THEME ARTICLE: TRANSFORMING SCIENCE THROUGH SOFTWARE: IMPROVING WHILE DELIVERING 100X

Providing a Flexible and Comprehensive Software Stack Via Spack, an Extreme-Scale Scientific Software Stack, and Software Development Kits

James M. Willenbring, *Sandia National Laboratories, Albuquerque, NM, 87185, USA*

Sameer S. Shende, *University of Oregon, Eugene, OR, 97403, USA*

Todd Gamblin, *Lawrence Livermore National Laboratory, Livermore, CA, 94550, USA*

To manage the complex demands of modern high-performance computing (HPC), software applications increasingly depend on software developed by other teams, often at other institutions. An HPC software ecosystem approach is required to support dependencies on third-party scientific software. An ecosystem approach provides layers of activity above the individual software product level that promote interoperability, quality improvement, porting, testing, and deployment. The U.S. Exascale Computing Project (ECP) developed its HPC software ecosystem using a three-pronged approach. First, the ECP adopted and invested in Spack, a package manager designed to handle complex HPC package dependencies. Second, the ECP created the Extreme Scale Scientific Software Stack, an effort that supports developing, deploying, and running scientific applications on HPC platforms. Third, the ECP supported software product communities, or software development kits, to develop and promote best practices, improve software interoperability, and other collaborative efforts. This article describes ECP contributions to HPC software ecosystem challenges.

Given the complexity of modern software architectures and the need for performant features that require specialized developer knowledge to design, implement, and sustain, software applications increasingly and necessarily depend on software developed by other teams for important functionality. Depending on another piece of software inherently carries some risks. The risk is increased when the software is developed at another institution, or when it does not share common funding streams, as that impacts project priorities. Many issues complicate the adoption of third-party software. Will the software build and run properly on all the target platforms? If so, is the software easily available on those platforms? Is the software interoperable with the application's other software dependencies? Is there a smooth path for upgrading to new versions, and will those versions continue to be interoperable with other needed software? Will the software dependencies continue not only to be maintained, but sustainably maintained into the future?

An important aspect to the aforementioned questions is that mitigation of these concerns requires more than a collection of software product teams independently following good software practices and having access to useful tools. The dependency graphs of high-performance computing (HPC) software are so complicated that excellence in software sustainability, delivery, and deployment by a single software product team, or even all software product teams, is necessary, but not sufficient. Rather, an HPC software ecosystem approach is required to support dependencies on

© 2024 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/
Digital Object Identifier 10.1109/MCSE.2024.3395016
Date of publication 30 April 2024; date of current version 5 July 2024.

third-party scientific software. An ecosystem approach provides layers of activity above the individual software product level that promote sustaining interoperability, quality improvement, porting to new architectures, and regular testing and deployment on a variety of existing architectures.

The HPC software ecosystem developed and strengthened under the software technology (ST) thrust of the U.S. Exascale Computing Project (ECP) is based on three foundational components: Spack, the Extreme Scale Scientific Software Stack (E4S), and software development kits (SDKs). These key efforts have contributed significantly to improving the overall level of software quality and software engineering practices within the ECP ST and beyond into other parts of the HPC software community. In this article, we review the software complexities inherent to the ECP's mission. We then describe the roles of Spack, SDKs, and the E4S software distribution. We look at the testing, deployment, usage, and security challenges that had to be overcome to ensure the reliability of the ECP software stack, and we briefly discuss post-ECP priorities and initiatives.

INCREASING SOFTWARE COMPLEXITY

The ECP was structured around three mission goals:

1) Deliver applications that use exascale computers to solve previously intractable problems.
2) Create an integrated software stack for exascale systems.
3) Deploy ECP software products to pre-exascale and exascale systems.

A number of trends influenced the path to exascale computing, but the main one was the need for accelerated computing. The end of Dennard scaling and the slowing of Moore's law led to increased prevalence of multicore chips and eventually to GPUs. GPUs, as the most power-efficient way to achieve high computational efficiency, became the heart of *all* of the U.S. exascale machines. The ECP software stack was therefore required to run across *three* different accelerated architectures.

Switching to a new GPU is not like switching to a new CPU. GPUs required rethinking the way parallel code was written, which could be considerably different depending on *which* GPU was used. Nvidia GPUs are programmed using CUDA,[a] AMD GPUs using HIP[b] and ROCm,[c] and Intel GPUs using a new standardized programming model called *SYCL*.[d] It is intractable for most application developers, who are typically computational physicists, to support and tune their code for all of these programming models. The learning curve and support burden are simply too high.

The ECP shielded application developers from the underlying GPU programming models through its software stack. Application developers, daunted by the prospect of writing and maintaining the same code in three different programming paradigms, were heavily incentivized to adopt so-called *performance portability* libraries like RAJA[1] and Kokkos.[2]

A NUMBER OF TRENDS INFLUENCED THE PATH TO EXASCALE COMPUTING, BUT THE MAIN ONE WAS THE NEED FOR ACCELERATED COMPUTING.

Math and infrastructure library teams, in many cases, also adopted these frameworks, or they made their libraries' portability layers in their own right—by implementing core numerical algorithms and methods in all of the different GPU programming models, exposing only their traditional application programming interfaces to applications. Many applications that previously would have never considered using an external library, let alone one written by a team at another institution, now seriously considered leveraging the considerable capabilities of ECP libraries.

Ultimately, GPUs and the need for portability on extreme-scale machines led ECP applications and ST to rely on each other much more heavily than in the past, and the ECP's stack evolved into a massive set of interconnected components.

To manage the integration of so many software components on so many architectures, new methods for integrated development, building, and testing were needed.

Previously, most developers worked on a single project, and the ramifications of any change were tracked only as they pertained to the project. Within the ECP, interdependent software packages were managed by different teams, and each package was updated frequently as its developers implemented new features and capabilities. Library upgrades can have repercussions in many other software packages, and many

[a]https://en.wikipedia.org/wiki/CUDA
[b]https://github.com/ROCm/HIP
[c]https://en.wikipedia.org/wiki/ROCm
[d]https://en.wikipedia.org/wiki/SYCL

libraries needed to break backward compatibility as they changed their interfaces to better handle GPUs. Finding compatible versions and testing new configurations manually led to many trial-and-error iterations and repeated builds of the software stack. This was already difficult when installing a single application's dependencies, but it quickly became unmanageable for facility-level deployments.

SPACK

Spack[3,4] is an open source package manager that was created to manage the complexities of software integration for HPC. A full description of Spack's capabilities is beyond the scope of this article, but at a high level, Spack comprises three primary components. First, the *spec syntax* allows users to concisely describe build options and constraints. Examples of the spec syntax are shown in Table 1. The spec syntax can be used on the command line or in configuration files to indicate how Spack should build packages. Second, Spack implements a *domain-specific language* for writing parameterized build recipes for software packages. Figure 1 shows a short example package recipe. Directives and constraints in the package recipe leverage the spec syntax. Finally, the *concretizer* is the engine at the core of Spack that configures parameterized builds according to package constraints and user preferences. The concretizer combines *abstract* (under-specified) spec constraints from the user and from packages, and it produces a *concrete*, or (fully specified) package build configuration.

Before Spack, developers who wanted to integrate many software packages would frequently need to build by hand. This typically involved downloading source code, configuring it for the target platform, ensuring that appropriate optimization flags were used, compiling and iterating to fix any issues, and then building additional dependent packages. The process was error prone and it could take weeks to build large software stacks, especially on unfamiliar platforms in configurations not yet explored.

TABLE 1. Examples of Spack's *spec* syntax.

Example syntax	Meaning
`hdf5%gcc`	Use a particular compiler
`hdf5@1.10.2`	Require version(s)
`hdf5%gcc@10.3.1`	Require compiler version(s)
`hdf5+mpi` `hdf5~mpi`	Enable (+)/disable (~) variant

`hdf5 mpi=true`
`hdf5 api=default`
`hdf5 target=skylake`
	Require a particular variant or build target value

```
1  class Example(Package):
2      """Spack recipe for a package called `example`."""
3      url = "https://example.com/example-2.0.0.tar.gz"
4
5      version("2.0.0", sha256="8ffb3927dd8d903272999f51a...")
6      version("1.1.0", sha256="29966e69cc02a958519f5a9ac...")
7      version("1.0.0", sha256="3251439aba391effcbf49958b...")
8
9      variant("bzip", default=True, description="enable bzip")
10
11     depends_on("bzip2@1.0.7:", when="+bzip")
12     depends_on("zlib")
13     depends_on("zlib@1.2.8:", when="@1.1.0:")
14     depends_on("mpi")
15
16     conflicts("%intel")  # Known failure with intel compiler
17     conflicts("target=aarch64:")  # Does not support arm
18
19     def install(self, spec, prefix):
20         configure(
21             f"--with-zlib={spec['zlib'].prefix}",
22             f"--with-bzip={spec['bzip2'].prefix}"
23             if "+bzip" in spec else "--without-bzip",
24         )
25         make()
26         make("install")
```

FIGURE 1. A `package.py` build recipe written in Spack's embedded Python domain-specific language.

Figure 1 shows some of the declarative mechanisms Spack provides to simplify this process. Line 3 tells us where we can download the package source code. Lines 5–7 declare 3 different versions, the download locations for which are extrapolated from the `url` on line 3. Each has an associated `sha256` checksum that can be used to verify that the source was downloaded without error. Line 9 declares an option, which allows support for `bzip2` compression (and the dependency on the `bzip2` library) to be enabled or disabled using `+bzip` or `~bzip`. The package has three dependencies, specified on lines 11–14. The `bzip2` dependency is optional. There is an unconditional dependency on the `zlib` library; later versions of this `example` package depend on later versions of `zlib`. Finally, the package depends on message passing interface (`mpi`). `Mpi` is a *virtual package* that represents the *source interface* to MPI implementations like `mpich`, `openmpi`, `cray-mpich`, and others. This package has two conflicts on lines 16 and 17: it does not work with `intel` compilers, and it does not support the `ARM` architecture. Finally, lines 19–26 are the imperative build recipe itself. The `install()` method is passed a concrete `spec` object, which it queries to construct arguments to `configure`.

At the beginning of the ECP, Spack had started to gain traction in the HPC community. In 2016, it had contributors from more than 20 different organizations (see Figure 2), and its repository contained ~422

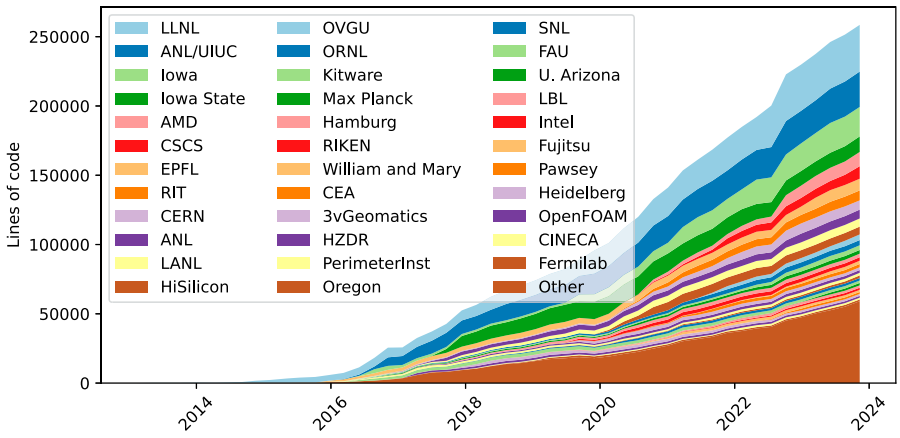

FIGURE 2. Contributions (in lines of code) to Spack packages by different organizations.

package recipes (see Figure 3). Although there were other tools that were also gaining popularity at the time, some even targeted at HPC, Spack was chosen for its flexibility and its ability to support not only system administrators deploying already-released software but also software developers managing many components in development concurrently. EasyBuild[5] is another installation system which, like Spack, targets HPC. However, its build recipes are rigid and do not allow the user to easily specify builds with new options or different compilers. Spack was inspired in part by the Nix[6,7] package manager and its sibling Guix.[8] Indeed, Spack adopts their deployment models to allow multiple concurrent installations of different versions of the same package. However, like EasyBuild, their package recipe formats are rigid and do not allow for version ranges or constraint solving. EasyBuild also does not support binary packaging, a feature on which many of the ECP's productivity advancements relied. Finally, the popular Anaconda[e] software distribution and its package manager conda *do* support dependency solving, but it only supports installing prebuilt binaries. ECP software developers frequently needed to build new configurations from source. Spack is designed to allow developers to quickly configure the package *build*, which enabled them to iterate rapidly as they integrated packages in new ways and ported them to exascale platforms.

One of the areas where Spack utilizes conditional logic is in the area of GPU integration. ECP packages typically needed to support CUDA, ROCm, SYCL, and CPU builds of many of their components, and applications frequently used different mixes of CPU- and

[e]https://docs.anaconda.com/

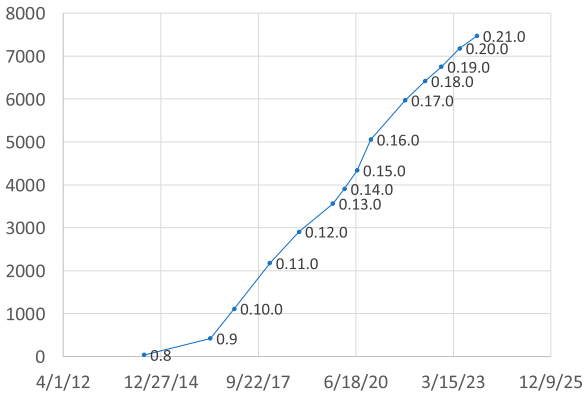

FIGURE 3. Number of packages in Spack over time, labeled by releases.

GPU-enabled builds depending on their own configuration. The ECP adopted Spack for software integration, and software projects participating in the ECP's ST thrust were all expected to be buildable and deployable via Spack. This provided ECP users with access to many different software products using a single but customizable installation interface. The leverage provided by Spack grew throughout the ECP. Spack's ecosystem grew from only 400 packages in 2016 to more than 7500 packages by the end of the ECP in 2023 (see Figure 3). Similarly, Its contributor base also grew, with more than 300 organizations around the world contributing to the package repository by 2023 (see Figure 2). The broad community behind Spack has strengthened the project's sustainability, and Spack is now used at every ECP computing facility.

SDKs

Packages in the HPC software ecosystem are not independent, and ensuring that packages *within* Spack continue to function correctly together could not be accomplished with individual package contributions alone. The ECP thus introduced the concept of SDKs: collections of related software packages in which coordination across package teams improves usability and development practices, fostering community growth among teams that develop similar and complementary capabilities. SDKs are typically developed, built, and tested together. The initial ECP SDKs were created by dividing ECP ST software products into specific groupings, each focused on a particular community of practice. SDKs have several important attributes, including the following:

- *Domain scope*: Each SDK comprises packages whose capabilities are within a natural functionality domain. Packages within an SDK provide similar capabilities that can enable leveraging of common requirements, design, testing, and similar activities.
- *Interaction models*: Packages may be compatible, complementary, and/or interoperable, and this determines how the packages interact with one another. Interoperability includes common data infrastructure, or the seamless integration of other data infrastructures, and access to capabilities from one package for use in another. SDKs with robust and well-tested interaction models allow development teams to focus on their areas of expertise, leveraging capabilities developed by other teams outside of those areas, and allow users to access a rich set of capabilities across multiple software packages.
- *Community policies*: These include expectations for how package teams will conduct activities, the services they provide, the software standards they follow, and other practices that can be commonly expected from a package in the SDK. The policies are aimed at increasing the sustainability of the software packages as well as the whole software ecosystem. Community policies exist at the E4S level, and optionally at the individual SDK level.
- *Community interaction*: Communication among teams that helps to bridge culture and build a common vocabulary.
- *Coordinated plans*: Development plans for each package will include efforts to improve SDK capabilities and lead to better integration and interoperability. Planning at the SDK level supplements rather than replaces planning at the individual package level.
- *Community outreach*: Efforts to reach out to the user and client communities will include explicit focus on the SDK as a product suite.

The ECP used the existing math libraries' SDK, the xSDK, as a model to establish SDK efforts in the areas of programming models and runtimes (PMRs), data and visualization, development tools, software ecosystems (of which the E4S was a part), workflows, and code analysis and data mining tools. The xSDK began in 2014[9] as part of the initial Interoperable Design of Extreme-scale Application Software project.[f] The first xSDK release in 2016, version 0.1, contained six software products. As of its second release in February 2017, the xSDK began to use Spack for integration, and by the most recent version 1.0 release in 2023, there were 28 member packages.

A key aspect of building an SDK community is clearly identifying the specific needs of the community that will make the SDK valuable. The value propositions for each SDK effort are different and highly specific to each SDK, but some examples of the kinds of needs that an SDK community can help to meet are the following:

1) Achieving and maintaining the interoperability of member packages. This is critical for users who want to leverage a variety of packages in a single ecosystem.
2) Agreeing on supported version(s) of third-party software commonly used by member packages as well as policies for adding or dropping support for new versions.
3) Providing coordinated support for commonly used third-party software that has been abandoned or is otherwise not sufficiently maintained.
4) Developing common testing infrastructure for continuous integration (CI) or other testing to help promote the portability and interoperability of member packages. When done in the smaller context of an SDK, as opposed to, for example, the E4S, it is sometimes possible to catch issues sooner, or closer to the tip of development, which can make testing interoperability at the E4S or other higher levels more sustainable.

[f]https://ideas-productivity.org/activities/ideas-classic/

5) Coordinating porting efforts such that common platform issues do not have to be resolved independently by many teams.
6) Providing a trusted source for complementary tools. In this way, a set of tools can be used together as a suite, which is more valuable than the sum of its parts.
7) Serving as a common point of contact for users who have unmet requirements. After gaining an understanding of the requirements, it may be appropriate for one or more member packages to coordinate to provide the necessary functionality.
8) Providing a forum to develop standard interfaces and critical best practices as well as engage in collaborative design. Although some diverse efforts in an area allow for the best ideas to proceed, it is important not to duplicate efforts unnecessarily.

For example, the members of the xSDK are highly interoperable, so item 1 is especially important in the math libraries area. As a result, a lot of effort is put into achieving and testing interoperability and developing a suite of examples[g] for utilizing the interoperability between xSDK packages. Items 2, 3, and 4 were important to the data and vis SDK effort. The CI testing established in the data and the vis SDK now contributes to Spack CI testing. The development tools that SDK undertook efforts in porting and testing enhanced the value of its member packages. The PMR SDK invested in testing and containerization efforts, which helped to make its member products more robust in the larger HPC ecosystem. Each SDK had a different focus and significant size differences in terms of staffing, but all the SDK efforts were targeted at strengthening the associated community.

E4S

The E4S project[10] aims to tame both the complexity and portability problems by providing a curated distribution of numerical libraries, runtime systems, and tools that lowers the barrier for entry for the HPC, artificial intelligence/machine learning (AI/ML), and electronic design automation (EDA) developer communities. The E4S is a community effort to provide open source software packages for developing, deploying, and running scientific applications on HPC platforms. It aims to deliver a modular, interoperable, and deployable software stack based on Spack and including ECP SDKs. The E4S provides both source builds for native, bare-metal installations and cloud-based images as well as containers of a broad collection of software packages for secure, reproducible, container-based deployments. The E4S exists to accelerate the development, deployment, and use of HPC software, lowering the barriers for developers and users. The E4S is the largest collectively tested piece of the ECP software ecosystem and encompasses all ECP ST on which applications rely.

Although Spack provides access to thousands of software packages, The E4S is focused on just a bit more than 100 packages that are central to the HPC software ecosystem. The E4S includes, via Spack, these 100 packages and more than 500 dependencies they require to run. The E4S is designed to support a complete HPC software ecosystem. Due to funding and other practical reasons, the set of packages included in the E4S so far has consisted primarily of products that were a part of ECP ST. The E4S team ensures that E4S packages can be built together in a Spack environment and are functional on key target platforms.

THE E4S TEAM ENSURES THAT E4S PACKAGES CAN BE BUILT TOGETHER IN A SPACK ENVIRONMENT AND ARE FUNCTIONAL ON KEY TARGET PLATFORMS.

The E4S provides a set of containers (base and full featured) that support GPUs from Intel, AMD, and Nvidia on x86_64, ppc64le, and aarch64 architectures. It also supports bare-metal installations. It provides tools that support the replacement of MPI in containerized applications (e4s-cl) and tools to customize containers (e4s-alc), starting from base container images. The E4S plays a critical role in the ecosystem due to its releases, CI testing, documentation, and community policies.

The E4S has a quarterly release cadence: both bare-metal releases on large computing facility machines, and containerized releases. The Spack release configurations provide instruction on how to build the software included in the E4S on a multitude of architectures, but almost as importantly, also clearly document any software packages that are not currently working for the various configurations. The E4S releases are not commonly used in their entirety. Rather, software development teams, computing facility staff, and individual users most commonly install a subset of the software products included in an E4S release that suits their individual needs. Often this means changing the version, or even pointing to a system version of one or more software products. For these reasons, the E4S releases are

[g]https://github.com/xsdk-project/xsdk-examples

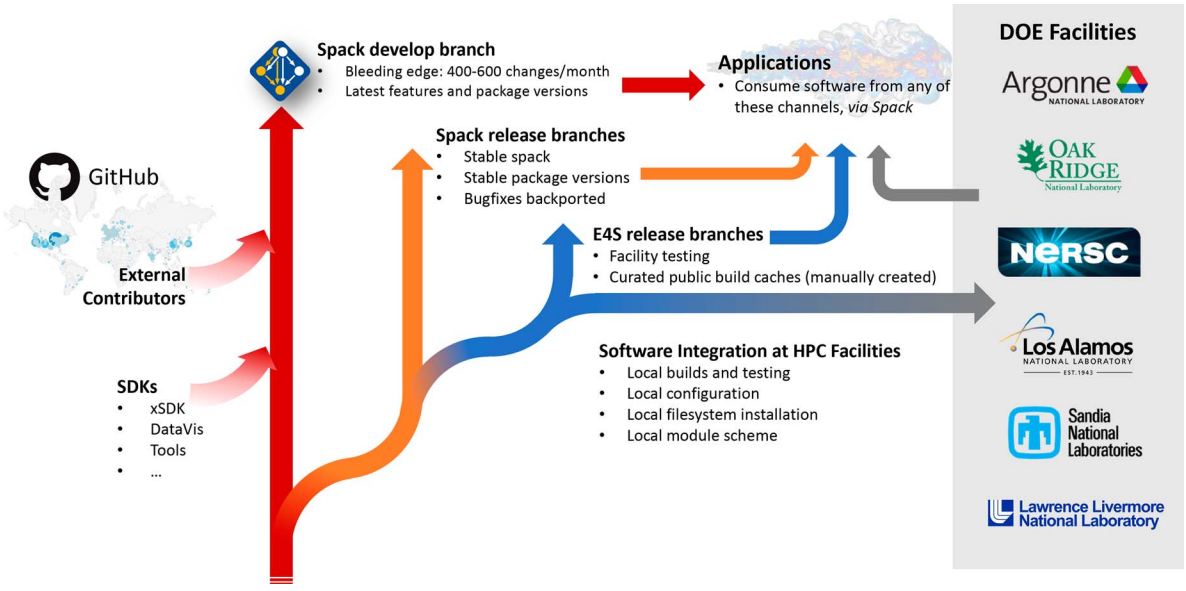

FIGURE 4. Contribution and release structure of Spack and E4S. NERSC: National Energy Research Scientific Computing.

often a reasonable starting point, rather than exactly what a user needs. This is to be expected, as users' needs vary dramatically. A single facility deployment can be very rigid to work with, but an E4S release provides the configuration files that can be modified slightly for each users' needs.

E4S CI testing provides the data necessary to maintain software package stability and interoperability. Additionally, the E4S staff assists in setting up new builds, triaging failures, and updating the E4S target release versions of software products. Another critical E4S contribution is the E4S Validation Test Suite,[h] which provides postinstallation sanity tests for nearly all E4S products.

The E4S Doc Portal[i] provides basic information about all E4S software products, including the type of software, a brief description, accelerator support, licensing information, summaries of test support, and a link to the product homepage. This information was initially requested by computing facility staff.

The E4S adopted a set of community policies to establish quality standards and promote quality improvement for member packages. The E4S Community Policies[j] are modeled after the xSDK Community Policies.[k] The process for developing the policies was led by the ECP SDK leadership group, with consultation and a buy-in from the ECP ST product development teams. The policies are designed to be E4S membership criteria. Although assessments of compatibility with the policies has been performed, a strict requirement of compatibility has not yet been enacted. The policies touch on a variety of topics important to software quality, including testing, installation, sustainability, documentation, accessibility, and error handling.

DEPLOYMENT AND USAGE COMPLEXITY

Depending on the application or software component, development within the ECP progressed at different rates, all of which needed to be integrated continuously for consumption by users, applications, and HPC facilities. Spack's flexibility allowed the ECP to implement a multitiered deployment process, organized into several streams with different update frequencies, cadences, and testing policies.

Figure 4 shows the cross-ecosystem coordination used during the ECP. All new package commits start in Spack's `develop` branch, which moves quickly, with 400–600 commits from hundreds of contributors per month. This includes commits from ECP SDK teams. Contributions can be feature updates, major package changes, or simple version bumps. Periodically, the Spack team publishes *stable* release branches, which change much less frequently. Package versions are fixed and only critical bug fixes are backported onto these branches. Users can choose to use them for longer-term stability. From Spack releases, the E4S

[h]https://github.com/E4S-Project/testsuite
[i]https://e4s-project.github.io/DocPortal.html
[j]https://e4s-project.github.io/policies.html
[k]https://figshare.com/articles/online_resource/xSDK_Community_Package_Policies_1_0_0/13087196

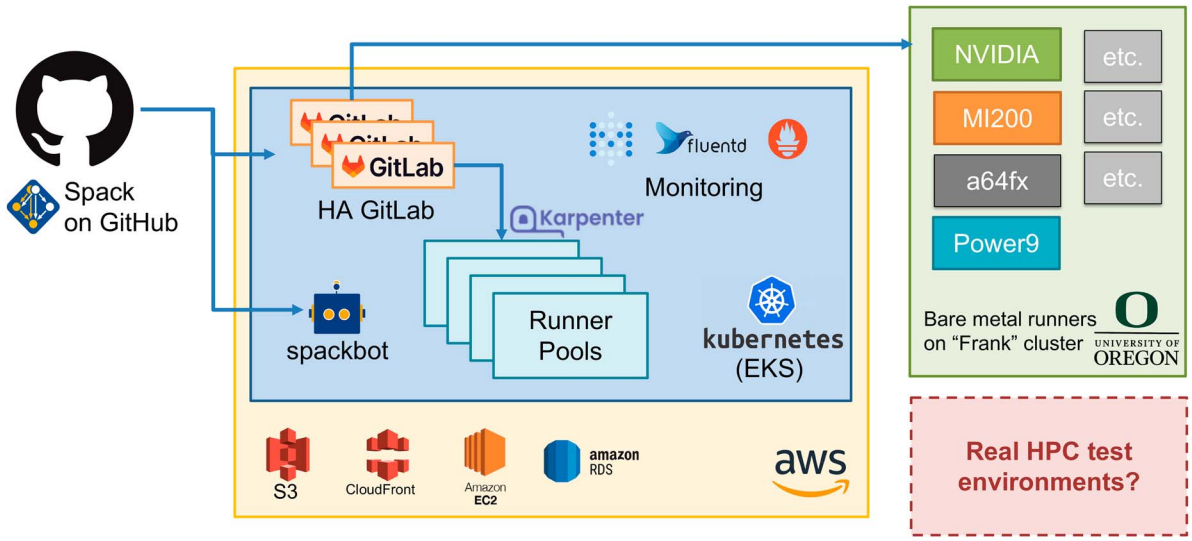

FIGURE 5. Cloud-based continuous integration system architecture. EKS: Elastic Kubernetes Service; CDN: Content Delivery Network; S3: Simple Storage Service; RDS: Relational Database Service.

team curates a set of configurations that are ported and deployed at ECP facilities. The changes needed for these per-facility ports are eventually upstreamed to Spack's develop branch, usually after each E4S release. Applications may consume packages from *any* (or all) of these branches, or from facility deployments of the packages, depending on their needs.

Spack's `develop` branch and its stable releases were tested automatically in a CI system split between the cloud and the University of Oregon's (UO's) test cluster, "Frank." Downstream E4S ports for facilities are tested with manual CI. The reasons for this are twofold. First, all of the U.S. Exascale systems are HPE[l]/Cray machines, and during the ECP, there was not a suitable license that allowed the Cray software environment to be run in public CI on non-HPE resources (like cloud instances). These runs therefore had to be done manually by the E4S team. Second, all the exascale and online pre-exascale systems are private and cannot run public CI jobs in response to contributions from GitHub. HPC facilities currently have a requirement that all code run on their system must be triggered by trusted users, and online contributors are not always known. Although the ECP developed the Jacamar CI system[m] to allow internal trusted HPC users to trigger jobs automatically, there still does not exist a tool that can associate GitHub user identities with internal HPC user identities, so Jacamar could not be used to solve the public CI problem.

[l]http://www.hpe.com
[m]https://ecp-ci.gitlab.io

CONTINUOUS INTEGRATION

The scale of integration in the E4S and the velocity of contributions to Spack required new levels of automated testing. As mentioned earlier, any update to a package in the ecosystem may have broad effects, breaking builds and introducing incompatibilities with past versions. To manage more than 600 packages in the E4S in this environment, continuous testing was needed to ensure that new commits did not introduce new bugs. The Spack team, in collaboration with Kitware, Amazon Web Services (AWS), and the E4S team, developed a reliable, high-availability (HA) CI system to ensure that configurations of interest were built and tested *on every pull request* to Spack.

Figure 5 shows the CI system's architecture. Source archives and builds are archived in Amazon Simple Storage Service buckets, and they are distributed to worldwide edge caches via the CloudFront content delivery network. The CI system tracks commits on GitHub, but builds are coordinated using an HA GitLab CI instance. We chose GitLab so that the same sort of builds can be orchestrated with the GitLab instances used on site at most of the ECP HPC sites. GitLab orchestrates builds on Amazon Elastic Compute Cloud (EC2) spot instances using an AWS tool called *Karpenter*, which ensures that builds take place on nodes with the correct microarchitecture, compute,

and memory capabilities. These systems are all orchestrated in containers using Amazon's Elastic Kubernetes Service implementation. A number of monitoring services run to ensure that the team can track system errors and fix issues rapidly. For example, the Ganglia[n] monitoring system is used to check the health of the nodes. The GitLab tests from Trilinos[11] and SOLLVE[o] themselves also produce a report that helps in maintaining the health of the system. Alongside the cloud services, the UO's Frank cluster[p] provides more exotic GPU nodes for testing. Missing are HPC test environments at the ECP facilities. As mentioned earlier, currently, security concerns prevent us from doing continuous testing on these machines. Fortunately, in many cases, Frank provides access to machines that have hardware similar to that of larger facility machines. Although not a perfect substitute, Frank resources have been useful for numerous code teams in both preparing for and maintaining portability to facility machines.

Together, the AWS and UO resources are able to deliver 115,000 software package builds per week. The builds range from small packages like *zlib*, which take minutes or seconds to build, to full builds of the ParaView visualization suite[q] and the TensorFlow ML library,[r] which can consume many more resources and may require 40 min to an hour to build.

When a package is changed in Spack's CI system, all of its *dependents* are built as well. This means that the packages deep in the dependency hierarchy may trigger hundreds or thousands of builds. This is necessary to ensure that new versions do not introduce application binary interface breaks or cause new inconsistencies within the stack. To ensure that our build system does not use an intractable number of resources, we use very aggressive caching. The builds completed in the pipeline are stored and can be reused by other pull requests to ensure that we do not (often) do the same build twice. Using Spack enables binary build caches to be used easily in CI; Spack uses fine-grained configuration hashes to identify and reuse builds cached in pipeline storage. The builds and weekly snapshots of five different versions of the E4S are made available on `cache.spack.io`,[s] and users can browse packages on `packages.spack.io`.[t]

[n]https://developer.nvidia.com/ganglia-monitoring-system
[o]https://www.bnl.gov/compsci/projects/sollve/
[p]https://oaciss.uoregon.edu/frank
[q]https://www.paraview.org
[r]https://www.tensorflow.org
[s]https://cache.spack.io
[t]https://packages.spack.io

SUPPLY-CHAIN SECURITY

The Spack build pipeline automates builds from hundreds of contributors to the Spack repository on GitHub. The repository is intentionally open for pull requests to encourage new contributions, and *anyone* with a GitHub username can submit a pull request for review and integration. This approach helps to ensure that packages stay updated, as no team within the ECP could sustain constant updates to hundreds or thousands of packages themselves. We rely on community contributions to keep the software ecosystem current.

Public CI is a difficult problem from a security perspective because the very openness that allows our stack to be sustained increases the risk that a malicious actor could inject harmful code into a binary package or into our cache. Although this was already a problem for source builds in Spack, the issue is amplified for binary builds because binaries are not reviewed by maintainers and can be cached. Without proper mitigations, an attack *could* be hidden in a binary within an untrusted pull request build.

To mitigate this issue in our pipelines, we developed a two-tier system of builds by which *public* binaries from the `develop` and release branches of Spack are always signed and are only built from reviewed and maintainer-approved code. The Spack project has ~40 trusted maintainers who can merge changes to packages. Our security model is that we (and users of Spack) trust these maintainers to ensure that Spack itself is secure. We wanted binary packages to be just as secure as the source packages we traditionally provided, so we devised a way to ensure that every *published* binary package was built only from recipes reviewed by maintainers.

Figure 6 shows the CI system. Untrusted pull requests automatically trigger builds, but they do not use public, signed build caches. Instead, untrusted builds *only* store to special pull request (PR)-only build caches, and these binaries cannot be reused by builds for releases or for the `develop` branch. They serve only to guarantee to maintainers that the build works before it is merged, and they allow contributors to easily iterate with maintainers on builds and fixes. Once a build is successful in the untrusted PR area, *then* maintainers review the changes to recipes, and if the maintainer approves the recipe, the entire package is built *only* from approved, merged recipes and binary caches. This ensures that maintainers have reviewed all the code and configurations in published, trusted caches, while still helping maintainers with automated builds. Together, these techniques allow us to *securely* manage

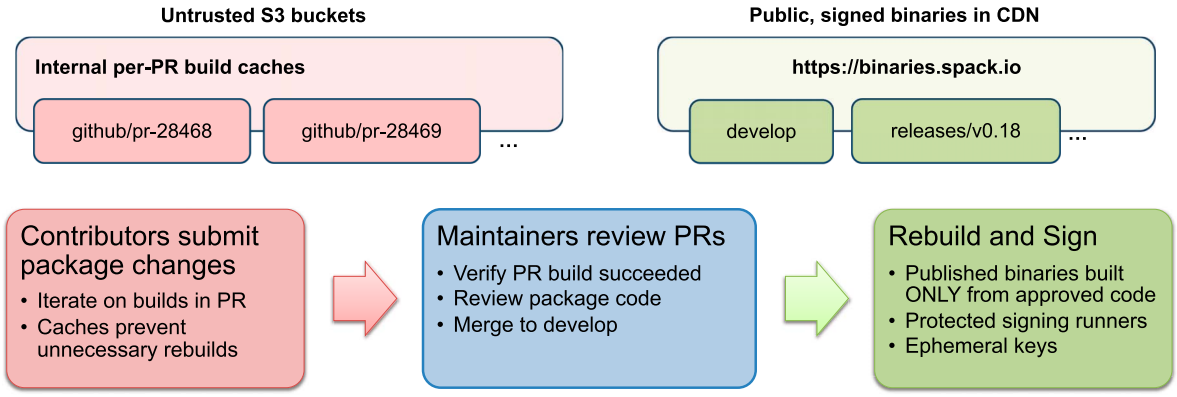

FIGURE 6. Security model for public CI.

the Spack ecosystem. Currently, we maintain ~5000 builds of more than 1000 packages in CI this way.

FUTURE ACTIVITIES

The end of the ECP has brought an opportunity to reflect on how the various components within the ecosystem have been developed. During the growth and exploration of the ECP, some overlapping activities emerged. For example, since the development of the E4S Validation Test Suite, Spack's `spack test` capability has matured considerably. This has been recognized in the E4S team's effort to trivially integrate tests supported through `spack test` into the E4S Validation Test Suite, but further efficiencies regarding test suites may be achieved. Also, similar efficiency improvements will be explored with regard to Spack, the E4S, and xSDK CI and release testing.

The E4S supports a broad collection of HPC tools. In the E4S AWS image, more than 50 open source EDA tools are supported, including OpenROAD,[12] OpenLANE,[u] OpenFASoC,[v] and Xyce.[w] In the future, the E4S will target more AI/ML tools based on Python. The E4S currently supports popular tools such as TensorFlow and PyTorch, which target GPUs on multiple platforms.

CONCLUSION

The ambitious goals of the ECP within the context of emerging software architectures and deep stacks of software dependencies required a coordinated software ecosystem approach. The ECP built a robust HPC software ecosystem that was based around Spack, the E4S, and SDKs. This approach facilitated dissemination of best practices, improved product integration and support for GPUs from three vendors, improved software interoperability and design, and included other collaborative efforts to foster industry partnerships.

Before the ECP, HPC software deployment on GPU platforms typically comprised users installing individual packages using CUDA. With the development of Spack and the E4S in the ECP, we now have a mature, flexible, and comprehensive software stack that targets GPUs from Intel, AMD, and Nvidia on multiple architectures based on the SDK approach.

ACKNOWLEDGMENTS

This research was supported by the ECP (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy (DOE), Office of Science, and the National Nuclear Security Administration.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC (NTESS), a wholly owned subsidiary of Honeywell International Inc., for the DOE's National Nuclear Security Administration under contract DE-NA0003525. This written work is authored by an employee of the NTESS. The employee, not the NTESS, owns the right to, title to, and interest in the written work and is responsible for its contents. Any subjective views or opinions that might be expressed in the written work do not necessarily represent the views of the U.S. government. The publisher acknowledges that the U.S. government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this written work, or allow others to do so, for U.S. government purposes. The DOE will provide public access to results of federally sponsored research in accordance with the DOE's Public Access Plan.

[u]https://openlane.readthedocs.io/en/latest
[v]https://openfasoc.readthedocs.io
[w]https://xyce.sandia.gov

REFERENCES

1. D. A. Beckingsale et al., "RAJA: Portable performance for large-scale scientific applications," in *Proc. IEEE/ACM Int. Workshop Performance, Portability Productivity HPC (P3HPC)*, Denver, CO, USA, 2019, pp. 71–81, doi: 10.1109/P3HPC49587.2019.00012.
2. C. R. Trott et al., "Kokkos 3: Programming model extensions for the exascale era," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 4, pp. 805–817, Apr. 2022, doi: 10.1109/TPDS.2021.3097283.
3. T. Gamblin et al., "The Spack package manager: Bringing order to HPC software chaos," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2015, pp. 1–12, doi: 10.1145/2807591.2807623.
4. T. Gamblin, M. Culpo, G. Becker, and S. Shudler, "Using answer set programming for HPC dependency solving," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, Dallas, TX, USA, 2022, pp. 1–15, doi: 10.1109/SC41404.2022.00040.
5. K. Hoste, J. Timmerman, A. Georges, and S. D. Weirdt, "EasyBuild: Building software with ease," in *Proc. High Perform. Comput., Netw. Storage Anal.*, Salt Lake City, UT, USA, 2012, pp. 572–582, doi: 10.1109/SC.Companion.2012.81.
6. E. Dolstra, M. de Jonge, and E. Visser, "Nix: A safe and policy-free system for software deployment," in *Proc. 18th Large Installation Syst. Admin. Conf. (LISA),* Berkeley, CA, USA: USENIX Association, 2004, pp. 79–92. [Online]. Available: http://dl.acm.org/citation.cfm?id=1052676.1052686
7. E. Dolstra and A. Löh, "NixOS: A purely functional linux distribution," in *Proc. 13th ACM SIGPLAN Int. Conf. Functional Program.,* New York, NY, USA: ACM, 2008, pp. 367–378, doi: 10.1145/1411204.1411255.
8. L. Courtès and R. Wurmus, "Reproducible and user-controlled software environments in HPC with Guix," in *Proc. 2nd Int. Workshop Reproducibility Parallel Comput.*, Vienne, Austria, Aug. 2015, pp. 579–591. [Online]. Available: https://hal.inria.fr/hal-01161771
9. R. Bartlett et al., "xSDK foundations: Toward an extreme-scale scientific software development kit," *Supercomput. Frontiers Innov.*, vol. 4, no. 1, pp. 69–82, Feb. 2017. [Online]. Available: https://superfri.org/index.php/superfri/article/view/127
10. M. Heroux et al. "E4S: Extreme-scale scientific software stack." GitHub. Accessed: Oct. 26, 2023. [Online]. Available: https://collegeville.github.io/CW20/WorkshopResources/WhitePapers/heroux-willenbring-shende-coti-spear-et-al-E4S.pdf
11. M. A. Heroux and J. M. Willenbring, "A new overview of The Trilinos Project," *Sci. Program.*, vol. 20, no. 2, pp. 83–88, 2012, doi: 10.1155/2012/408130.
12. T. Ajayi et al., "OpenROAD: Toward a self-driving, open-source digital layout implementation tool chain," in *Proc. Government Microcircuit Appl. Crit. Technol. Conf.*, 2019, pp. 1105–1110.

JAMES M. WILLENBRING is a senior member of the R&D Technical Staff in the Center for Computing Research and the Software Engineering and Research Department, Sandia National Laboratories, Albuquerque, NM, 87185, USA. His research interests include the research of software sustainability and the application of software engineering methodologies for high-performance computational science. Willenbring received his M.S. degree in computer science from St. Cloud State University. Contact him at jmwille@sandia.gov.

SAMEER S. SHENDE is a research professor and director of the Performance Research Laboratory, University of Oregon, Eugene, OR, 97403, USA, and the president and director of ParaTools (USA) and ParaTools SAS (France). His research interests include scientific software stacks, performance instrumentation, and compiler optimizations. Shende received his Ph.D. degree in computer and information science from the University of Oregon. Contact him at sameer@cs.uoregon.edu.

TODD GAMBLIN is a distinguished member of the technical staff in the Livermore Computing division, Lawrence Livermore National Laboratory, Livermore, CA, 94550, USA. His research interests include dependency management, open source, and software engineering. Gamblin received his Ph.D. degree in computer science from the University of North Carolina at Chapel Hill. Contact him at tgamblin@llnl.gov.