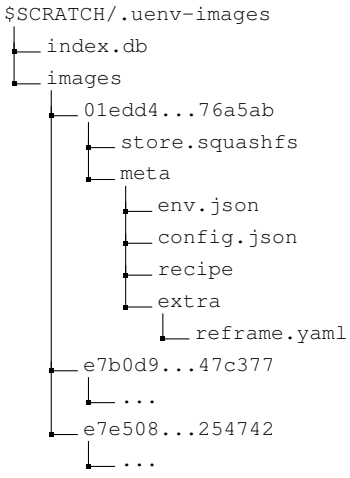

Fig. 6: The directory layout of a uenv *repository*.

There are two locations where uenv images are stored:

- *remote*: a *registry* is a container registry from which users can pull images to local storage;
- *local*: a *repository* is a directory on a local file system, from which uenv can be run.

Local storage for uenv images is a *repository* – a directory with an SQLite database at the root, and uenv images stored in an images sub-directory. The database `index.db` at the root of the repository allows for mapping the hash of each uenv to meta data like name, version, micro-architecture and target cluster. Each image is stored in a path that matches its hash, with the following information:

- `store.squashfs`: the SquashFS file;
- `env.json`: information about the mount point, uenv description, and view descriptions (their name, and the environment variable updates that they aplly).
- `recipe`: the original recipe used to build the image;
- `extra/reframe.yaml`: description of the testable features the uenv implements, and how to configure the environment to test them.

The uenv CLI tool uses the domain-specific language `name/version:tag@system%uarch` to describe uenv.

Users can query images in the registries and repositories using the `uenv image find` and `uenv images ls` commands respectively, for example the following will search for matching images in the users local repository:

```
uenv image ls prgenv-gnu/24.11:v1@daint%gh200
# show all versions or prgenv-gnu
uenv image ls prgenv-gnu
# show all versions or prgenv-gnu for santis
uenv image ls prgenv-gnu@santis
# show all images deployed for daint
uenv image ls @daint
# show all images deployed for gh200
uenv image ls %gh200
```

The `uenv image pull` command is used to pull an image from a remote registry. The following example searches for all vasp images built for the system Daint, pulls one of them, then uses the `image ls` command to show the downloaded image in the local repository:

```
$ uenv image find vasp@daint
uenv            arch   id                size(MB)
vasp/v6.4.3:v1  gh200  86edce79074e3478  7,963
vasp/v6.4.3:v2  gh200  d4753368f8b2baca  7,954
vasp/v6.5.0:v1  gh200  b7b097cb03d36451  5,709
$ uenv image pull vasp/v6.4.3:v2@daint
pulling d4753368f8b2baca 100.00% ------ 7954/7954
updating vasp/v6.4.3:v2@daint%gh200
$ uenv image ls vasp@daint
uenv            arch   id                size(MB)
vasp/v6.4.3:v2  gh200  d4753368f8b2baca  7,954
vasp/v6.5.0:v1  gh200  b7b097cb03d36451  5,709
```

It is possible add and remove images from a local repository using the `uenv image add` and `uenv image rm` commands respectively.

Copying and deleting images in a registry using `uenv image copy` and `uenv image delete` respectively requires permission, which is restricted to the team at CSCS responsible for image deployment. The `image copy` command is used to deploy by copying images from the *build* namespace (where the CI/CD pipeline pushes them after), to the *deploy* namespace:

```
# deploy from the build namespace
uenv image copy --token=$HOME/.ssh/jfrog-token \
                build::icon/25.2:1656679281 \
                deploy::icon/25.2:v2@daint
# deploy the image to a different cluster
uenv image copy --token=$HOME/.ssh/jfrog-token \
                deploy::icon/25.2:v2@daint \
                deploy::icon/25.2:v2@santis
```

Note that only one SquashFS image is stored after the two copies in the example above – each copy creates a new label that is attached to the original SquashFS image.

Loading uenv

Before describing how users can load uenv using the uenv CLI, we first define what it means to load a uenv environment. Loading a uenv provides the software in the SquashFS by mounting the squashfs image and setting environment variables.

The uenv CLI uses the `squashfs-mount` command line utility – a small `setuid` executable that creates a new mount namespace, mounts the SquashFS file through `libmount`, drops privileges and executes a given command. The following example starts a bash shell with the Squashfs files `img1.sqfs` and `img2.sqfs` are mounted at `/mnt1` and `/mnt2` respectively:

```
squashfs-mount img1.sqfs:/mnt2 \
               img2.sqfs:/mnt1 bash
```

The utility is open source, available on GitHub[7], and includes RPMs for installation on Cray EX.

The second part of loading an environment is to set any relevant environment variables. Each uenv image can provide any number of *views*, which are defined defined in the `meta/env.json` file in the SquashFS file. Each view is a list of environment varaiables and modifications to make to them,

[7] github.com/eth-cscs/squashfs-mount

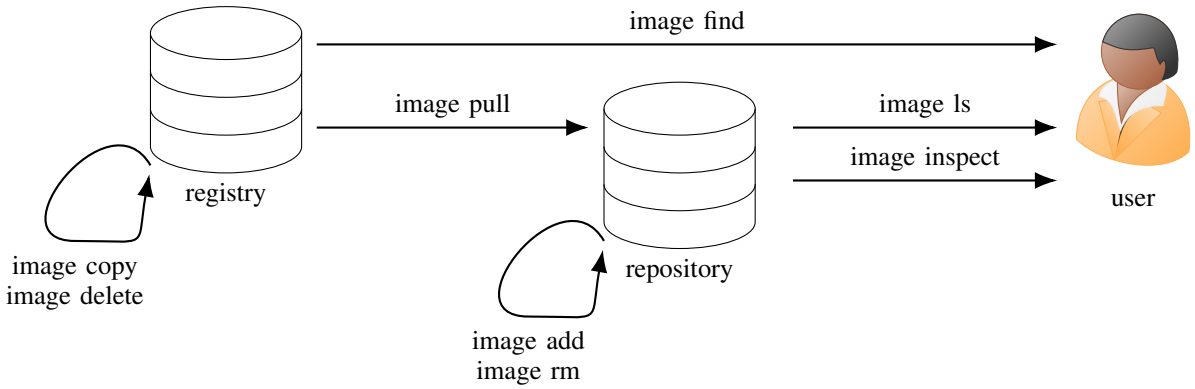

Fig. 7: The uenv image commands are used to manage uenv images in registries and repositories, and to query information about uenv images.

which are applied to the environment variables set in the calling environment.

There are two categories of environment variables:

- *Scalar* are environment variables with a single value, for example `CUDA_HOME`, `BOOST_ROOT`, and `TERM`. Each scalar variable in a view is a key-value pair, where the key is the variable name, and value is a string or `null`. If the value is a string the variable is set (or overriden) to the new value, and if it is `null` the variable is unset.
- *Prefix* are environment variables that represent a colon-separated list of paths where the order of paths is significant, for example `PATH`, `LD_LIBRARY_PATH`, and `PKG_CONFIG_PATH`. Prefix variables are representated as an ordered set of *updates*, where each update is one of *set*, *prepend*, *append*, *unset*.

There are are two views that are automatically generated by Stackinator when the view is built:

- `modules`: prepends the module path inside the the uenv to the `MODULE_PATH` environment variable, effectively making modules for packages in the uenv visible.
- `spack`: sets environment variables that: point to the Spack configuration for all packages installed inside the uenv; provide the url for the Spack repository; and the branch/-commit of Spack that was used. Together these variables provide all information required by users to use Spack to build their own software using the packages inside the uenv.

Recipe authors can define additional views, that configure the environment similarly to loading a set of modules. For example, the GROMACS uenv images provide three additional views:

- `gromacs`: set all environment variables so that the GROMACS executable is in the path and ready to use.
- `plumed`: set all environment variables so that a version of GROMACS with the popular PLUMED plugin is in the path and ready to use.
- `develop`: set all environment variables so that all of the libraries, compilers and tools used to build GROMACS are ready to use, without GROMACS, for users who want to build a custom version of GROMACS.

Using uenv

The uenv CLI provides two methods for using uenv images:

- `uenv run`: runs a command in the uenv environment before returning to the calling shell.
- `uenv start`: starts a shell with the uenv environment loaded, for interactive use.

The `uenv run` command is the most flexible, because it can be used interactively and in scripts including sbatch scripts. The examples below show how it can be used to "wrap" individual calls in an environment:

```
# configure a build using cmake, then build in a
    separate call
uenv run --view=default prgenv-gnu/24.11:v1 -- \
    cmake -DUSE_GPU=cuda -DUSE_MPI=on ..
uenv run --view=default prgenv-gnu/24.11:v1 -- \
    make -j64

# use a scientific application provided by a uenv...
# then run a visualisation script using another uenv
uenv run --view=gromacs gromacs/2024:v1 -- gmx_mpi
uenv run --view=default paraview/2024:v1 -- \
    ./post-process.sh
```

In practice there are no performance overheads from wrapping each launch using `uenv run` – indeed it is much faster than loading and swapping modules then executing software that is installed on a parallel filesystem. Even when a SquashFS image is stored on Lustre, it is faster and more reliable than accessing the equivalent software stack installed directly on Lustre, because only one file is accessed, and SquashFS efficiently caches files and meta data in memory.

The `uenv start` command is for interactive use cases, for example building software; testing and development; using debuggers and profilers; and using Python or Julia REPLs. In the next example, the `prgenv-gnu` uenv image is used to create and test a PyTorch virtual environment on a Grace-Hopper node. The version of Python in the uenv is available through the `default` view:

```
# now we are in a shell with the uenv loaded
uenv start --view=default prgenv-gnu/24.11:v1

python -m venv .env
source ./.env/bin/activate
pip install torch torchvision --index-url \
    https://download.pytorch.org/whl/cu126

# start a Python repl to test torch
python

# end the session: returns to the calling shell
exit
```

Lessons Learnt

The first version of uenv was written in Python, and was able to modify the calling environment similarly to modules. For example, the following command would set environment variables for a uenv that was running:

```
uenv view develop
```

A full rewrite of uenv was conducted, based on the lessons learnt from this first version.

The first lesson, which is somewhat subjective, is that Python was not an ideal language for deploying to production for the following reasons:

- the call to Python was designed to use Python 3.6 installed as part of SUSE, and had to be isolated;
- the project was deployed as a set of directories containing the Python implementating
- language features like the lack of type safety and exception-based error handling made implementing a robust, error-free code difficult as the size of the implementation increased.

This was simplified greatly by installing a single statically-linked C++ executable, written in C++20 to take advantage of modern language features for error handling, filesystem operations, and sanitizers.

The second lesson was that being able to modify the calling environment over-complicated the implementation, and enabled some bad practices.

In order to modify the calling environment, the `uenv` command was bash function that forwarded the arguments to the Python implementation: `echo "$($UENV_CMD $flags "$@")"`, where `UENV_CMD` is the path of the implementation. The implementation would process the arguments, and print a series of shell commands to stdout, which the calling bash function would then exec. For example, the command `uenv view default` might echo the following to stdout:

```
export PATH=/user-environment/env/default/bin:$PATH
export LD_LIBRARY_PATH=/user-environment/env/default
    /lib:$PATH
export CUDA_HOME=/user-environment/env/default
```

The wrapper-based implementation made it difficult to debug, required further wrappers inside calls to `uenv start` and `uenv run`, and made support for different shells difficult – the original implementation only supported bash. The new implementation only allows setting views when the uenv is loaded, which is performed by creating a new `const char* environ` array that defines the new environment, and forwarding this to `execvpe`. This has the benefits:

- the implementation is significantly simplified;
- it is shell agnostic - there is no bash-specific code anywhere in the implementation.

The final benefit is more subtle: specifying the target environment up front is declarative. User tickets are easier to debug because the views they are loading are explicitly listed as flags to the uenv call, and all environment modifications are captured in the logs of the `uenv start`, `uenv run` or SLURM `srun --uenv=? --view=?` calls.

D. User Experience: SLURM

The SLURM plugin integrates support for mounting uenv images and configuring the environment. A default uenv can be set using `#SBATCH --uenv` that provides the uenv inside the sbatch script, and loads it by default for every srun call. The uenv can also be specified using the `srun --uenv`, for example:

```
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=4
#SBATCH --gpus-per-task=1
#SBATCH --uenv=prgenv-gnu/24.11
# this view adds all software to PATH
#SBATCH --view=default

# use sqlite3 provided by prgenv-gnu
query="SELECT config FROM materials LIMIT 1;"
material=$(sqlite3 materials.db "$QUERY")

# run with prgenv-gnu on each compute node
srun ./test-material.sh $material

# use a different uenv to generated images
srun --uenv=paraview/5.13.0:rc1 ./generate-image.sh
```

V. CPE in a Container

CSCS is phasing out using CPE as an underlying layer on top of which we build software for users, and we now recommend that users use either uenv or containers for building and running applications and workflows, for the reasons described in Section III. However, we want to continue providing CPE in a way that meets the objectives in Section II.

Containers can meet the objectives, because container images can be built and modified by staff and deployment pipelines without root permission; can be pulled and run without modifying the underlying cluster; and users are not forced to upgrade when a new version is released.

At CSCS we have been testing monolithic containers that contain the full CPE, provided by HPE, since 2021. These containers had the downside of being very large, and were pre-built which minimised the opportunities customization for CSCS' specific needs.

More recently, in 2024, HPE have started to release RPMs for CPE components in a repository that HPE customers can

access, and providing recipes for building bespoke CPE containers[8]. In this section we will provide detailed instructions on how CSCS builds CPE containers, and links to repositories with the Containerfiles, followed by a discussion about how the containers are deployed and used on Alps.

A. Creating CPE containers

To access the RPM repository one needs an HPE Passport account, and generate a token[9]. Once we have an account with HPE, generated an access token and settled on the operating system, we can set up the package manager to pull packages from HPE's package repository, by pointing it to the URL `https://update1.linux.hpe.com/repo/cpe/<CPE-VER>/base/<OS>/<OS-VER>/<ARCH>`, where

- <*CPE-VER*> is the CPE version, e.g. 24.7, or 25.3
- <*OS*> is the operating system, i.e. `rhel` or `sle`
- <*OS-VER*> is the operating systems's version, e.g. `15.5`
- <*ARCH*> `x86_64` or `aarch64`, optionally needed (CPE-24.7 requires the flag, CPE-25.3 has an architecture agnostic repository setup)

When building CPE in a container with GPU support, it is necessary to install also NVIDIA CUDA, which is provided as an RPM repository by NVIDIA. A `Dockerfile` to build CPE in a container can be found on CSCS' GitHub[10]. The Dockerfile is parametrized to allow installing different versions of CPE, Every CPE release requires slightly varying packages to be installed, mostly the versions need to be adapted. We are building CPE container images for the GNU and Cray programming environment, with one container image for each. This allows us to load default modules when the container is instantiated, such that the programming environment is directly available without, for example, typing `module load PrgEnv-gnu`.

The default list of modules loaded at startup is also parametrized in the Dockerfile, with the defaults chosen to initialise an environment that is ready to compile applications for the target architecture. For example, this is the module list for `cpe-gnu` on the Grace Hopper nodes is:

- craype-arm-grace
- craype-network-ofi
- craype
- xpmem
- PrgEnv-gnu
- cray-mpich
- cuda
- craype-accell-nvidia90

Note that the craype-accell-nvidia90 module is loaded, unlike the CPE where users have to explicitly load it, which reduced the number of user tickets complaining about the "GPU_SUPPORT_ENABLED is requested, but GTL library is not linked" errors.

[8] cpe.ext.hpe.com/docs/latest/install/installation-guidance-container.html

[9] cpe.ext.hpe.com/docs/latest/install/token-authed-repo.html

[10] github.com/eth-cscs/cpe-containers

Another advantage of building one container image per programming environment is the reduced image size of the image. Admittedly the sum of `cpe-gnu` and `cpe-cray` container images is larger than building both in the same container, because some packages like cuda-toolkit, are installed in each container image. However a smaller size per programming environment will have long term advantages, where users typically are only interested in one environment, and will build their container image on top of the programming environment of their interest.

B. Deploying CPE containers

In the repository `eth-cscs/cpe-containers` GitHub repository we have a CI/CD pipeline to build, test and deploy the container images for the different CPE versions and programming environments. The pipeline builds a new container image when a PR is opened, which is also tested on the target system. On a merge to the `main` branch the container image is then deployed to the CSCS container registry. Once the image is deployed to the registry, it can be pulled on the target system, and made available for users.

C. using CPE containers

Alps uses forks of the enroot container runtime and Pyxis SLURM plugin to provide an integrated container solution. With the SLURM integration from Pyxis, starting a job with the CPE is simple, for example starting an interactive shell on a compute node with a specific release of CPE:

```
srun --environment=cpe-gnu-24.7 --pty bash
```

The container environment is transparent to users, because the SLURM plugin mounts the scratch filesystem so that software built on scratch will persist between invocations. The instantiation of the container is equivalent to `module load PrgEnv-gnu`.

VI. Use Cases

A. JupyterHub

CSCS provides a JupyterHub web portal for users to access Alps through Jupyter notebooks, which requires the Jupyter software stack installed on the cluster where the session will run. The approach taken by CSCS is to create a simple `jupyter` uenv that provides only Jupyter and its requirements. This uenv is automatically mounted by the JuputerHub service, alongside a uenv selected by the user, which provides the scientific software required.

User feedback is that the environment feels more responsive than when it was installed on a shared filesystem, which is a result of using SquashFS.

B. Weather Service Production

CSCS hosts the operational cluster of The Swiss Weather Service (MeteoSwiss) on Alps – a system with GPU nodes (4 × A100 GPU per node) for the weather forecast, and CPU only nodes (2 AMD Xeon CPU per node) for the other tasks in the operational weather forecast pipeline. The MeteoSwiss

pipeline requires two distinct software stacks: an NVHPC stack for their Fortran+OpenACC ICON application; and a GNU toolchain for the other applications in the workflow. Both stacks are implemented in one uenv, that is mounted permanently at `/mch-environment`.

Due to their strict operational requirements, MCH are naturally very conservative and cautious. Using uenv has been beneficial because to test a new operational environment the CLI and SLURM plugin can be used to create a session with the new stack mounted on the current stack to perfectly replicate what the permanent deployment would look like. As we have gained confidence with this approach, we have been able to start splitting the MCH workflow into completely separate environments that are updated at different cadences – an approach that they would not have agreed to two years ago.

C. Optimizing Python installations with SquashFS

The uenv CLI and SLURM plugin support mounting multiple uenv in the same session – a feature developed for mounting tools like the debuggers and profilers at the same time as a programming environment or application. The SquashFS files do not need to be uenv, any SquashFS file can be mounted at an arbitrary location.

A common Python workflow is to use a uenv that provides Python, MPI and other libraries, then use pip to install additional packages in a virtual environment on the Lustre filesystem. For example the following installation of PyTorch:

```
uenv start prgenv-gnu/24.11:v1 --view=default
# create a working path on SCRATCH
mkdir $SCRATCH/env
cd $SCRATCH/env
# create and activate the empty venv
python -m venv ./.venv
source ./.venv/bin/activate
# install software in the virtual environment
pip install torch torchvision torchaudio \
    --index-url \
    https://download.pytorch.org/whl/cu126
```

Installing this on a Grace-Hopper node created over 22,000 inodes in the virtual environment, which leads to long initialisation times for Python scripts that load the modules therein.

The Python venv in `$SCRATCH/env/.venv` can be turned into a single SquashFS file that can be mounted over the original venv as followd:

```
# make a squashfs image of the venv
mksquashfs $SCRATCH/sqfs-demo/.venv \
    py_venv.squashfs -no-recovery -noappend
# start the uenv with the venv mounted
cd $SCRATCH/env
uenv start --view=default \
    prgenv-gnu/24.11:v1,$PWD/py_venv.squashfs:
    $SCRATCH/env/.venv
source .venv/bin/activate
```

Using SquashFS to reduce file system pressure can have a significant impact – users report up to 5× speedup for some Python workflows. Improving support for creating and using SquashFS images in common workflows will be a focus of future work.

VII. Discussion

A. The Objectives

Here we review the original objectives.

1) Provide optimised software: Deployment of key software like MPI, compilers and NVIDIA tools via uenv is on par with or better than CPE.

Our efforts to deploy libfabric, OpenMPI and NVIDIA libraries that are tuned for the system are ongoing, and we have provided them for early testing on Alps. Robust deployment of these components will require the release of Spack 1.0, and improved versioning and release management of the open sourced CXI, Cassini headers and libfabric from HPE.

2) Gitops deployment: The uenv recipes are managed in a GitHub repository, and the tools used by the deployment pipeline are managed in separate GitHub repositories.

The pipeline deploys uenv as discreted artifacts to a container registry that is accessible from the different clusters on Alps.

Finally, each uenv contains the original recipe and other meta data required to rebuild the image at a later date. Currently the process of rebuilding from a uenv image is quite manual, and is something that will be the focus of future work.

3) Decouple environments: New versions of uenv images can be deployed, without removing or modifying existing environments, so users are not affected and can upgrade at their own pace.

This paper did not cover how changes to the base OS that is installed on a cluster are tracked in a separate configuration repository. To date there has only been one breaking change on the system, where the libfabric installed as part of the Slingshot host software changed location after an upgrade. We were able to work around this issue without rebuilding uenv images, however we have the ability to rebuild images in such situations, and our ongoing efforts to install Slinghshot from source inside uenv will further migigate these risks.

Finally, uenv that provide the latest version of cray-mpich are deployed days after it is released, along with uenv and CPE containers that use the pre-release of cray-mpich 9.0, for early testing.

4) End to end responsibility for software support teams: This has been one of the main outcomes of this work – teams across CSCS are deploying full software stacks with very little central coordination, and with no interaction with system administrators. See Section VII-C for an example of how users have started deploying their own software stacks.

B. Maintenance overheads

Maintaining the Stackinator tool, the CI pipeline, and uenv tools is not free – a core team of two staff performs this work. One works 75% on the effort, and the other about 25%. CSCS staff actively contribute fixes of their own volition, and contribute to discussions about the architecture and design of the system. Now that the system is mature, the effort required to maintain it is less than during its development. Overall, the amount of effort required to set up the deployment pipeline and

tooling was greater than what was required for the system that it replaced, however this system is more powerful and flexible.

One side effect of this effort is that by decoupling deployment of software environments, a lot more software is being deployed than on the old Piz Daint Cray XC system. The total amount of effort spent on building and testing software is roughly the same, however more complicated software stacks are being deployed to half a dozen clusters and for a wider range of domains than before. This leads us to a key observation from CSCS efforts to support multi-tenancy through increased automation and flexibility: the effort required to perform a task like deploying a cluster, building a software stack or adding a dashboard goes down, but the total amount of work stays the same because the number of clusters, software stacks and dashboards will expand to fill capacity.

C. Empowering staff empowers users

One of the objectives was to give staff end-to-end responsibility for software deployed on the system (see II-D). This was achieved through enabling users to build fully self-contained uenv that include software that was previously installed by system administrators (e.g. `cray-mpich`). The CI/CD pipeline makes maintaining the recipe, and deploying to different clusters relatively straightforward.

The climate and weather community have started to maintain their own software stacks, by creating recipe repositories and pipelines using the same CI/CD infrastructure. The registry contains uenv for the production MeteoSwiss environment, and for the software environments used on the climate and weather platform.

The significance of this can't be overstated – in the past CSCS maintained these challenging software stacks, and the community was often frustrated with forced upgrades or having to iterate with CSCS to define their environment. It was one of the key aims of the move to multi-tenancy and use-case specific clusters that tenants and communities would take more responsibility for software – however how this would manifest was not clear. The current solution was not part of a master-plan, instead it is a happy consequence of the user-first design to empower staff who don't have root.

D. Deploying outside CSCS

Much of what has been presented here is site-specific: the on-site container registry and ci/cd infrastructure would be different if this was implemented elsewhere. The processes and lessons are transferable, however what we have presented is not a turnkey solution.

The main software tools for building and using uenv are quite easy to get running locally. Stackinator can be used to build squashfs images, or even to install isolated solftware stacks in a directory on a shared file system. The uenv CLI tool and SLURM plugin have some site-specific customization related to:

- the container registry address and some quirks of dealing with its API;
- determining the name of the cluster on which the tool is being used;
- determining the username;
- and configuring credentials for interacting with the registry.

These are implemented in a separate C++ header and implementation file, which can be customised. Note that these features are only required for interacting with a remote registry: the tools will still function with a local repository. If there is interest from other sites, this could be refactored to make registry interaction more plug and play.

VIII. Future Work

The software tools – Stackinator, uenv CLI, SLURM plugin and the pipeline implementations – are quite mature and robust. Much of our focus will be on quality of life improvements and small feature requests.

We have seen significant performance improvements from using SquashFS, with the corresponding reduction in IOPs that benefits other users on the system. The use case of using SquashFS for user-installed software and datasets is promising, and is supported by uenv uenv tooling, as shown in Section VI-C. However, the workflow for doing this is ad-hoc. We will add support for "environment definitions" in YAML files, that provide a list of uenv and additional SquashFS images to mount, with additional environment variables to set.

Currently the tools can't be deployed as a turnkey solution at another site. Take, for example, the container registry support: this uses hard-coded address of the JFrog artifactory at CSCS. However, all interactions with the registry are generic and could be used with any OCI container registry, for example DockerHub or the Github Container Registry. If there is interest from other sites, we could make it easier to configure uenv for different sites with little effort.

References

[1] S. R. Alam, M. Gila, M. Klein, M. Martinasso, and T. C. Schulthess. Versatile software-defined hpc and cloud clusters on alps supercomputer for diverse workflows. *The International Journal of High Performance Computing Applications*, 0(0), 0.

[2] J. Coles, B. D. Cumming, T.-I. Manitaras, J.-G. Piccinali, , S. Pintarelli, and H. Stoppels. Deploying alternative user environments on alps. *CUG 2023*, 0(0), 2023.

[3] F. A. Cruz and M. Martinasso. Firecrest: Restful API on cray XC systems. *CoRR*, abs/1911.13160, 2019.

[4] P. Forai, K. Hoste, G. Peretti-Pezzi, and B. Bode. Making scientific software installation reproducible on cray systems using easybuild. In *Proceedings of the Cray Users Group Meeting (CUG2016)*, 2016.

[5] T. Gamblin, M. P. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and W. S. Futral. The Spack Package Manager: Bringing order to HPC software chaos. In *Supercomputing 2015 (SC'15)*, Austin, Texas, November 15-20 2015.

[6] P. Jahan Elahi, C. Di Pietrantonio, M. De La Pierre, and D. Kumar Deeptimahanti. Automating software stack deployment on an hpe cray ex supercomputer. In *Proceedings of the Cray Users Group Meeting (CUG2023)*, 2023.

[7] S. Sugiyama and D. Wallace. Cray dvs: Data virtualization service. In *Proceedings of the Cray Users Group Meeting (CUG2008)*, 2008.