Automating Software Stack Deployment on an HPE Cray EX Supercomputer

Pascal Jahan Elahi
Pawsey Supercomputing Research Centre
Kensington, WA, Australia
pascal.elahi@pawsey.org.au

Cristian Di Pietrantonio
Pawsey Supercomputing Research Centre
Kensington, WA, Australia
cristian.dipietrantonio@pawsey.org.au

Marco De La Pierre
Pawsey Supercomputing Research Centre
Kensington, WA, Australia
marco.delapierre@pawsey.org.au

Deva Kumar Deeptimahanti
Pawsey Supercomputing Research Centre
Kensington, WA, Australia
deva.deeptimahanti@pawsey.org.au

***Abstract*— The complexity and diversity of scientific software, in conjunction with a desire for reproducibility, led to the development of package managers such as Spack and EasyBuild, with the purpose of compiling and installing optimised software on supercomputers. In this paper, we present how Pawsey leverages such tools to deploy the system-wide software stack. Two aspects of Pawsey software stack deployment are discussed: the first comprises organisation, accessibility, interoperability with the HPE Cray EX environment, and the choice of technologies such as containers, derived from a set of policies and requirements; the second is the (almost) automated, self-contained, deployment process using Spack and Bash scripts. This process clones a specific version of Spack, configures it, runs it to build the software stack using environments, deploys Singularity Registry HPC to setup desired containers-as-modules and then generates bespoke module files. The deployment is tested using a ReFrame framework. Meeting the requirements of our user base necessitated patching Spack, writing new Spack recipes, patching existing recipes and/or source code of software to properly build within the Cray Programming Environment. The whole Spack configuration at Pawsey is made publicly accessible on GitHub, for the benefit of the broader HPC community.**

Keywords*— *HPC, software installation, automation, Spack, containers, ReFrame, regression tests

I. Introduction

A supercomputing software stack comprises of libraries and applications that implement or support scientific and industrial research. Some of the software is free and open source, some proprietary, and some may have restrictive licenses. Supercomputing centres spend considerable time and effort in providing a performant software stack for their users that satisfies their requirements. This often entails installing multiple versions of each software for various architectures, compilers, and dependent libraries. Furthermore, the software stack needs to be rebuilt for each vendor-specific system update such as an OS (Operating System) update or Cray Programming Environment (CPE) update.

In this paper, we present our automated process of installing a full software stack and how it can handle updates to the underlying HPC system. We first present our software policies and motivation behind the design of the Setonix software stack. We then describe the organisation of the Pawsey Supercomputing Research Centre's [1] software stack on the top of the HPE-Cray EX environment, including key aspects of the Spack instance and the rationale for targeted use of containers. Next, we outline the key features of the automated deployment procedure that enables the full installation of the software stack on our supercomputer. To provide a measure of effectiveness for the proposed approach, we discuss the pros and cons of the two full stack deployments we have already made with it on Setonix. Finally, we summarise ongoing and future works and make our conclusive remarks.

II. Background

Due to rapidly evolving HPC world, most HPC users lack the knowledge to configure their software to best take advantage of the latest hardware. Additionally, the same application or library used by multiple research groups can be installed once by HPC staff for all users, avoiding replication of software installation and effort. Therefore, the provision and maintenance of scientific software on a supercomputer by HPC experts crucially enable researchers to interact with a ready-to-use, highly optimised environment. The challenge for HPC centres in providing this environment is the complexity of scientific software, which makes deployment a time-consuming process, and prone to mistakes and unforeseen errors.

To facilitate this, Pawsey previously used an in-house automated system named Maali [2], which used a complex Bash script to automate the process and bash scripts as recipes. However, with the growing user base and increase in the number of application requests, it became very time-consuming to prepare recipes for each requested software package by our users. Additionally, system updates often broke the recipes, which then needed to be promptly fixed to reproduce the software stack.

Our approach required a rethink and revision, particularly considering the deployment of our new system, Setonix, Pawsey's latest HPE Cray EX supercomputer. Pawsey investigated the suitability of Spack [3, 4, 5], an open-source package manager developed by Lawrence Livermore National Laboratory, as a replacement. Our choice was motivated by the large and ever-growing number of pre-existing installation recipes, the large, active community base, easy-to-use interface,

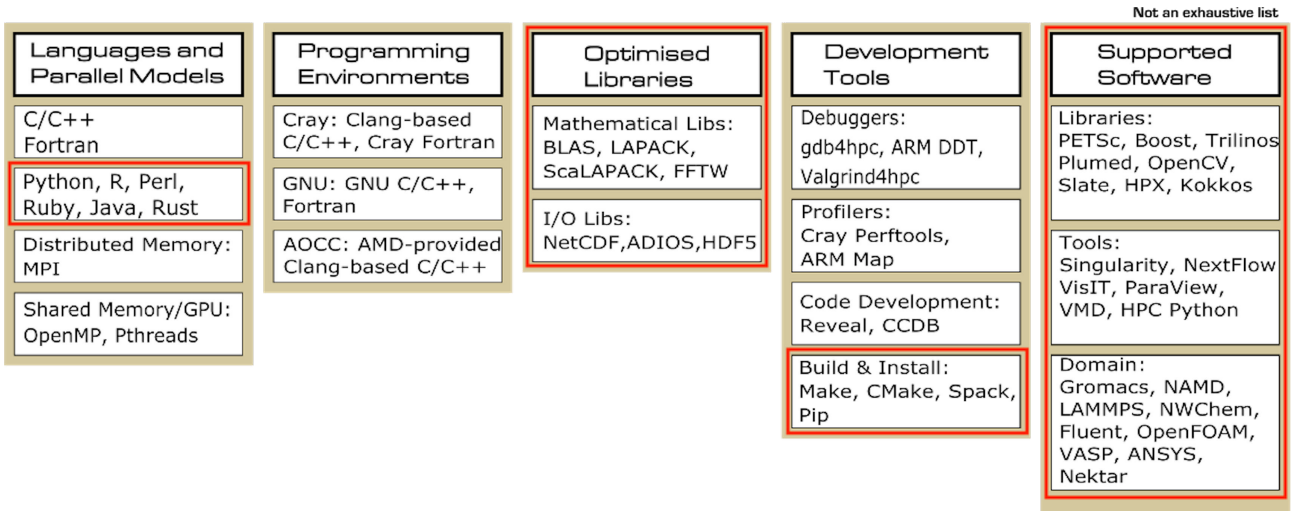

Fig. 1. Summary of available software on Setonix, here grouped by rough type. Software packages highlighted by a red frame are deployed via this automated process.

and the existence of a product roadmap to improve AMD support in collaboration with HPE. Additionally, the use of rpaths by Spack improves reproducibility. Initial trials by our Applications team on the Magnus, a Cray XC40 supercomputer were successful: the team was able to easily configure and install several software packages, although there were challenges installing some packages like Amber. Due to the positive results, Pawsey selected Spack as the package manager on Setonix, its latest HPE Cray EX supercomputer.

Although most software is best installed as bare-metal builds, some are best provided through containers, a notable example being bioinformatic software. To provide these software packages in a fashion like bare-metal builds produced by Spack, Pawsey uses the Singularity Registry HPC (SHPC, [6]), a publicly available utility to that can provide containers-as-modules.

To simplify the complex procedure of installing the entire software stack and improve reproducibility, the Pawsey Supercomputing Applications team developed a mostly automated software deployment process to provide a performant software stack on its latest HPE Cray EX supercomputer, Setonix. Implemented through a set of well-designed Bash scripts and several other well-known tools, the process is reproducible and understandable, runs in less than a day, and relies on other well-known tools like git.

The design decisions pertaining to the software stack's organisation and deployment is guided by our software policies [7], developed by the Supercomputing Applications team at Pawsey. This document captures the high-level software stack organisation, how to determine what the stack contains, and protocols for support, deployment, updates, and maintenance. Briefly, the main points related to how the design of the deployed software are:

- Software is organised in three-level of support: system-wide (fully supported and maintained by Pawsey), project-wide and user-specific (maintained by Pawsey users).
- The software stack consists of bare-metal installations and containers that are exposed to users through module files. Module files are organised in informative, user-friendly categories.
- Supported software package versions follow a general 3+1 rule, accounting for an implicit classification of one "legacy", one "stable" and one "latest" version. The stack can also contain several versions of a given piece of software to cover compatibility requirements of the user community, examples of which is Boost.

III. Software Stack Components and Organisation

An HPE Cray supercomputer comes with pre-installed, high-performance applications and libraries that complement the hardware and efficiently implement fundamental operations and tasks, such as multi-node communication primitives through MPICH and mathematical functions through Cray LibSci. Built on top of this low-level, system-wide stack, the Pawsey software stack provides users with optimized installations of relevant applications and libraries, and the means to easily install their own additional software, to design and run scientific computations. An example of software packages and what is installed through this automated process is presented in Figure 1.

A. High Level Organisation

There are five components making up the Pawsey software stack, each having a top-level directory in the installation prefix. First, the Spack package manager is installed within the installation prefix, within the Spack directory. Most applications

and libraries are compiled from source using Spack and they are placed under the software directory.

Not all software packages are built from source due to either having a high number of dependencies, suffers from a complicated installation procedure, or requirement for portability. For this reason, they are provided within containers that can be run using Singularity [8]. Containers are installed using the SHPC tool and placed under the `containers/` directory. We install software with a build system different than Spack, such as pre-built binaries, under the custom directory.

```
setonix/DATE_TAG/
├── containers/
│   ├── modules-long/
│   ├── openfoam-sif/
│   ├── sif/
│   └── views/
│       └── modules/
├── custom/
│   ├── modules/
│   └── software/
├── modules/
│   ├── zen2/
│   └── zen3/
├── pawsey/
│   ├── modules/
│   └── software/
├── pawsey-spack-config/
├── software/
│   ├── bin/
│   ├── cray-sles15-zen2/
│   ├── cray-sles15-zen3/
└── spack/
```

Fig. 2. High-level organisation of the *system-wide* software directory.

Finally, any software, independently of how it is installed, is made visible to users through Lmod modules and placed under the `modules/` directory. At present, modules have different paths depending on the type of build approach taken for the corresponding software (Spack, SHPC, or custom).

Pawsey identifies three software deployment levels: system-wide, project-wide, and user-wide. Popular applications and libraries accessible to all users are deployed system-wide by Pawsey staff as seen in Figure 2. Project-wide installations are meant to be used by members of a research group and no one else; usually a member of the group is appointed to manage such installations. Finally, there are a user's personal installations, an example is shown in Figure 3. All of them share the same instance of Spack and SHPC by clever use of its multi-level configuration lookup.

All the software on Setonix resides on a dedicated Lustre filesystem aptly named `/software/` with each deployment level having its own installation prefix. System-wide software deployment is installed under `/software/<system>/<date-tag>`. The `<system>` parameter is the target supercomputing system, accounting for the fact that the `/software/` filesystem may be mounted on multiple supercomputers in the future. The `<date-tag>` parameter is the deployment date and allows the coexistence of multiple deployments.

B. Spack Instance

Pawsey adopts the Spack software manager for optimized and reproducible deployments. A stable release of Spack is firstly customised and then installed in the system-wide software stack. The Spack version currently deployed on Setonix is 0.17.0. However, we already moved to Spack 0.19.x in the development branch of this work.

```
username/setonix/
├── containers/
│   ├── modules-long/
│   ├── sif/
│   └── views/
│       └── modules/
├── manual/
├── modules/
│   └── zen3/
├── python/
├── r/
├── shpc_registry/
├── software/
│   ├── bin/
│   ├── cray-sles15-zen3/
└── spack_repo/
```

Fig. 3. High-level organisation of the *user-specific* software directory.

Because Spack does not natively support the distinction between user and project installations, a wrapper script has been developed to use Spack with an alternative configuration directory enabling project-wide installations, see the file scripts/templates/spack_project.sh in the Pawsey Spack Config repository [9]. The user's Spack cache and configuration directory would normally reside in their home directory, which is often subject to strict quota limits. We moved the `~/.spack/` directory from the user home to within the versioned user-wide software stack prefix to avoid filesystem limitations and interference between different Spack deployments; see file `fixes/dot_spack.patch` in the Pawsey Spack Config repository.

Several package recipes have been fixed or adapted to work with the software environment present Setonix. Missing dependencies or incorrect dependency specs, additional source code patches, and integration with the Cray environment are the among the main reason to modify existing Spack recipes. Of notable mention are recipes for Amber and Charmpp. The former used to build dependencies within the Amber build process, missing the opportunity of using the Spack dependency mechanism. The latter did not properly recognise the Cray libfabric library. Finally, we created several for software packages in domains such as quantum computing (Qutip, Xanadu Pennylane, Xanadu Strawberry Fields), bioinformatics (tower-agent, tower-cli, nf-core tools), and radioastronomy (astropy, casacore, wsclean, vcstools). We contributed all these improvements back to the main Spack repository for the benefit

of the greater community. One of the authors is also a Spack maintainer for some of the Pawsey-created recipes.

Configuration files have been customised to ensure use of the Cray Programming Environment (CPE). Cray MPICH, libfabric and xpmem are exposed as external packages. Unfortunately, we have found Cray LibSci quite poor in terms or performance and reliability, with several bugs affecting the library. At the time of the first deployment, we made the decision not to use it. We will re-evaluate this decision periodically. Finally, the template file for module files has been customised using dedicated Jinja syntax, to partition them onto several categories, hence simplifying user navigation. Another template modification adds build information, including compiler, compiler flags and build variants. See the template file `systems/setonix/templates/modules/modulefile.lua` in the Pawsey Spack Config repository.

Spack version 0.19.x introduces a few options to personalise the behaviour of the concretizer. By default, Spack only compiles software for the CPU architecture it currently runs on. For heterogenous systems, like Setonix that hosts multiple CPU architectures, this new behaviour increases the complexity of software deployment processes. We removed this restriction by setting the option `host_compatible=False`. The new Spack reuse feature, that allows the concretizer to be more flexible when deciding whether to use an already-installed package, had to be turned off because resulting in unwanted behaviour.

For each CPE-provided compiler, there exists an entry in the `compilers.yaml` configuration file, specifying name and version, and executables for Fortran and C/C++ code. In addition, we instruct Spack to load related CPE environment modules needed for the compiler to work correctly. This is another important way in which the Cray environment takes part in our software stack. For every compiler, we also provide the debug and profiling variations, with all the relevant flags specified in the configuration file. All customised Spack configurations for Setonix can be found under the directory `systems/setonix/configs/` in the Pawsey Spack Config repository.

C. Organisation and Software Module Hierarchies

A scientific software stack has an inherent complexity due to compiling multiple instances of a package that arise from having different compilers[1], target CPU architectures, and API library implementations used in the code (for instance, MPI). This motivates the use of software hierarchies, where for instance installation paths reflecting the characteristics of a given build.

Software hierarchies are one of the key functionalities of the Lmod module system [10], and for this reason, we have selected it as the default one on Setonix. Notably, Cray provides a customised installation of Lmod, which among other things implements an un-documented custom way of handling hierarchies, in substitution for the standard Lmod way. On Setonix, an HPE Cray EX system, the code can be found at `/opt/cray/pe/admin-pe/lmod_scripts/lmodHierarchy.lua`. Each of the compiler, CPU architecture and MPI library modules scans the shell environment for a specific variable that sets additional module paths for that specific component; these paths are set (unset) when the component module is loaded (unloaded). For example, modules for a GCC compiler will look for the variable `LMOD_CUSTOM_COMPILER_GNU_8_0_PREFIX`, and so on.

On HPE Cray systems, switching between compilers is done through switching the Programming Environment (`PrgEnv-*`) modules because of the mechanism described above. We implement the process of switching between the software hierarchies of the Spack-built applications through a Pawsey provided module, `pawseyenv`, which sets all the necessary environment variables. This module file is automatically loaded at shell login for every user; the corresponding template file is `scripts/templates/pawseyenv.lua` in the Pawsey Spack Config repository.

Setonix hosts a variety of AMD CPU architectures, specifically Zen-2 for data mover nodes and Zen-3 for compute nodes. Consequently, we categorise software also based on the target architecture it is compiled for using `lscpu` at user login time to detect the CPU model. The model is saved in a variable and used to later form search paths for module files. The result is that on a data mover node, a user can only see software compiled for Zen-2 CPUS, which happens to be related to data movement. On a compute node which has Zen-3 CPUs, the entire software stack is visible.

Spack can add the MPI implementation to the software hierarchy. Only one MPI implementation is available on Setonix in the current CPE version of and, as such, is unused.

The resulting user access of software through Lmod module files is one in which the architecture hierarchy is static and set at shell login time and the compiler one is dynamic and dependent on the current active Programming Environment. To further simplify user navigation, applications have been divided onto multiple categories, as seen in Figure 4 in the case of the Zen-3 architecture.

We decided to make a major change in user experience by forcing module load commands to request both module name and module version. Although requiring extra typing, we believe this choice enforces a good practice that improves reproducibility of user workflows.

The meta-information provided by the `module whatis` command for Spack-built packages provides additional details around compiler, compiler build flags and Spack build variants.

D. Container Modules

Certain software categories are provided system-wide as containers, that can be executed by means of the Singularity runtime container engine. These containers are outside the software hierarchy of bare-metal builds. This includes bioinformatics packages, some older releases of OpenFoam (a computational fluid dynamics package), and a bespoke Python stack for HPC (which comprises all Python scientific packages

[1] Cray, GNU and AOCC are currently present on HPE Cray EX systems. Currently, our software stack is built using GCC within the GNU programming environment. Because the compiler is part of the hierarchy, however, multiple software stack builds, one for each compiler, can coexist on the system.

officially supported on Setonix). The benefits here are: time-effective deployment of domain-specific packages (bioinformatics); poor IO performance resolved using overlay filesystems mounted to the running containers (OpenFoam); and overall improved reproducibility and portability of workflows relying on articulated and conflict-prone dependency trees (Python-based).

```
modules/
├── zen2/
│   ├── aocc/
│   │   └── 3.2.0/
│   ├── cce/
│   │   └── 14.0.3/
│   └── gcc/
│       └── 12.1.0/
└── zen3/
    ├── aocc/
    │   └── 3.2.0/
    ├── cce/
    │   └── 14.0.3/
    └── gcc/
        └── 12.1.0/
            ├── applications/
            ├── astro-applications/
            ├── benchmarking/
            ├── bio-applications/
            ├── dependencies/
            ├── developer-tools/
            ├── libraries/
            ├── programming-languages/
            ├── python-packages/
            ├── utilities/
            └── visualisation/
```

Fig. 4. High-level organisation of the system-wide module directory for Spack builds. As an example, breakdown of the module categories is given in the case of `zen3/gcc/12.1.0`.

These containers are exposed to users as so-called container modules using Singularity Registry HPC [6] (SHPC), a publicly available utility to automate the deployment of containerised software as modules, to which some of these authors have made significant contributions in terms of concepts, feature-set and implementation[2]. For each containerised application, the module will provide a binary path containing a set of wrapper shell scripts, encapsulating the Singularity syntax required to execute a given set of executables relevant to that application. In this way, users of a package can load the corresponding module and execute its application binaries by using the same syntax required for the standard package build, thus effectively removing the usage barrier for containers.

Unlike the bare-metal builds, the containers module category is flat since containers do not depend on compiler. Here, the `module whatis` provides meta-information such as tool name and version, and container repository of origin, whereas `module help` outputs usage information, such as the available executables.

```
pawsey-spack-config/
├── fixes/
├── LICENSE
├── NOTES.md
├── README.md
├── repo/
├── scripts/
├── shpc_registry/
└── systems/
    ├── askapingest/
    │   └── configs/
    ├── garrawarla/
    │   └── configs/
    ├── setonix/
    │   ├── configs/
    │   │   ├── project/
    │   │   ├── site/
    │   │   └── spackuser/
    │   ├── environments/
    │   │   ├── env_apps/
    │   │   ├── env_astro/
    │   │   ├── env_bench/
    │   │   ├── env_bio/
    │   │   ├── env_devel/
    │   │   ├── env_io_libs/
    │   │   ├── env_langs/
    │   │   ├── env_num_libs/
    │   │   ├── env_python/
    │   │   ├── env_roms/
    │   │   ├── env_s3_clients/
    │   │   ├── env_utils/
    │   │   ├── env_vis/
    │   │   └── env_wrf/
    │   ├── settings.sh
    │   └── templates/
    └── topaz/
        ├── configs/
        └── environment_compchem/
```

Fig. 5. High-level organisation of the `pawsey-spack-config` repository.

IV. Automated Deployment

Deployment is the process of installing and maintaining scientific software, tools and libraries on the supercomputer, together with all the necessary mechanisms to make it easily accessible to users. Historically, software deployment at Pawsey was manual leading to a not easily reproducible, error-prone, and tedious process. Spack and SHPC abstract away much of this process, reducing the manual burden on system administrators. Pawsey deployment process employs these tools in a set of sophisticated Bash scripts containing all the necessary commands to deployment of a clearly defined software stack

[2] Notably, one of the Pawsey contributions targeted the automated and scalable discovery of executables within containers [13], to enable automation of the process that generates modules for very large collections of container images; a major outcome of this project was the release of container module recipes for the collection of 8000+ BioContainers, widely popular in bioinformatics, now available on demand on Setonix through this Software Stack Deployment.

according to our policies and design choices. These scripts are non-interactive, allowing the applications team to run them in a Slurm job, therefore achieving automation.

A. Repository Organisation and Tools

The software stack deployment procedure presented here is publicly available on a GitHub repository. The scripts use Spack, SHPC, and other common tools (e.g., `git, sed, patch`). The repository is structured in the following hierarchy of directories.

- `fixes/` contains patches implemented by Pawsey staff and to be applied to Spack prior to production use. They are meant to improve usability of Spack for Pawsey-specific use cases.
- `repo/` collects custom Spack package recipes for software not yet supported by Spack or that needed modification in the build process to work on Pawsey systems.
- `shpc_registry/` are custom Singularity-HPC (SHPC) recipes to deploy containers.
- `scripts/` contains Bash scripts used to automate the deployment process. There are several of them, each taking care of a well-defined task, such as installing Spack, or creating the directory hierarchy hosting module files.
- `systems/<system>` is a directory containing configuration files to customise the software stack deployment with respect to a specific system.

To make use of these scripts, a supercomputing system must have a directory within systems containing:

- `configs/` directory contains yaml configuration files for Spack. These point Spack to available compilers, system packages, and specify how generated module files look like, among other things. There are three sets of configuration files, based on who is executing Spack and the scope of the installation. The config files residing in `configs/site/` define how user installations behave as well as general settings for Spack valid in all use cases. They are moved to the `$spack/etc/spack` directory during the deployment process. Next, the `configs/project/` set targets project-wide installations. These configuration files will be positioned in the project-wide software stack of each project and override the default ones when Spack is executed through the `spack_project.sh` wrapper script. Finally, the `configs/spackuser/` configuration files shape system-wide installations, part of the software stack deployment process, performed by Pawsey staff. They will be placed in the personal space of the spack Linux user, a special account dedicated to software stack management, and override the system-wide configuration.
- The `environments/` directory groups Spack environment yaml files which outlines packages and their specifications to be installed, together with the compilers to use and the architecture to target (in our case, zen3).
- The `templates/` directory hosts a module file template for Spack to generate module files of installed packages.
- A `settings.sh` file defining variables used throughout the scripts, such as the installation paths, the version of Spack and SHPC to use, among other things.

B. Deployment Execution

Pawsey software stack is made of many complex parts interconnected with each other. For example, SHPC containers deployment requires that Singularity is installed, which in turn depends on Spack installing the `env_utils` environment. Therefore, its deployment must go through many steps and order of execution might not be straightforward. Automation drastically reduces deployment times and makes sure the process is performed correctly.

Deployment on Setonix is performed using a special Linux user, aptly named *spack*. Members of the applications team can log in as spack user using the `sudo su - spack` command. The main reason is to add an extra layer of protection against accidental changes by Pawsey staff; only the spack user has permissions to modify the system-wide software stack. In addition, the spack user has specific Spack configuration files overriding the default ones. They instruct Spack to install software in the system-wide deployment level instead of the default user-private location where installations by other users are directed to.

A fresh software stack deployment for Setonix that relies on sensible defaults requires only two command lines, as shown in Figure 6. The second command is the primary script that orchestrates the entire deployment. The script can be submitted for execution as a Slurm job on compute nodes.

```
$ git clone
https://github.com/PawseySC/pawsey-spack-
config

$ bash pawsey-spack-
config/scripts/install_software_stack.sh
```

Fig. 6. Minimal set of commands required to deploy the software stack on Setonix, using the default parameters in the **`pawsey-spack-config`** repository.

Before proceeding with a new installation, the settings.sh file may require updates, such as changing the `DATE_TAG` variable that is used to differentiate between multiple software stack deployments. Updating SHPC version and the list of containers to be installed is another easy and useful change that can be done prior to deployment. Any other change may require extensive testing and will be discussed in a later section.

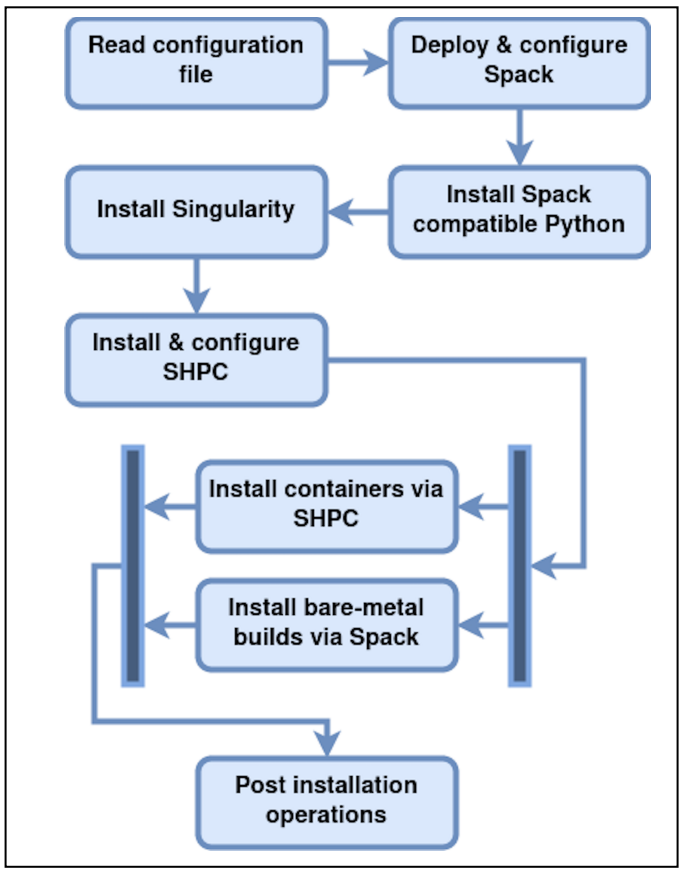

Fig. 7. Schematic representation of the automated installation procedure, highlighting its key conceptual steps.

The deployment process has two logical phases, pictured graphically in Figure 7. During the first phase, Spack and SHPC are installed and configured. In the subsequent one, these are used to install scientific software.

The deployment process starts with the execution of the `install_spack.sh` script. It downloads Spack on the system and applies the patches contained in the `fixes/` directory. Configuration files are instantiated with system-specific information and copied within the directory of the Spack instance (system-wide defaults and settings for user installations) and the spack user's home directory (for Pawsey-managed deployments). It also creates the directory structure that hosts the various components of the system-wide software stack.

Spack is then used to install a Spack-compatible Python (>=3.7), which is then set as default interpreter for Spack itself[3]. Subsequently, Singularity and SHPC are installed.

At this point, system-wide software and containers are ready to be deployed. To build and install bare-metal via Spack, we make use of Spack Environments, though we do not use these environments for access. Specifically, using the Spack yaml files contained in the `systems/<system>/environments/<env>` directories, we activate the environment, force concretisation, install the software packages and then deactivate the environment.

The *env_num_libs* and *env_utils* Spack environments are installed first because they provide critical dependencies to all others. Then, the remaining environments are installed in parallel. This is a critical moment of the deployment when one might see errors due to packages not being installed, especially if testing the deployment on a new version of the HPE Cray software, for a variety of reasons.

Module files are created by Spack using the templates for an explicit set of software to produce simple, human-readable module name [4]. Consequently, conflicts in module file generation may arise as well where there are multiple instances of the same software with different Spack specifications that do not result in a unique name. The ideal solution is to avoid the root cause, duplication. This is still hard to achieve in practice, and we resorted to "manually" pick which specification for which to generate the module file.

Independently and at the same time, containers are deployed through SHPC using the `install_shpc_containers.sh` script.

A post installation script is executed to tidy up the installation and apply final changes. First, it makes sure all module files for the installed software are present by regenerating them using a dedicated Spack command. The script also restricts access to licensed software by means of Linux groups. Spack has some support for this type of operations, but we decided to have more flexibility by employing our own Bash script. Another post-installation operation worth mentioning is the customisation of Singularity module files to mount Pawsey file systems and to bind HPE Cray libraries, see the template file at `systems/setonix/templates/modules/modulefile.lua` in the Pawsey Spack Config repository.

C. *Adapting the Deployment Process for a New System*

The deployment process is designed to be as much system independent as possible. System-specific knowledge and configurations are confined to within the systems directory. Those include Spack configuration files, parameter definitions such as installation location, compilers versions, and CPU architectures. This means that Pawsey, and any other centre, can easily adopt this very same deployment process for other systems. The easiest way to start is to create a copy of `systems/setonix/`, give it a sensible name, and start modifying the relevant information. While only the settings.sh file is technically required, it only makes sense to use Pawsey deployment system if embracing the three-level deployment approach. Hence, we suggest heavily borrowing from our Spack configuration files. The `environments/` directory is there to organise the software in categories and to introduce these in the module hierarchy. Once again, we encourage the use of such

[3] At the time when this part of the process was developed, the cray-python version was too old to properly support Spack.

[4] By default, Spack produces module names that contain a unique hash associated with the software. This would make for long and uninformative module names. Therefore, we limit modules to an explicit list. For all other software, we produce hidden modules that do contain a unique hash.

feature. In general, the farther one moves away from our approach, the more the generalised scripts must be modified.

D. Rebuilds, Upgrades and Maintenance

The software stack needs to be rebuilt when external dependencies get updated. This usually happens after an HPE delivered software upgrade. Another case is when the Pawsey staff decides to use a newer compiler version that is significantly more performant than the previous one. At the time of writing, Pawsey has not gone through a redeployment yet.

Several deployments can live at the same time on the system because they are versioned by deployment date. If an older deployment still works, it may be left on the system for at least the duration of the current allocation cycle. Thanks to the `pawseyenv` module, users can choose among the various software stacks available. This will affect not only the modules available system-wide, but also the project-wide and user-wide deployments.

Within the same deployment, software upgrades and maintenance can be done manually using Spack. It can be adding a new software or removing a faulty one. In most cases, redeploying the whole software stack can be overkilling and forces users re-installing their local software when they do not need to. The changes, however, must then be implemented appropriately in the GitHub repository too.

E. Reframe Regression Testing

The last step in our automated deployment process is to verify the installed software stack through regression tests. This has been automated using the ReFrame framework [11] developed by the Swiss CSCS HPC centre. Pawsey staff have written several tests to test the sanity and performance of various application software and numerical libraries in the installed software stack, checking that the modules are present and running regression tests for the correctness and acceptable performance values. These tests are run by our Pawsey administrators to test the readiness of the systems after each maintenance and any incidents to ensure the software stack is functional.

V. Evaluation

The first production deployment of the software stack as outlined in this paper was performed in May 2022, ahead of the public release of the Phase 1 of Setonix. At that time, all configurations were already stored on the Github repository, and the installation procedure was entirely scripted and partially parametrised. This procedure was not fully automated yet, and instead relied on about ten independent scripts, each requiring manual actioning. Despite the required higher degree of human integration, this semi-automatic character was indeed left on purpose, and was instrumental to inspecting the whole procedure, and verifying it for correctness and consistency. The positive outcomes of this first experience were: installation issues arising from the latest versions of system compilers and libraries were investigated and fixed; generation of module files against specifications and user requirements was assessed; interplay with the available hardware (Zen-2 and Zen-3 nodes) and the Cray software platform was tested and debugged, as well as integration with software components provided by other Pawsey teams (e.g., integration with shell login initialisation). The deployment was tested and completed over the course of about two weeks, and at first user access it was met with generally positive feedback.

Pawsey had to redeploy the same software stack in November 2022 after HPE Cray updated their software on Setonix. The only major change was in the compiler used; we moved from `gcc/11.2.0` to `gcc/12.1.0`. More importantly, the person in charge of executing the operation changed, making this deployment an ideal case to test usability and reproducibility of the process. By following the procedure documented in a README file, the newly appointed staff member managed to deploy the software stack in three days, with some of that time spent troubleshooting. A very good result, but more could be done. It is during this time that a driver script was developed, encoding the knowledge contained in the README file on how to execute the many scripts. We pushed towards complete automation by simplifying the scripts, reducing the number of them, and removing unnecessary user interactions. After several refactoring passes, we managed to deploy the software stack in a day on the test system.

Based on user feedback and our own evaluation, we noted a few things to improve. For instance, keeping the `~/.spack` cache directory between deployments lead to misbehaviours, such as software being installed under the `gcc/11.2.0` hierarchy despite `gcc/12.1.0` being used. This is one of the reasons we now have a different cache directory per deployment. A user suggested better tooling for managing project-wide installations, for example support for garbage collection.

VI. Software Stack with Accelerators

Setonix has both CPU-only and AMD GPU-nodes. We have developed our own build system for ROCm, called ROCm-from-source [12], to deploy the AMD GPU development platform independently of HPE Cray and without service disruption. ROCm libraries are added to the Spack configuration as external dependencies to be used for GPU-accelerated software builds.

Modules for GPU applications are marked with a suffix indicating the GPU architecture they are compiled for. The suffix is `amd-gfx90a` for the hardware currently installed on Setonix. These modules live in the same categories as the CPU applications, meaning that the current configuration does not introduce GPU acceleration in the software and module hierarchies. One of the reasons is that CPU libraries are likely still needed within GPU accelerated applications as well as pre, post and complementary processing steps. The choice of adding a suffix allows for multiple different GPUs to coexist.

Pawsey is also aiming to provide quantum computer acceleration and QPU-enabled software as the Centre houses a Quantum Brilliance room-temperature quantum computer [13], and plans to potentially host additional quantum kits in the future. In the same fashion, these software stack will make use of dedicated suffixes to identify QPU-enabled software running on the distinct types of quantum hardware.

VII. Conclusion and Future Work

In this paper, the authors presented an automated software stack deployment process for an HPE Cray EX supercomputer. It was designed to smoothly install the many components a modern software stack in minimal amount of time with improved reproducibility and usability. Users will directly experience the advantages of using modern technologies such as Spack and SHPC, as the same instances used for system-wide deployments are available for project-wide and user-private installations.

The deployment process is subject to constant development to add functionality and improvements. Based on experiences with the past two deployments, we have already identified several potential improvements. Updating the extensive list of external dependencies listed in the configuration files requires extensive work and we would like to reduce those to a minimum, letting Spack build them. This reduces the effort in deploying software after an OS update but also increases the chance that the prior software stack deployment can be reused.

Another aspect requiring attention is the resolution of module generation conflicts that block us from having a complete automated deployment. Besides looking into better defining Spack specifications for the software installed system-wide, we think the situation will improve once Spack matures and the chance of unwanted software duplication diminishes.

Future work is to integrate the deployment of reframe tests completely within the deployment scripts.

Lastly, would like to improve tools and support for project-wide and user-private installations. For instance, users should be able to easily migrate their currently installed software when a new software stack is made available.

Acknowledgment

We would like to acknowledge the Whadjuk people of the Noongar nation as the traditional custodians of this country, where the Pawsey Supercomputing Research Centre is located and where we live and work. We pay our respects to Noongar elders past, present, and emerging. This work was supported by resources provided by the Pawsey Supercomputing Research Centre with funding from the Australian Government and the Government of Western Australia.

We would also like to thank the lead developers of Spack, who generously donated their time for a Q&A session to address the shortcomings identified by our initial tests. We would like to thank the entire Pawsey Applications Team for their contributions to this work. PJE would like to thank all the friends and family met during a long, much need holiday that reenergised them.

References

[1] "Pawsey Supercomputing Research Centre," [Online]. Available: https://pawsey.org.au. [Accessed 4 April 2023].

[2] R. C. Bording, C. Harris and D. Schibeci, "Using Maali to Efficiently Recompile Software Post-CLE Updates on a CRAY XC System," in CUG2015 Proceedings, 2015.

[3] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski and S. Futral, "The Spack Package Manager: Bringing Order to HPC Software Chaos," in SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2015.

[4] M. Melara, T. Gamblin, G. Becker, R. French, M. Belhorn, K. Thompson, P. Scheibel and R. Hartman-Baker, "Using Spack to Manage Software on Cray Supercomputers," in CUG2017, 2017.

[5] NERSC, "Spack Artifacts," [Online]. Available: https://gitlab.com/NERSC/nersc-user-software/-/tree/main/spack-artifacts. [Accessed 4 April 2023].

[6] V. Sochat and A. Scott, "Collaborative Container Modules with Singularity Registry HPC," Journal of Open Source Software, vol. 6, no. 63, p. 3311, 2021.

[7] Pawsey, "Software Stack Policies," [Online]. Available: https://support.pawsey.org.au/documentation/display/US/Software+Stack+Policies. [Accessed 4 April 2023].

[8] 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.

[9] Pawsey Supercomputing Research Centre, "Pawsey Spack Config," [Online]. Available: https://github.com/PawseySC/pawsey-spack-config. [Accessed 4 April 2023].

[10] TACC, "Lmod: a New Environment Module System," [Online]. Available: https://lmod.readthedocs.io. [Accessed 4 April 2023].

[11] CSCS, "ReFrame," [Online]. Available: https://reframe-hpc.readthedocs.io. [Accessed 4 April 2023].

[12] C. Di Pietrantonio, "Building AMD ROCm from Source on a Supercomputer," in CUG 2023, unpublished.

[13] S. N. Saadatmand, S. Yin, M. L. Walker, M. W. Doherty, M. Cytowski and U. Varetto, "qbOS: a Python framework for the development of coprocessing quantum-classical applications," in First International Workshop on Integrating High-Performance and Quantum Computing, 2021.

[14] V. Sochat, M. Muffato, A. Stott, M. De La Pierre and G. Stuart, "Automated Discovery of Container Executables," Journal of Open Research Software, p. in press, 2023.