Received: 31 May 2021 | Revised: 19 November 2021 | Accepted: 24 January 2022

DOI: 10.1002/spe.3075

SPECIAL ISSUE PAPER

WILEY

EESSI: A cross-platform ready-to-use optimised scientific software stack

Bob Dröge[1] | **Victor Holanda Rusu**[2] | **Kenneth Hoste**[3] | **Caspar van Leeuwen**[4] | **Alan O'Cais**[5] | **Thomas Röblitz**[6]

[1]Center for Information Technology, University of Groningen, Groningen, The Netherlands

[2]Swiss National Supercomputing Centre, Eidgenössische Technische Hochschule Zürich, Zürich, Switzerland

[3]Department of ICT, Ghent University, Ghent, Belgium

[4]Compute Services, SURF, Amsterdam, The Netherlands

[5]Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, Jülich, Germany

[6]IT Division, University of Bergen, Bergen, Norway

Correspondence
Bob Dröge, Center for Information Technology, University of Groningen, Groningen, The Netherlands
Email: b.e.droge@rug.nl

Abstract

Getting scientific software installed correctly and ensuring it performs well has been a ubiquitous problem for several decades now, which is compounded currently by the changing landscape of computational science with the (re-)emergence of different microprocessor families, and the expansion to additional scientific domains like artificial intelligence and next-generation sequencing. The European Environment for Scientific Software Installations (EESSI) project aims to provide a ready-to-use stack of scientific software installations that can be leveraged easily on a variety of platforms, ranging from personal workstations to cloud environments and supercomputer infrastructure, without making compromises with respect to performance. In this article, we provide a detailed overview of the project, highlight potential use cases, and demonstrate that the performance of the provided scientific software installations can be competitive with system-specific installations.

KEYWORDS

high-performance computing, scientific software, supercomputing

1 | INTRODUCTION

Almost two decades ago, the paper entitled 'Why Johnny can't build [portable scientific software]'[1] was published. It outlined a set of reasons why it is hard to 'create significant, portable scientific software'. One of the main ones highlighted was that many typical scientific packages have a large set of externally developed dependencies, which in turn bring a large set of related requirements for each unique build, and these may be complex to resolve consistently on different systems. This situation is frequently referred to as 'dependency hell', and affects both developers and end users of application codes.

Unfortunately this observation is still valid today, and is exacerbated by a number of additional factors like the expansion of computational science, and the changing landscape of scientific computing infrastructure (see Sections 1.1 and 1.2). We propose, however, that the tools are now in place to overcome this problem in such a way that Johnny or Julie, as an end user of any software application (or dependency), no longer *have* to build scientific software themselves

Abbreviations: EESSI, European Environment for Scientific Software Installations; HPC, high-performance computing.

in order to get an installation that is properly optimised for the advanced capabilities of their hardware: they can simply use an existing, centralised deployment, that is easy to access, available worldwide, and contains optimised installations for a wide range of modern hardware architectures, while at the same time having a minimal storage footprint on the local client.

Effectively building and maintaining such a centralised deployment is a challenging endeavour however, and requires collaboration throughout the scientific community to achieve the best possible outcome. In this article, we introduce the *European Environment for Scientific Software Installations* (EESSI) project,[2,3] which has set out to achieve this ambitious goal. Key components to EESSI are the use of a shared filesystem accessible via the public internet (see Section 3.1) to offer a centralised deployment that is globally available, a compatibility layer (see Section 3.2) to provide the necessary isolation from the host operating system (OS), and a software layer (see Section 3.3) which provides the scientific software installations, optimised for a wide range of hardware capabilities.

1.1 | Expansion of computational science

Over the last number of decades, we have seen tremendous growth in the capabilities of high-performance computing (HPC) systems, with roughly a 1000-fold increase in computational power every dozen years since the Cray X-MP in 1982.[4] Coupled with the advent of revolutionary technologies like graphics processing units (GPUs) which can be used for general-purpose computing, this has fuelled the interest of researchers from an ever-widening variety of scientific domains to leverage this wealth of computational power to drive their research.

Traditionally, HPC systems had primarily been used for research in a relatively limited subset of scientific domains, such as physics simulations, molecular modelling, weather predictions and so forth. In recent years the scope of *computational science* has expanded significantly, with new fields such as machine learning, artificial intelligence, bio-informatics, and others, emerging as prominent user communities. In several cases, the increased interest in HPC was a direct result of a huge influx of available data to be processed or utilised: the so-called *big data* revolution. Prominent examples of this are *deep learning*[5] and next-generation sequencing (NGS).[6]

This broader spectrum of scientific domains seeking to employ computational methods to make sense of the wealth of data, together with the push to make the software that was developed to support scientific research available publicly, has led to an explosion of available scientific software.[7] This not only impacts the support teams of these high-end systems, who often find themselves struggling to keep up with support requests, but also the scientific researchers themselves, since they frequently lack the necessary expertise to efficiently manage the wealth of tools, libraries and applications that they require,[1,8] and sometimes find themselves limited to the software that they can get working in the short term, or waiting until others can facilitate the installation of the tools that they would like to use.

1.2 | Changing landscape of scientific computing infrastructure

To further complicate matters, the landscape of scientific computing has been undergoing significant changes recently in terms of available hardware resources. The vast majority of HPC systems large and small that were installed in the last decade or so have been powered by microprocessors manufactured by Intel or AMD which support the ubiquitous `x86_64` *instruction set architecture* (ISA), to the point where over 90% of systems in the Top500 list of supercomputers fell in that category.[9] Only a small minority of systems were employing processors supporting a different ISA, like PowerPC. This is still mostly true today, but ARM-based processor architectures are already used in top supercomputers such as Fugaku,[10] and are quickly becoming mainstream in scientific computing thanks to increasing adoption by cloud providers and hardware manufacturers. In addition, systems with RISC-V-based accelerators, or even general-purpose processors, are on the horizon, a prominent example being the European Processor Initiative (EPI) project.[11]

Next to major shifts in the diversity of hardware architectures, there is also a growing interest in leveraging the infrastructure that is provided by commercial cloud providers for scientific workloads, either as a replacement for on-premises systems, or as an additional resource to do *cloud bursting* when the on-premises infrastructure is insufficient to keep up with the demand for compute resources.

This increased variety in available compute resources has a significant impact on the effort that is required to migrate workloads from one system to another: software that was compiled for an ISA like `x86_64` must be recompiled to

run on processors that support another ISA. This raises concerns about established practices like software packaging (e.g., RPMs), or containerizing software to achieve so-called 'mobility of compute', which either break down in this new context, or at best require lots of additional effort to allow for installing and using the software on systems with different ISAs.

1.3 | Sacrificing performance

In fact, sacrifices are already being made to facilitate the current situation even when focusing on a single ISA like `x86_64`. Typically the range of systems on which pre-packaged software or containers can be used is purposely maximised by compiling the software included in them *generically*, that is, by not relying on specialised subsets of the ISA like vector instructions (AVX, AVX2, AVX-512, etc.) which are only compatible with sufficiently recent generations of microprocessors. This comes at a potential cost in performance however, especially for scientific software in which specialised instructions that steer specific functional units in microprocessors typically result in significant speedups.

To illustrate this point, Figure 1 shows the performance of running a molecular dynamics simulation with GROMACS,[13] and highlights the impact of using a GROMACS installation that was compiled with support for (only) specific (sub)sets of vector instructions. These results quantify the potential impact of using *generic* binaries on modern microprocessors that support different generations of vector instructions, as is the case on Intel microprocessors of the Cascade Lake generation which support the SSE 4.1, AVX, AVX2, and AVX-512 sets of vector instructions. When using a generic GROMACS binary that only uses SSE 4.1 vector instructions, which will run on basically any modern day Intel or AMD microprocessor, we observe a performance of about 1.18 ns/day (see the left bar in Figure 1). Using a binary that utilises AVX or AVX2 instructions already results in a significant speedup over the generic binary with 1.62 and 1.63 ns/day (middle bars in Figure 1), respectively, equivalent to a speedup of about 38%. When using a binary that was compiled for this specific generation of Intel microprocessors that uses AVX-512 instructions (and all previous generation vector instructions like SSE 4.1, AVX, and AVX2), which will only run on systems powered by Intel microprocessors of this generation that support these advanced vector instructions, we achieve 1.86 ns/day (see the right bar in Figure 1). This is 57% faster compared to the generic binary, and about 13% faster than the binaries utilizing only AVX and AVX2 instructions. We would like to stress here that the GROMACS source code itself was not modified in any way: the only difference is the configuration option which controls the vector instructions that can be generated by the compiler.

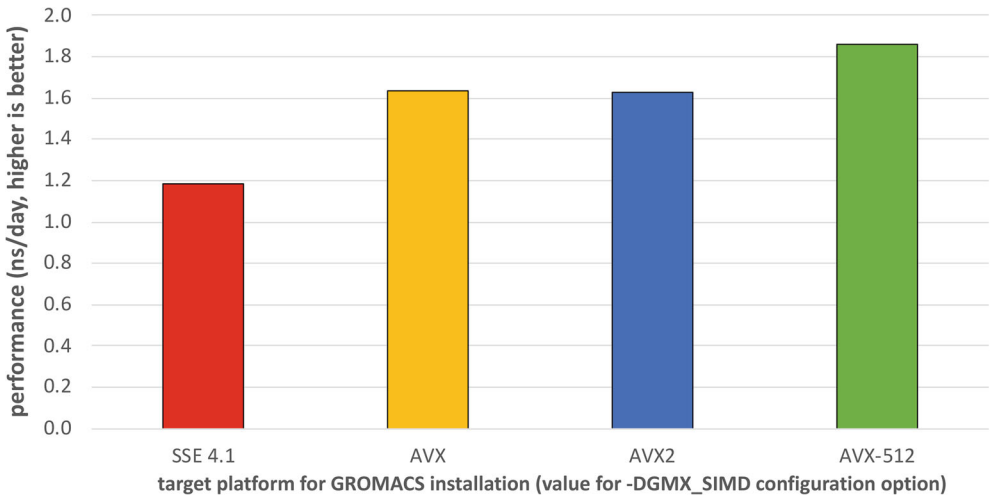

FIGURE 1 Performance for a molecular dynamics simulation with GROMACS, quantified as the amount of nanoseconds of simulated time per day of running the simulation (abbreviated as ns/day, higher is better), for GROMACS version 2020.4 on an 2x 18-core Intel Xeon Gold 6240 (Cascade Lake) system, when using a GROMACS binary that was compiled to leverage different sets of vector instructions supported by the `x86_64` family of microprocessors. The input file was obtained from the PRACE-UEABS benchmark suite version 2.1 (Test Case B)[12] (which is also used in Section 6)

One potential approach to create binaries that are both portable and make the best use of advanced instruction sets is to create so-called *fat binaries*: binaries that contain multiple code paths, for example, one optimised for SSE 4.1, one for AVX and one for AVX2. At runtime, the right codepath is selected based on the hardware capability of the host. However, currently, only the Intel compilers can create such fat binaries for CPUs, while the more commonly used GNU compilers cannot. Moreover, the optional code paths only work on Intel CPUs: when such binaries are run on non-Intel CPUs, the baseline code path (e.g., SSE 4.1) is used. Finally, fat binaries only resolve the issue of using optimised instruction sets *within* a single ISA, they don't create portability between ISAs. These aspects strongly limit the scope of this solution.

Of course, GROMACS is just one particular example, and the performance impact will vary wildly across different use cases, scientific software, scientific domains, compute infrastructure and so forth. Nevertheless, to achieve the best performance, it remains vital to use software that was compiled to fully leverage the capabilities of the microprocessor on which the software will be run.

1.4 | The European Environment for Scientific Software Installations

The EESSI project is a collaboration between different HPC sites and industry partners, with the common goal to set up a shared repository of scientific software installations that can be used on a variety of client systems, regardless of which type, flavour or version of operating system is used, which type of microprocessor they are powered by, and whether it is a full-size HPC cluster, a virtual system in the cloud, or a personal workstation.

With this project, we aim to relieve scientific researchers from the burden of installing the tools, libraries, and applications they require for their work. Our (ambitious) overall goal is to provide easy access to a consistent stack of scientific software across different types of client systems, while avoiding compromises that would have a significant effect on the performance of the provided software installations.

To achieve this, we combine multiple established existing open-source tools to build a solution consisting of three layers. Each layer is responsible for one particular aspect: distributing the software installations, ensuring compatibility with different types and versions of client operating systems, and providing installations of scientific software which are optimised for different families of microprocessors.

1.5 | Relevance to extreme-scale computing

At the hardware architecture level, leadership-class HPC resources are usually innovative in a number of potential ways: CPU, accelerator, I/O, interconnect, and any novel combination of these. The software ecosystem used by researchers is, however, so broad that it can take several years before many scientifically significant workflows are fully ported to these systems. Researchers given access to novel resources frequently experience 'dependency hell' when trying to port their application. They can spend enormous effort adapting their specific software stack requirements to a new system before they even begin to consider their own application. This issue is even more pronounced when one considers workflows where they may be using multiple high-level applications, each with their own dependency tree. The majority of this effort, and the knowledge that is generated as a result, is frequently not documented nor actively shared with other research communities.

Tools such as EasyBuild[14,15] and Spack[16] can help address this situation, but they both require that you must make yourself somewhat familiar with the tool itself. In addition, there still is a lot of duplicated effort in terms of dealing with the problems that inevitably arise despite using these well-established tools. EESSI, however, provides the *actual software installations themselves*, optimised for specific system architectures. It does so in a consistent way, such that the software stack that is available on a system will be almost identical to the one you find on another system, in terms of available software applications and versions.

Software developers of scientific software who leverage EESSI will be able to develop on their local system using a software stack that can also be found on the largest exascale systems (and everything in between), spending their time primarily on their own application and not on its dependency tree.

Expanding upon this approach, EESSI also has the capability to enable continuous integration workflows at a level previously unheard of for HPC resources. With EESSI already having support for Intel, AMD and ARM processors, and actively seeking to support POWER and RISC-V processors in the future, the possibilities in this space are numerous.

Introducing such structure to the development and deployment process is also of tremendous value not just to developers but also to the end user communities of application codes. If an application code is supported by EESSI, they will find it easy to transition between whatever compute resources they have access to.

The remainder of this article presents current practice in Section 2, gives a detailed overview of the EESSI project in Section 3, outlines multiple different use cases enabled by EESSI in Section 4, demonstrates how to get access to the proof-of-concept EESSI pilot repository in Section 5, and evaluates the performance of one particular scientific software package (GROMACS) provided through EESSI in Section 6. In Section 7, we discuss the challenges and limitations of the EESSI project, related work is discussed in Section 8, future work is outlined in Section 9, and we conclude in Section 10.

The contributions of EESSI in the context of this article can be summarised as:

- the provision of a rich stack of scientific software that is compatible with, and optimised for, a broad range of systems;
- providing easy access to included software across a broad range of platforms, including HPC systems, personal workstations and cloud environments;
- the automated selection of the best suited software stack for the CPU microarchitecture of the client system;
- a user-friendly way of globally distributing scientific software installations;
- a centrally managed software stack that is open to contributions from the scientific community;
- facilitating correctness, performance, and scalability testing of included software;
- describing multiple use cases that are supported or enabled by EESSI;
- a preliminary evaluation of leveraging EESSI to run a molecular dynamics simulation on a large-scale system;
- a critical discussion on the challenges and limitations of the current design choices of the EESSI project.

The article may, in parts, use technical jargon or terms and reference applications and tools that are perhaps familiar to an HPC domain audience, but may be unfamiliar to a wider audience. For this reason, we include a glossary of terms used in Appendix C and an overview of software and tools referenced in Appendix D.

2 | CURRENT PRACTICE

Getting scientific software installed has been a challenge for several decades, and a multitude of tools have been developed to help deal with this ubiquitous burden, which come with varying tradeoffs.

2.1 | The peculiarities of scientific software

Before discussing these tools, we want to highlight a couple of important aspects of scientific software that should be kept in mind.

As already mentioned in Section 1, scientific software stacks are typically *complex*, and involve lots of externally developed dependencies, including a variety of low-level libraries that implement specific functionality that is commonly used in scientific software. Examples of this are libraries like Open MPI[17] and MPICH[18] that implement the message passing interface (MPI)[19] standard that is often used by software that supports distributed parallel computing, and libraries like OpenBLAS,[20] BLIS,[21] and the Intel Math Kernel Library,[22] which collect a set of low-level routines for linear algebra operations as specified by the Basic Linear Algebra Subprograms (BLAS)[23] and Linear Algebra Package (LAPACK)[24] standard interfaces. A complicating factor is that libraries within the same family, for example those implementing the MPI standard, or even different versions of the same library, generally do not have a compatible application binary interface (ABI), which means that you cannot easily swap one library for another, unless the software using that library is rebuilt from source. In addition, it is quite common to use multiple different (versions of) compiler suites to build scientific software, based on the support for specific programming languages (like Fortran, or recent versions of the C++ standard) and the quality of the binaries that are produced on a specific hardware platform. This leads to several disjoint groups of compatible software installations (with installations from different groups *not* being compatible with each other), because they are built with a particular (version of a) compiler suite, and/or built on top of the same low-level libraries.

The *performance* of the software installations that are employed is often very important, especially when running large-scale simulations on bleeding-edge hardware, since even a speedup of just a few per cent can be equivalent to a significant reduction in energy cost required to complete a workload, and allows other workloads to use the saved resources.

The *expansion of computational science* we discussed in Section 1.1 results in a much more diverse set of scientific software applications. Scientific researchers from domains that have only recently started running their workloads in complex environments like a supercomputer or a system that employs accelerators like GPUs often lack extensive expertise in getting the most out of these resources.

The *multi-tenant nature* of HPC infrastructure, where the resources are shared between a (potentially) large and diverse group of users, implies that multiple different versions and configurations of the same software may be required to be available side-by-side. This is because different researchers will have different requirements or preferences: some may want to use stable (older) software versions, while others may prefer the very latest updates. Similarly, requirements regarding how applications are configured (e.g., whether specific optional features are enabled, or not) may vary for different researchers and use cases.

It is common that end-user software applications need to be *integrated* with key software components (e.g., specific libraries, drivers, or kernel modules) that are provided by the vendor of specialised hardware components like network interconnects (e.g., InfiniBand) or accelerators (GPUs) in order to obtain optimal performance, which further complicates the installation process for scientific software.

2.2 | Traditional package managers

Making complex software easy to install is a desire that is not at all unique to scientific software: it is a common aspect of any software-driven technology. In a general context several well-established *package management* systems for software and accompanying tools are employed to help facilitate the management of large complex software stacks. Well-known examples on Linux operating systems include RPM packages and tools like `yum` or `dnf` to install them.

Although these traditional package managers are a key component in managing the software on modern systems, they often lack extensive support for most of the peculiarities of scientific software we discussed in the previous section, like having multiple different (and very recent) versions/configurations of a software application installed at the same time. Updating installed software packages with a traditional package manager implies *replacing* existing software installations with new versions. In a broader context, only the most recent software version is generally useful, and old software versions should be uninstalled with security in mind. This is very important for software that typically runs with elevated privileges, but conflicts with the multi-tenancy aspect of HPC systems where different users have different needs regarding available versions of (scientific) software, most of which do not need elevated privileges at all.

In addition, the software installations that these packages provide are typically deliberately built *generically* to maximise their use across systems both old and new. As discussed in Section 1.3, this can have a significant negative impact on the performance of the software on modern hardware, especially for scientific software, which goes directly against one of the key objectives when using scientific software on high-end supercomputing systems.

Moreover, traditional package managers are primarily intended for managing a system-wide software stack, and hence usually require administrator privileges to install software packages. This severely limits their applicability for scientific researchers wishing to install the software they require, since they often lack the permissions required to install packages system-wide, especially on HPC systems that are fully managed by a dedicated team of system administrators.

2.3 | Central software stack and environment modules

As a result of their specific features and design choices, traditional package management systems are less suitable for managing scientific software on HPC infrastructure. Therefore, other methods and tools have been developed for this purpose.

Although there are exceptions, a centrally managed software stack that includes both special-purpose libraries (MPI, BLAS, LAPACK, etc.) and a broad collection of end-user scientific software applications is typically available on HPC

systems through a shared filesystem, which is accessible from both login nodes and compute nodes. The software provided this way is usually accessed via an *environment modules* tool[25] like Lmod,[15,26] where specific software installations can be 'activated' by *loading* a so-called *module file*, which corresponds to making the changes to the session environment that are needed to make that software ready to use. This way, multiple different versions or configurations of a single software package can be installed side-by-side, and the needs of multiple users with conflicting requirements can be fulfilled, while providing easy access to the available software.

Maintaining a central software stack can be a daunting task for the HPC support team however, depending on the variety of the scientific researchers that use the infrastructure, both in terms of scientific domain and expertise. It is not uncommon that this involves managing hundreds or even a couple of thousand of different software installations, where a significant portion consists of complex software that is challenging to install correctly while ensuring it also performs well.

In recent years, tools like *EasyBuild*[14,15] and *Spack*[16] that facilitate maintaining a stack of scientific software have seen wide-spread adoption in the HPC community, both in terms of usage and active development, to the point where not using a tool like these has become a daunting prospect.

Although these tools foster collaboration across HPC sites worldwide, the effort of maintaining a stack of scientific software remains substantial. In practice, it still takes significant time and expertise to become sufficiently familiar with these tools in order to efficiently exploit them to their full capabilities, which can result in a select group of people who are effectively managing the central software stack. As such, a large opportunity to further reduce the required effort and for more extensive cross-site collaboration remains.

In addition, although tools like EasyBuild and Spack significantly reduce the burden of maintaining a central software stack, there is still a lot of duplicated effort since each HPC site constructs their own central software stack, which is usually only available on, and useful for, their specific HPC infrastructure. A broader collaborative effort that employs these software installation tools, but aims to cater to a wider set of platforms, would be very beneficial to the HPC community.

2.4 | The Conda package management system

Conda[27] is an open-source package management system, which is particularly popular in the scientific community. It supports creating, using, and managing one or more *environments* that correspond to sets of software packages that are used for specific use cases or projects. Conda was originally created to facilitate working with tools and libraries implemented in Python, but has gradually expanded its scope to also cover software implemented in other programming languages. Next to packages that are available through the standard Anaconda *channel*, additional channels like *Bioconda*[28] that cater to specific scientific domains can also be used for installing packages in environments.

Conda empowers scientific researchers to self-manage the software stack they require, without requiring administrator privileges to create environments and install packages in them. Using it comes with a number of trade-offs however, which are largely similar to those discussed for traditional package managers in Section 2.2. Software packages that can be installed via conda are usually built generically such that they can be used on a wide range of different CPU generations which may severely impact performance, especially for scientific software (see Section 1.3).

Since conda is primarily intended for letting *individuals* manage their own software stack, it is not well suited for managing a *central* software stack in a multi-tenant environment. In addition, conda installs software packages (and additional metadata like a cache of downloaded packages, etc.) in the user's home directory (at least by default). This is often problematic on HPC systems due to the enforcement of strict limitations regarding available disk space and number of files that can be stored in the home directory. These restrictions are put in place to avoid excessive use of that home directory, which primarily serves as an entry point to the system, and to avoid imposing significant load on the infrastructure, for example, by having a significantly large (set of) software stack(s) installed there that is actively used across a large-scale HPC cluster, since this goes well beyond the intended usage of a home directory.

Combining software installations that are available already available system-wide or through a central software stack with a set of packages installed via conda frequently leads to problems, due to different assumptions and design decisions that were made for these different software stacks. Some software that is eagerly installed by conda, like,

for example, Python, MPI implementations like Open MPI, or the OpenSSL library, is often not properly configured for a specific system, or even conflicts with already available installations. It also contributes to the amount of software packages that get pulled in when creating conda environments, which can significantly complicate managing them. This often leads to end-users of HPC systems involuntarily shooting themselves in the foot when trying to self-manage their software installations using conda, mostly due to being unaware of how to best get scientific software installed while taking into account the specifics of the HPC infrastructure regarding hardware resources and software environment.

2.5 | Containers

Another practice that has seen widespread adoption in recent years to help deal with complex (scientific) software applications is the use of *containers*, where a *container image* that includes everything that is needed to run a particular (set of) software application(s) is launched via a *container runtime* tool. This technique has been used for a while in enterprise environments, and has recently also become popular in the HPC community through projects like Singularity,[29] which supports features that are particularly important on HPC systems, like integration with specialised hardware such as high-speed interconnects and GPUs, and the lack of a central daemon that runs with elevated permissions (which is a security concern on HPC systems).

The use of containers for running scientific software on HPC systems has become quite popular, yet it comes with several shortcomings and tradeoffs.

In some sense, container images are just a static bundle of prebuilt software packages, so most of the points raised in Section 2.2 also apply here. For example, container images usually contain generically optimized binaries, so that they can be used across a wide range of systems. This implies sacrificing performance for so-called *mobility of compute*, as we discussed in Section 1.3.

In addition, container images for even a moderate software stack can become quite big (several gigabytes is not uncommon). This quickly becomes problematic for large software stacks, or when a lot of different container images are used (e.g., in a multi-tenant environment), or when the image is used across a range of different systems (HPC clusters, workstations, cloud resources, etc.).

Moreover, efficiently *building* container images for complex software stacks remains a problem. Although tools like EasyBuild and Spack can be used here too, short-term solutions involving manually running commands or using crude scripts are often employed when constructing container images, which raises questions about reproducibility, maintenance in case of software updates and so forth. Container images are also very static, in the sense that making changes to them, like, for example, installing additional software, or updating an installed software package, can be challenging depending on how the original container image was constructed.

Although technical solutions for most of these problems already exist or can be (and are being) developed, it seems like containers are currently mostly used as a *workaround* for the challenges that arise when managing complex scientific software stacks, as opposed to being a complete solution to these challenges.

2.6 | CernVM-FS

In the context of grid computing (where the compute resources are widely distributed), *CernVM-FS* (CernVM File System) is a common tool used in particular by the high-energy physics community. CernVM-FS is a read-only, globally distributed filesystem that is optimized for distributing software.[30,31] It was developed in the context of the Large Hadron Collider (LHC) research project[32] and the Worldwide LHC Computing Grid (WLCG),* to efficiently distribute application software across LHC project members worldwide.

A CernVM-FS repository contains actual software installations, not packages that contain software like a traditional package management repository we discussed in Section 2.2. A system that mounts a CernVM-FS repository offers access to the software stack provided by that repository as if those installations were available locally on the system. Specific features implemented by CernVM-FS, such as compression and deduplication of data, aggressive caching, and on-demand

*See https://wlcg.web.cern.ch/

downloading of files and file metadata, cater to the specific use case of storing and globally distributing software while optimizing the required disk space and network bandwidth.

CernVM-FS has seen strong adoption in the High Energy Physics (HEP) community over the last decade, but much less so in the broader HPC community (with some exceptions, see Section 8.1), despite its focus on software distribution for large-scale systems.

An interesting recent development in CernVM-FS is the support for ingesting container images into a CernVM-FS repository,[†] which is done to gain easy access to large collections of existing container images while exploiting CernVM-FS features like data deduplication and compression that facilitate distribution, thus limiting the impact of using large container images.

While CernVM-FS is focused on facilitating the global distribution of software installations, the software available in a CernVM-FS repository is often specific to a certain environment, like a particular virtual machine image or operating system version. This significantly limits the applicability of these software installations, since despite their global availability they can only be employed easily on specific systems.

We see an opportunity for wide-spread adoption of CernVM-FS in the HPC community as well, since it tackles a key problem in managing HPC systems in a performant and scalable way. To effectively see broad adoption however, CernVM-FS repositories that provide scientific software stacks that were built for HPC systems (and beyond) need to be available first, and take into account critical aspects like the variety of HPC system configurations, the importance of the performance of those installations, and integration with special-purpose hardware, as discussed in Section 2.1.

As a key component in EESSI, CernVM-FS is discussed in more detail in Section 3.1.

3 | PROJECT OVERVIEW

One of the main goals of the EESSI project is to easily make the scientific software stack available to a large variety of operating systems and hardware, including different CPUs, accelerators (such as GPUs), and network interconnects. This is not limited to HPC systems, but can also be, for instance, a personal workstation or a virtual machine in a cloud environment. Aiming to support such a wide range of systems means that we cannot, and do not want to, make too many assumptions about the machines where the software stack will be used. It also requires the software stack to be self-sufficient in terms of dependencies like operating system tools and libraries, and flexible with respect to how it integrates into the machine where it will be used.

In order to achieve this ambitious goal, the project has been split into three inter-operating and stacked layers that each serve a different purpose. We will first introduce the concepts and purposes of these layers.

For this, let us assume that a scientific researcher wants to use the EESSI software stack on a system they have access to, which we refer to as the *client system*; again, this can be any kind of system. As illustrated in Figure 2, this client system has to provide some operating system itself; for now this has to be a Linux distribution, and it can also be run under the Windows Subsystem for Linux. In the future we still hope to support client systems running macOS as well. If applicable, drivers for (special-purpose) hardware such as accelerators and interconnects also have to be provided by the client system, and it could also provide additional tools like a resource manager (e.g., Slurm). As we will show later on, EESSI ships its own version of the `glibc`[33] library, which has some implications for the minimum kernel version of the client system. Therefore, we plan to develop a script that does some compatibility checks regarding kernel and driver versions, and picks up the driver libraries from the host.

On top of what the client system provides, EESSI first adds a *filesystem layer* that is responsible for transparently distributing the software installations provided by the EESSI repository from centrally managed servers to this client system. The implementation of this layer should allow researchers and other users of the stack to easily get access to the EESSI software stack on their systems.

The second layer is the *compatibility layer* that provides operating system libraries that are required by the scientific applications. This layer is crucial to ensure that the software installations provided by EESSI work on different operating systems (or versions thereof), as it alleviates depending on libraries provided by the operating system of the client. Without this layer we would have to make very specific assumptions about, or add very specific requirements for, the client system.

[†]See https://cvmfs.readthedocs.io/en/stable/cpt-containers.html

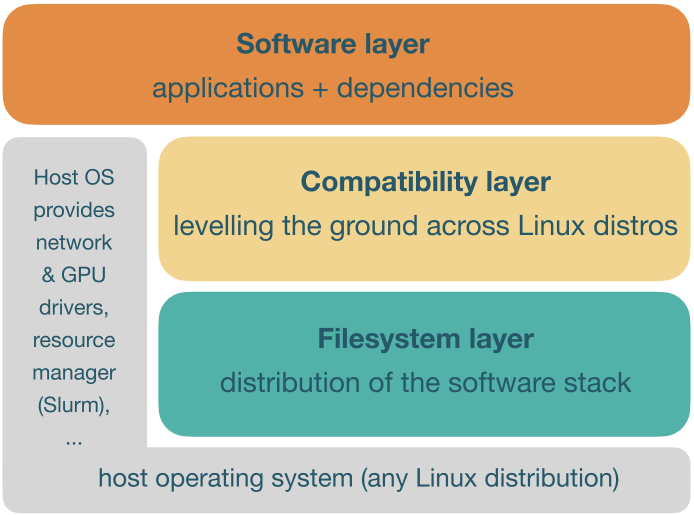

FIGURE 2 Overview of the layered design of EESSI

Finally, the third layer is the *software layer* consisting of the actual scientific software installations; these are the applications that the end users will be using, and which are optimised for different generations of microprocessors. This layer should provide the users of the stack with easy access to the actual scientific applications, and preferably we need a tool that allows us to easily install all these applications in a uniform way and optimise them for different architectures.

In the following subsections, we will outline these layers in more detail and describe the open source tools that we have chosen to use to construct them. We will also provide more details about the procedures that we use for building software, releasing versions of the software stack, deployment, and automation, and finally, testing the various components.

3.1 | Filesystem layer

The purpose of the filesystem layer is to distribute the EESSI software repository to, potentially, an enormous number of clients around the world. We note that software installations have specific characteristics such as often involving lots of small files which are being opened and read as a whole regularly, frequent searching for files in multiple directories, hierarchical structuring and so forth. Given these characteristics, we have chosen to use CernVM-FS[30] in our filesystem layer.

CernVM-FS is a software distribution service developed by CERN to make high energy physics software available to worldwide-distributed computing infrastructures, and which is heavily tuned to cater to this specific use case of distributing software installations. Files in CernVM-FS repositories are only pulled in to the client machine whenever they are accessed by that client. It uses aggressive caching and reduces access latency by, for example, automatic file de-duplication and compression.

CernVM-FS is a network filesystem, which you can mount in Linux or macOS via FUSE (Filesystem in Userspace), and on Windows in a WSL2 virtualised Linux environment. The installation of the CernVM-FS client, required for any user who wants to access the EESSI stack, is generally a very easy process, as installation packages are provided for many different platforms.

In some ways CernVM-FS is similar to other network filesystems like NFS[34] (Network File System) or AFS[35] (Andrew File System), but there are a number of aspects to it that are considerably different. The files and directories that are made available via CernVM-FS are always located in a subdirectory of `/cvmfs`, and are provisioned via a network of servers that can essentially be viewed as web servers since only outgoing HTTP connections are used. This approach makes it easy to use CernVM-FS in environments that are protected by strict firewall rules or when the use of an HTTP proxy is required.

CernVM-FS is also a *read-only* filesystem for client systems that access it; only those who administer it are able to add or change its contents. Contents are only added at the central *Stratum 0* server, which can be considered

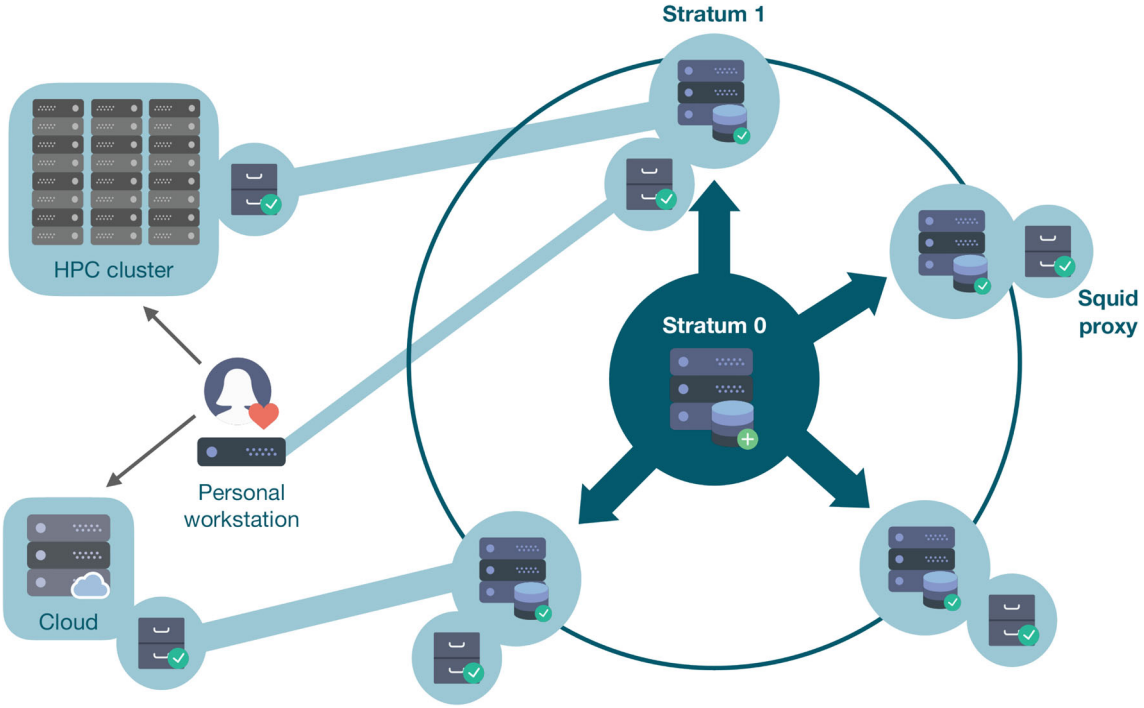

FIGURE 3 Overview of the CernVM-FS components used in EESSI (icons from Smashicons[36])

to hold the master copy of the contents (cf. Figure 3). Internally, CernVM-FS uses content-addressable storage and Merkle trees in order to maintain file data and meta-data, but the filesystem it exposes is a standard POSIX filesystem.

Figure 3 illustrates the main components of CernVM-FS used in EESSI: a central Stratum 0 server hosts the EESSI repository, several Stratum 1 servers hold a complete replica of the repository, proxies take care of caching data, and clients are the systems where the repository is mounted under `/cvmfs`. The Stratum 1 mirror servers are geographically distributed to ensure that clients can fetch data (i.e., the files that they are actually accessing) from a Stratum 1 server that provides low data download latency. To further decrease this latency, clients themselves keep a cache with frequently accessed files, and one or more nearby Squid proxies may be used for adding another, shared, cache layer.

3.2 | Compatibility layer

In order to support many different client operating systems, EESSI provides its own set of operating system libraries and tools which are leveraged by the provided scientific software installations in the software layer. This ensures that the software does not depend in any way on libraries provided by the host operating system on the client.

Because these operating system dependencies have to be shipped with the stack, we need a mechanism to basically install a minimal operating system into a non-default location (under `/cvmfs`). This is one reason why we have chosen Gentoo Prefix[37] for building the compatibility layer, as it is a Linux distribution built from source that can be installed in any given path (the 'prefix'). Furthermore, it already supports many different ISAs, including x86_64, Arm, and POWER, and can be used on both Linux and macOS systems, which is another important reason why Gentoo Prefix is a very good match for the EESSI project. In general, Gentoo Prefix provides users with a way to install their own Linux distribution that they can fully control and modify to their needs with independence from the host operating system, even on a system where they do not have special privileges. Other potential options for the compatibility layer included Nix[38] (and GNU Guix[39] which is based on the Nix package manager), which had previously been evaluated for this purpose by Compute

Canada,[40] though they also currently use Gentoo Prefix (see Section 8.1 for more on the collaboration between EESSI and Compute Canada).

The Gentoo Prefix installations for EESSI are installed into a `compat` subdirectory of the CernVM-FS repository, and we provide one for each ISA. More details about the directory structure can be found in Appendix A.

The default Gentoo Prefix installation contains a quite minimal set of tools and libraries, including, for instance, `glibc`, `OpenSSL`, and compilers and build tools. All these tools and libraries are built from source using Gentoo's package manager Portage, which usually provides very up-to-date versions of the packages. This not only ensures that we can provide recent versions of these packages, but also that we can quickly update in case of security vulnerabilities.

On top of the default Gentoo Prefix installation, we add some additional packages that we need for our software layer, like communication packages such as `rdma-core` and the environment modules tool `Lmod`.[15,41] A very important aspect here is that all packages in the compatibility layer provide only generic binaries, as the packages in this layer are not the ones that are critical for the performance of the scientific applications, and this allows them to be reused across different CPU microarchitectures. If an (operating system) library is needed that does have a significant impact on the performance, we will install this package in the software layer instead. There may be some corner cases for certain applications or libraries, and some could even be installed in both layers: for instance, even though the compatibility layer already provides the GCC compilers, we need more control over how it is compiled and which exact version(s) are available for the scientific applications in the software layer, and therefore we install additional versions in that layer.

In order to ensure that the Gentoo Prefix installations are capable of correctly resolving entities like usernames, groups, and hostnames, and detecting the correct timezone of the client, we have to make some additional patches. Many of these issues can often be solved by replacing files in the installation by symbolic links to the same file on the client system, so that the corresponding settings are inherited from the client.

3.3 | Software layer

The software layer of the EESSI project holds the actual scientific applications, and is what most users will be interacting with. In practice, the software layer is a subdirectory of EESSI's CernVM-FS repository, with different subtrees for all the different microarchitectures that are natively supported, as can be seen in the repository's directory structure in Appendix A.

In the current implementation, we have chosen to use the EasyBuild installation framework to build, optimise, and install each scientific application or library for every supported microarchitecture by running it on appropriate build nodes with the corresponding microarchitecures.

Building software with a non-standard system root directory, which is the case with our Gentoo Prefix installations, can often lead to issues with software picking up files from the standard system root directory. However, EasyBuild already actively supports building with a non-standard system root directory, and provides patches for applications with hard-coded paths to the standard system root directory. This feature makes EasyBuild a perfect candidate to use on top of a Gentoo Prefix installation.

EasyBuild also takes care of generating the module files that are used in conjunction with Lmod to present the end user with a software module environment, allowing them to load the modules that they need into their environment. Regardless of the microarchitecture of the end user's system, they will always see the same software tree that is optimised for their hardware. The only differences between these software trees could be related to applications that are not supported on certain (micro)architectures; in this case they will be missing in the corresponding tree(s).

As selecting the right software tree, that is, the one that is best optimised for their hardware, may be a burden for a lot of end users, this is taken care of by leveraging the *archspec*[42] tool. It allows for detecting a machine's microarchitecture, and in case there is no native support for it within the EESSI software stack, it can query for the best matching compatible microarchitecture that is supported by EESSI.

By sourcing a shell initialisation script that is provided in the repository, or potentially just loading a module file, this process is automated: the paths to the right software tree automatically get reflected into the user's environment. In other words, the user only has to run a single command to get access to a full-blown scientific software stack that is optimised for their hardware.

3.4 | Software build procedure

By design, CernVM-FS only allows you to add new files to the repository on the main repository server (Stratum 0), or on dedicated publisher machines that are allowed to modify the repository. Configuring a machine for being a publisher node does require root privileges. We, however, typically want to do all the software builds without root privileges since this would make it possible to use any cluster node or virtual machine as a build node for the corresponding microarchitecture.

Our current solution is to split the building and publishing. The entire build is done inside a Singularity[29] container, to make sure that we have a controlled and isolated build environment, and to allow for building without root privileges. In order to minimise the chances of picking up some library on the host, which is especially relevant here due to non-standard root directory that we have, we keep the number of installed packages inside this container to a minimum. Also, as mentioned previously, EasyBuild should greatly help in preventing host libraries from being picked up by the build process.

Finally, we need to make sure that we can install the software into its final location (i.e., into `/cvmfs`) at the end of the build process since relocating it later on could potentially break the software. The CernVM-FS mount point is, however, read-only so we work around this by using Singularity's FUSE mount feature in combination with `fuse-overlayfs`[43] to add a writable overlay on top of the mount point. This gives the build process the impression of being able to write to the repository, while the actual installation ends up in an `upper` directory of the overlay on the host. By making a tarball of this directory, we can easily ship the installation from the build node to the node where we can ingest it to the repository.

3.5 | Versioning and releases

As the EESSI project consists of several components and layers, versioning and releases can be done at various levels. For instance, the filesystem layer mostly consists of having CernVM-FS infrastructure available. For configuring CernVM-FS clients – note that every EESSI end user will require such a client – we provide a configuration package, which has its own versioning and releases; these can be found on the GitHub repository of this layer.

However, the main release of EESSI will concern the actual software stack itself, which basically consists of the software layer and the compatibility layer, as all software installations depend on the latter. By having versioned releases of the software stack, we have the flexibility to easily start a completely new combination of compatibility and software layer, for instance when we want to do a (major) update of the packages in the compatibility layer. Furthermore, this allows us to retire old versions. Therefore, we currently make versioned directories at the root of the EESSI CernVM-FS repository that contain the year and month of the (initial) release of this software stack, for example, `2021.06`. In this directory, the compatibility layers for the different architectures and the software trees for the different micro-architectures can be found.

It is important to note that the content of a release is not immutable, that is, it *can* be altered after making the release available. While we do try to keep the number of changes to the compatibility layer of a certain release to a minimum in order to reduce the chances of breaking applications, we will have to deal with security vulnerabilities in this layer. In such cases, we have to update the affected package(s) in order to mitigate the vulnerability, as we will discuss in more detail in Section 7.3. Other than this, we will postpone updates of other packages to the next release of the entire repository. For reproducibility purposes, we can still make older versions of a release available to use. CernVM-FS offers a way to let clients choose specific revisions of the repository, allowing them to go back to an earlier revision, at their own risk. Furthermore, we can still provide older versions in alternative forms, for example, by dumping the contents of a specific release into container images.

The software layer of a release, on the other hand, will (have to) be updated, or rather extended, much more frequently. New software installations will be added continuously to keep up with new software releases and requests from our end users.

For our current pilot repository we make new releases every few months. This involves reinstalling compatibility layers with up-to-date packages, and reinstalling all the software stacks. We also remove older pilot versions quite soon after a new one has been released.

For the production stacks, we still have to establish a clear policy. We currently envision that we will do a full new release once or twice a year, while old releases will only be retired after some years. As reproducibility is vital to science, we are working on alternative ways of retaining and providing access to retired installations, for example, via containers or tarballs.

3.6 | Deployment and automation

For the deployment and installation of the three layers and related components, we make extensive use of automation, configuration, and deployment tools. Many parts of these layers are quite complex and prone to subtle mistakes; automation takes humans out of the loop as much as possible, and prevents us from making such errors.

The CernVM-FS infrastructure, which is part of the filesystem layer, consists of several components that have to be configured. For all of them, ranging from clients to servers, we offer Ansible[44] playbooks, which heavily rely on the `ansible-cvmfs` role provided by the Galaxy Project.[45] For instance, sites that want to host a full replica of the EESSI repository, can set up their own CernVM-FS Stratum 1 server, which can be easily configured by running the playbook `stratum1.yml`.

For some of the (long-running) CernVM-FS servers, for example, one running in Amazon AWS, we also leverage Terraform[46] to spin up and prepare the actual machines themselves. This ensures that the configuration of the machines is clearly defined, and the machine can be easily recreated if necessary.

In the compatibility layer, we also heavily rely on an Ansible playbook to automate the entire installation of this layer. This playbook does the regular Gentoo Prefix installation, and adds all our customisations in terms of additional packages and patches. The build nodes for the compatibility layer, one for each architecture, can again be easily started using an infrastructure deployment script based on Terraform.

Finally, the software layer still requires further automation. The software trees are currently built by launching a build script inside a Singularity container. This script contains all the EasyBuild configuration and commands for generating the entire software stack. We are currently focusing on automating the deployment of the software layer by automatically spinning up build nodes using Terraform, launching the builds, storing tarballs of the software installations in a remote location, and ingesting them semi-automatically after this has been checked and approved.

3.7 | Testing

While software testing is important to individual HPC centres that offer software installations to their users, it is exponentially more important in a project like EESSI: even if only a handful of HPC centres would adopt EESSI to provide software installations, the impact of a (breaking) change in the software stack could affect a very large number of end-users.

3.7.1 | The test suite

Following modern software practices, the EESSI project aims to test each individual component (filesystem, compatibility and software layer) and follow a proper continuous integration and continuous deployment (CI/CD) process. The goal is to enable running these tests on any of the systems that provide the EESSI software stack. Additionally, the test suite aims to: (i) be easy to run on any system that mounts the EESSI software stack; (ii) run within reasonable time; (iii) test single node functionality; (iv) test multi-node functionality and multi-node performance (scaling), where applicable; and (v) test *all* the supported end-user applications.

The three main use cases for the EESSI test suite: (i) run a CI/CD pipeline when changes are made in the compatibility layer; (ii) run a CI/CD pipeline when additions are made to the software layer; (iii) testing the software stack when the EESSI stack is deployed on a new client system.

The first use case, changes in the compatibility layer, will require testing of all software in the software layer. In order to limit the computational footprint of this pipeline, only very light ('smoke') tests should be included here. These smoke tests will not verify performance, but will only validate whether the software runs correctly (without errors and warnings) and produces sane output. The second and third use cases will make use of more extensive performance tests that will run larger scale problems in order to obtain sufficiently accurate performance numbers. Thus, for each part of the software stack, the EESSI test suite will actually contain multiple tests: smoke tests and performance tests, each potentially at multiple scales (e.g., numbers of nodes). For testing the software stack on a new client system, it will also be possible to easily select a relevant subset of tests (e.g., only single node tests in case the client machine is just an individual virtual machine in the cloud).

Note that since both the smoke tests and performance tests may require multiple nodes, or specialised hardware (e.g., GPUs) in order to run, the CI/CD pipeline cannot be run on just any machine. Since many parties involved in EESSI are

HPC centres, one option is to use these facilities to run the CI/CD tests. Alternatively, cloud resources may be used for this purpose. Whichever solution is adopted, the goal is to fully automate the software testing pipeline, so that contributions of, for example, new applications to the software stack can be easily processed – even if they grow in volume.

Testing functionality of the filesystem and compatibility layers, in general, does not depend directly on system specific hardware and can be tested using unit tests, for example, checking whether certain paths and executables exist or checking the output of executables using very minimal commands like `--help` or `--version`. Such tests are very fast, easy to specify and could be done in any typical unit testing framework. However, these layers are also composed of important tools that the software stack is built upon, such as *binutils*[47] and the *gcc*[48] compiler. For these type of tools additional functional testing is required in order to check, for example, if the compiler-*binutils* combination is able to generate proper micro-architecture code.

In general, testing functionality of the software layer involves a much more complex *set* of tasks and site specific dependencies. Among other things, HPC software testing may require: (i) interaction with (various) batch systems; (ii) interaction with a module environment; (iii) a large amount of compute resources; (iv) sanity checks on non-deterministic outcomes; (v) performance analysis; (vi) different runtime options depending on available hardware (GPUs, CPUs, architectures, etc.); and (vii) different network interconnect technologies.

This makes it quite different from regular software testing. The ReFrame framework[49] is designed specifically for testing HPC software and has a number of key features to support the above requirements. While any framework would suffice for the filesystem and compatibility layer tests, we aim to use ReFrame for testing all three layers in order to maintain testing consistency across the different layers and facilitate contributions to the test suite.

3.7.2 | Challenges and benefits of the test suite

It is unavoidable that system specific information is required to run HPC software tests. In order to make the EESSI test suite portable between systems, it is important that the system specific information is separated from the test definition as much as possible. ReFrame (partially) supports this by specifying system-specific information in a configuration file. At the same time, ReFrame tests often still contain *some* system-specific information. The EESSI project is working closely with the ReFrame developers to further separate system-specific information from the test definitions. Ideally, end-users of the test suite would only have to specify their system specifics in the ReFrame configuration file (and the creation of such a configuration file could even be automated in the future).

An HPC software test suite that is designed with portability between systems in mind could be a significant contribution to the HPC community. While testing software installations is important, individual HPC centres often lack sufficient manpower to design and run proper software testing for *all* their software installations. Being a collaboration, EESSI brings economy of scale to this challenge. Even HPC centres that would *not* use the EESSI software stack themselves could still benefit from using the EESSI test suite to test their own, local software installations.

Another benefit is collaboration between EESSI and software developers. Considering the potentially very large community of end-users, having their software in the EESSI stack run correctly would be of interest to the scientific software developers themselves. We see a win-win situation here: software developers know best how to design a proper test for their application and could assist EESSI in creating these. At the same time, this also ensures that the users of said software in EESSI have a good end-user experience.

4 | USE CASES

The EESSI project enables a wide range of use cases benefiting researchers, developers of scientific software, system administrators in multiple different ways which we will discuss in the following sections.

4.1 | A uniform software stack across HPC clusters, clouds, servers and laptops

Today, researchers may have access to several HPC clusters and cloud environments. Having access to the same software environment on all of these resources will greatly lower the threshold to use these more efficiently and decrease the effort needed for adapting workflows and job scripts. In many cases, users do not build scientific software themselves on HPC systems, but use packages that are pre-installed by system administrators or application/user support teams. These

installations may vary in various aspects such as versions of operating system components (e.g., `glibc`), versions of toolchains used to build software stacks as well as versions of the libraries and scientific software packages being provided. At best, these variations only require effort to adapt scripts (using different modules, adjusting parameters). At worst, the differences render software stacks incompatible and thereby limit researchers in which resources they may use for a given task.

Using cloud infrastructures becomes increasingly important for researchers to satisfy needs which may not be easily fulfilled by traditional HPC systems. While cloud resources can be made quickly available, often they are launched without any scientific software installed. Hence, they are not used to their full potential, or only by those users comfortable in installing many software packages themselves. Providing a common software stack through EESSI removes this bottleneck entirely. The software is not only available almost instantly on a newly launched cloud instance, it is also available with the same user interface and has been optimised for the hardware it will run on in the cloud.

While EESSI's key target platforms are HPC clusters and cloud environments, it is, however, designed to run anywhere. Therefore it can also be used on servers, workstations and laptops. In practice, this enables researchers to use the most convenient system for a given task. For example, they can develop features of their applications and test the functionality on their own machines, and then run performance tests on larger machines. While EESSI is fully compatible with Linux and therefore naturally works on Linux based systems, EESSI integrates well with machines running Windows via Windows Subsystem for Linux. Furthermore, a macOS port is under active development. Thus, virtually any modern client will be able to use the software provided through EESSI.

4.2 | Customising and extending the software stack

No matter the extent of the EESSI software stack, there will always be some software missing for someone, or how the stack is presented to the end user may not be consistent with how sites have done this in the past. EESSI is being designed with customisation in mind from a number of different perspectives:

- EESSI will provide a number of different *views* of the software stack (via the use of *environment modules*). It will also provide instructions on how to construct a completely new view of the stack (or subset of the stack). Such a new view will not require any re-installation.
- EESSI will provide detailed instructions on how to reliably extend the available software stack.
- EESSI will be able to integrate with vendor-provided software. A key example of why this would be required is the performance of MPI libraries, where vendor installations are expected to be system-tuned and have better performance. EESSI will ship with open source MPI implementations but will provide the mechanisms for users to override these with ABI compatible alternatives. All 4.x versions of Open MPI[17,50] are ABI compatible, and there are currently 6 partners in the MPICH[18] ABI Compatibility Initiative.[51] The override mechanism can also potentially be leveraged more generally for other packages that respect ABI compatibility.

4.3 | EESSI for continuous integration, porting and benchmarking

As mentioned in Section 1.5, EESSI opens the door to a level of continuous integration capabilities for HPC applications that was previously unheard of. The complexity of the dependency tree of applications, coupled with the variety of possible hardware, can make running and maintaining continuous integration workflows a challenge. Developers may spend huge amounts of time preparing the environment inside a container, or they may have to maintain a container themselves to run their tests. Both of these cost substantial effort, and also carry the substantial risk that the testing environment does not accurately reflect the environment or architecture where the code will actually be built and executed.

As an example of the power of EESSI in this regard, EESSI has already prepared a GitHub Action which can leverage the EESSI stack.[52] This action means that within less than a minute, a developer can have the full EESSI stack available to use for building and testing (including recent compilers, MPI implementation, math libraries, etc.).

This also means that EESSI can facilitate the porting and benchmarking of an application on a variety of hardware since it will consistently provide the supporting software stack. Developers could, for example, spin up architecture-specific nodes in cloud environments for initial porting work as a low threshold method for access to hardware.

4.4 | EESSI as part of an HPC training infrastructure

Access to HPC resources on the scale required to meet the training needs of all countries is a serious concern. Even dedicated training infrastructures are frequently not maintained to the same extent as production systems (at least in terms of their scientific software stack).

There are a number of solutions available that can dynamically provision virtual HPC systems in a public cloud (open source examples include Magic Castle[53] and Cluster in the Cloud[54]). Such approaches require a recent software stack to be relevant for training purposes, and ideally one that is consistent with any production-level resources that the learners might have access to. EESSI can provide a software stack that ensures a consistent user experience between such sets of resources.

EESSI has been integrated as an option in both Magic Castle and Cluster in the Cloud, meaning one can dynamically create temporary event-specific HPC clusters for training purposes with a full scientific software stack provided by EESSI.

4.5 | Enhanced collaboration with software developers and advanced users

Building and installing scientific software correctly and tuning it to make best use of the hardware (CPU family, microarchitecture, GPUs, interconnect, etc.) can be a daunting task even for experienced HPC system administrators. Therefore, large HPC sites often seek collaboration with developers and/or advanced users such that highly popular software is optimised for their system. While such collaborations may happen naturally, the reach of the optimisations is often limited to the user base of the HPC system on which the optimisations were performed. At best, one can hope that the results are integrated into the software codes and build instructions, and thereby more users at other sites may benefit from such improvements eventually too.

With EESSI such collaborations are significantly enhanced: a software developer or user is not merely collaborating with one HPC site on a (single) HPC system, but rather simultaneously with a whole class of HPC systems with similar hardware features. That is, all HPC sites operating a system of a specific class bearing similar hardware features will easily and quickly benefit from the optimisation work. EESSI is interested in quickly integrating the optimisations into new releases of its stack, making them available instantly at all sites deploying EESSI and, importantly, without any additional effort by software developers nor HPC system administrators at other sites.

For the software developers, EESSI provides a well defined and fast means to distribute optimised builds of their scientific applications. Beyond this, through the use of common and well-tested software stacks to deliver required dependencies, EESSI also provides a predictable and reproducible environment for developing, tuning, building, and testing software for those who may still need to compile a customised version of any particular package.

4.6 | Community software and portable workflows

Many communities, particularly in the bioinformatics domain, employ complex workflows requiring a broad spectrum of software tools to work together in concert. In EESSI, we wish to support the idea of a community-specific view of the software stack, filtering the available software to what is most relevant to that community. The goal is to collaborate with a *community champion* who will help communicate the needs of the community and work with EESSI developers to mould these into actionable requirements.

One immediate use case we see is the ability to define and support portable workflows, where the complete set of related software requirements are provided by the EESSI software stack. Such workflows can potentially be shared (e.g., through tools such as Nextflow[55] and Snakemake[56]) and distributed with EESSI itself.

5 | DEMONSTRATION OF PILOT SETUP

While working towards a production-ready version of the repository, the EESSI project has adopted an Agile-like way of working,[57] where we regularly release versions of a proof-of-concept EESSI pilot repository. This allows us to add and test new functionality, features, and software, and fix any issues found in the next version of the pilot repository.

In the examples below, we will make use of a symbolic link named `latest`, which always points to the latest release. At the time of writing, this pointed to pilot version `2021.06`. Up-to-date information about the latest version can be found on the 'Pilot repository' page of the EESSI documentation.[‡]

Note that the releases of our pilot repository currently focus on testing and setting up the overall infrastructure and configuration, solving issues, automating as many steps as possible and so forth. Therefore, we intentionally keep the number of supported (micro)architectures and installed software applications limited; we mainly focus on a couple of families of microprocessors supporting either the `aarch64` or `x86_64` ISA, and some popular scientific applications from different fields. This can be easily extended later on when we reach a production-ready status.

By going through three simple steps, we will describe how easy it is to access the EESSI pilot repository on a Linux system, and start using the provided scientific software stack easily.

5.1 | Step 1: Mounting the EESSI pilot repository

The EESSI repository is offered as a CernVM-FS repository, and, hence, every client first needs to install the CernVM-FS client. It is recommended to use native operating system packages, which are available for various (popular) Linux distributions, and detailed installation instructions can be found in the 'Getting Started' section of the CernVM-FS documentation.[§] Do note that this requires root privileges. There are alternative ways to do this without root privileges, for instance by using a Singularity container or by using the `cvmfsexec`[¶] package. These are less optimal for production systems, though, and will therefore not be discussed here.

Besides the client itself, some configuration for the EESSI repository is necessary too. We make most of this process easy by offering configuration packages at the GitHub repository of the filesystem layer; the latest versions of these packages can always be found at the rolling `latest` release.[#] Some machine-specific configuration still has to be done manually.

As an example, this step can be done in the following way on a RHEL/CentOS 7 machine:

```
# install latest version of CernVM-FS client (see https://cernvm.cern.ch/fs/)
sudo yum install -y https://ecsft.cern.ch/dist/cvmfs/cvmfs-release/cvmfs-release-latest.noarch.rpm
sudo yum install -y cvmfs

# install CernVM-FS configuration files for EESSI repositories
# (see https://github.com/EESSI/filesystem-layer)
sudo yum install -y \
   https://github.com/EESSI/filesystem-layer/releases/download/latest/cvmfs-config-eessi-latest.noarch.rpm

# create local CernVM-FS configuration file
# (single client, no proxy; 10GB for CernVM-FS cache)
sudo bash -c "echo 'CVMFS_CLIENT_PROFILE=single' > /etc/cvmfs/default.local"
sudo bash -c "echo 'CVMFS_QUOTA_LIMIT=10000' >> /etc/cvmfs/default.local"

# set up CernVM-FS
sudo cvmfs_config setup

# check access to EESSI pilot repository
ls /cvmfs/pilot.eessi-hpc.org/latest
```

[‡]See https://eessi.github.io/docs/pilot/
[§]See https://cvmfs.readthedocs.io/en/stable/cpt-quickstart.html
[¶]See https://github.com/cvmfs/cvmfsexec
[#]See https://github.com/EESSI/filesystem-layer/releases/tag/latest

5.2 | Step 2: Sourcing the EESSI initialisation script

After the CernVM-FS repository for the EESSI pilot has been made available, the user of the repository needs to find and select the right software stack for their hardware. This can be done manually, but it is much easier to use the provided `bash` shell initialisation script:

```
source /cvmfs/pilot.eessi-hpc.org/latest/init/bash
```

This script will use *archspec* to detect the microarchitecture of the user's processor, find the best matching and compatible software stack in the repository for this particular microarchitecture, and finally set up the Lmod environment module system so that it will find all the modules in this software stack. Note that no additional packages are required on the user's computer for this to work.

5.3 | Step 3: Find and load the right module(s), and start computing

Now that the module environment is configured, the user can start searching for modules using `module avail` and/or `module spider`, load what they need, and start doing actual computations. For instance, to load and start the default version of GROMACS (assuming the existence of the necessary input file `ion_channel.tpr`):

```
module load GROMACS
gmx mdrun -s ion_channel.tpr -maxh 0.50 -resethway -noconfout -nsteps 1000
```

6 | PERFORMANCE EVALUATION

While the concept behind EESSI as outlined is very attractive, its success will depend on its ability to *perform* as advertised. In particular, of critical interest is the ability of the EESSI software stack to efficiently leverage the hardware available on HPC resources. Given the number of nodes that appear in typical HPC infrastructures, it is the performance of EESSI on the interconnect that is of initial primary concern.

Our first investigation was to verify basic point-to-point latency and bandwidth performance. For this we used the JUWELS system (a SkyLake architecture with EDR Infiniband interconnect) and the OSU Micro Benchmarks suite.[58] We found a minimum latency of $<1\ \mu$s and a point to point bandwidth of about 12 GB/s for the EESSI Open MPI installation. This is entirely consistent with the performance of the system-provided Open MPI stack.

While such a confirmation is encouraging, it is the performance of the applications themselves that is of most interest. The comparison we wish to make is from the perspective of an *end-user of an application*, that is, someone who will typically be using an application as it is provided/supported by the user support team of the supercomputing site. In this scenario, the user does not decide how the application is configured and compiled, and is only concerned about whether the installation works, and performs well, for their use case. The system upon which we will base this comparison is the CPU-only partition of the JUSUF system at Juelich Supercomputing Centre (AMD EPYC with HDR100 interconnect).

We select one of the pilot applications from the EESSI pilot repository, GROMACS, using a specific version that is available both from EESSI and from the system-supported software stack. We use a benchmark case for GROMACS and compare the performance of the EESSI installation (using the MPI implementation provided by EESSI) with that of GROMACS as provided as part of the default software stack of the system (which uses the vendor-recommended MPI installation). For full details of the application installations, the system, and the specific test case, please see Appendix B.

In its output, GROMACS provides the number of nanoseconds it can simulate per day, which ideally should scale linearly with the node/core count. It is this number that is of primary interest to a GROMACS end user, since it has an inverse relation to the time to solution (more nanoseconds per day means a shorter time to solution). In Figure 4, we

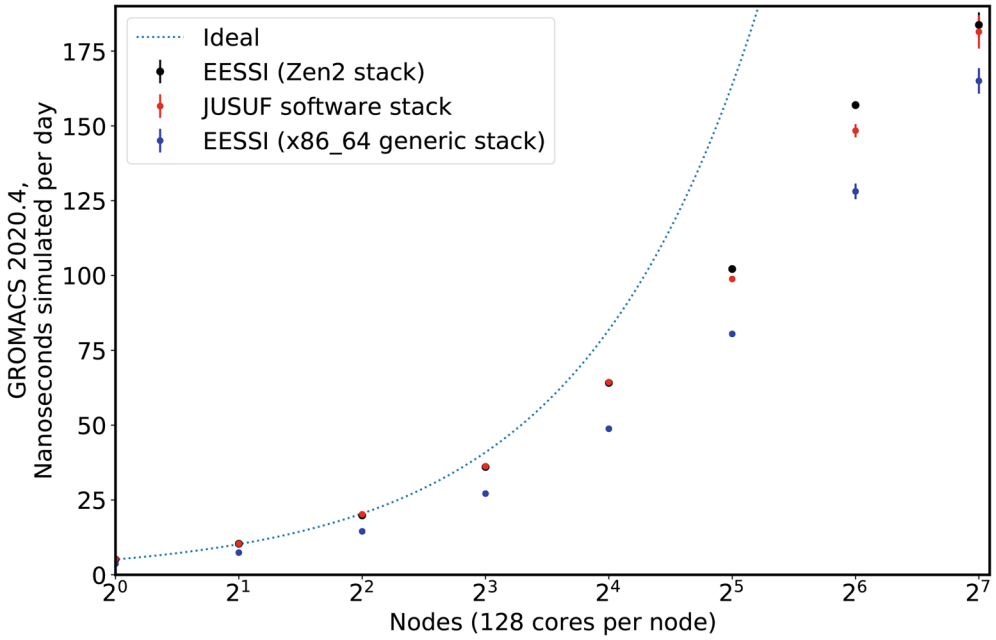

FIGURE 4 Comparison of the scalability of GROMACS as provided by EESSI and that of the same version of GROMACS provided by the JUSUF default software stack. The number of nodes is on the *X*-axis, reaching a maximum of 128 nodes (16,384 cores), and the number of nanoseconds simulated per day (which ideally should scale linearly with the node/core count) is on the *Y*-axis. For EESSI, we include results for both the architecture-optimised Zen2 build and a generically optimised `x86_64` build

provide a comparative analysis of the scaling behaviour of GROMACS (for the particular version of GROMACS and use case) using

- the architecture-optimised EESSI installation (labelled as the 'Zen2 stack'),
- the architecture-optimised system-provided installation (labelled as the 'JUSUF software stack'), and,
- a generically optimised EESSI installation (labelled as the '`x86_64` generic stack').

It is highly encouraging to note that the performance of both architecture-optimised installations is consistent to 128 nodes (16,384 cores). For the generically optimised EESSI installation, we can see that the performance degradation for using non-optimised binaries is about ~25% for this particular workload (up to 128 nodes). We expect the level of impact for non-optimised binaries to vary significantly from application to application since it is dependent on, for example, the level of support for efficient vector instructions within the compiled code.

At larger node counts it would appear that EESSI does marginally better than the system-provided stack, but there are a number of minor qualifications to the results that should be taken into consideration:

- the particular test case used scales poorly beyond 16 (2^4) nodes (2048 cores);
- we have used 4 OpenMP threads per physical node for all node counts;
- there was no native installation of CernVM-FS on JUSUF, the EESSI installation was run from inside a Singularity container;
- no effort was made to tune the MPI communication settings for larger node counts (where such tuning is likely to have an impact).

Taken together, we would suggest that the performance difference between the architecture-optimised installations coming from EESSI and the system is negligible.

Given that we only consider a single application on a single system, this is clearly not a comprehensive performance comparison. What is clear from this analysis is that the performance of software installations provided through EESSI

can indeed be competitive with system-provided installations, both on single nodes and at scale: using EESSI does not have to come with a performance penalty.

7 | CHALLENGES AND LIMITATIONS

In this section, we discuss several challenges and limitations that we have encountered in EESSI so far.

7.1 | Challenges derived from using CernVM-FS for software distribution

As discussed in Section 3.1, CernVM-FS is used in the filesystem layer of EESSI to distribute the provided software installations.

This imposes some requirements for being able to use EESSI, the most prominent one being that administrator privileges are required to install and configure CernVM-FS, since the EESSI repository must be mounted under the system-wide `/cvmfs` mount point.

This is an obvious challenge for individual researchers who do not have administrator permissions on all infrastructure they use (in particular not on multi-tenant environments like HPC systems). An easy workaround is to use a small container image that includes CernVM-FS and mount the EESSI repository in the container, which is relatively trivial for single-node workloads. Using a container for more extensive use of EESSI, and in particular for running multi-node workloads on an HPC cluster leveraging EESSI, requires additional work however, since then one needs to configure the CernVM-FS cache hierarchy appropriately[‖] and use specific mount settings for the container.

A proper system-wide deployment of EESSI does require the involvement of system administrators, however. This is especially true if low access latencies and persistent access to the EESSI software stack must be ensured, even in case of a network outage (see also Section 7.4), if cluster worker nodes are diskless, or if worker nodes do not have network connectivity to the global internet.

While technical solutions are already available for these challenges, which we will document extensively, additional alternative distribution mechanisms next to CernVM-FS can also be evaluated, so a broader range of options for accessing the EESSI software stack is available.

Nevertheless, we feel that the benefits of getting access to a rich software stack like EESSI vastly outweigh these concerns regarding the use of CernVM-FS.

7.2 | Long-term sustainability

Long-term sustainability is an important goal of the project. Users need to feel confident when basing their work on it, and, from the EESSI perspective, it allows collaborators to accept initially higher effort investments in developing the service, assuming that they are paid back by higher quality and lower maintenance for the software provisioned (despite increasingly more diversity of hard- and software).

It is noted that there is a significant overlap between the developers and the users in the project. Particularly, HPC centres contribute significantly to the development of the service, and at the same time they will benefit greatly from its use. Hence, there is a strong motivation for them to maintain and to further develop the service in the future. Already to date, the project receives contributions of different forms and from various sources. While the effort put in by HPC centres is focusing on developing various aspects of the service and on testing regular releases on their premises, the project also currently receives significant support from commercial cloud providers (AWS and Microsoft Azure) in the form of credits to run components of the infrastructure, to test components, and to build software stacks on several CPU microarchitectures.

In a relatively short time, the project has already developed a working prototype (minimal viable product) which can, and is, being actively used for demonstrations and testing. Offering the service in production quality, however, requires that more of the internal procedures to build and to maintain software stacks are automated as much as possible to

[‖] See https://cvmfs.readthedocs.io/en/stable/cpt-configure.html#cache-settings

minimise the operational cost, that is, to realise sustainability in the long term. It is also being investigated if and how service-level agreements with large scale users, for example, other HPC sites, could be put in place. For the users this would increase their confidence in the reliability and the longevity of the service. For the project partners, this would establish a formal framework for sharing and limiting the risks in (jointly) providing the service. One way forward could be to form a new non-profit organisation, as is common when establishing infrastructures for research. Framing EESSI as a research infrastructure would also allow us to pursue a number of different funding opportunities, for example, within Horizon Europe.[59]

7.3 | Impact of security updates

As the compatibility layer of the EESSI repository ships its own operating system libraries, it is inevitable that security vulnerabilities will be found for the versions of packages that we have installed in this layer. The impact of such issues should in principle be limited, as everything from the EESSI software stack is running in user space. However, due to the possibly large number of systems that make use of EESSI, it is still extremely important to deal with this properly. We have to keep in mind that doing in-place updates of vulnerable packages can be a challenging task, since many of the scientific tools in the software layer do depend on these packages. Note that this is not specific to EESSI, but is something that every HPC centre already has to deal with.

As we have mentioned in Section 3, we do not have a choice but to install these security updates as soon as possible. Gentoo provides a tool (`glsa-check`) that allows us to regularly scan our compatibility layer for vulnerable packages. This is something that we already do for our pilot repository, and we send out notifications to some of the maintainers of the repository when vulnerabilities are found. In this case, we do a quick in-place update of the affected package(s).

In order to find out if these updates break anything, it is crucial for us to do proper testing. As outlined in the subsection about testing in Section 3, we are developing tests at various levels to make sure that the software stack is working properly. We can leverage this test suite to re-run all the tests right after a security update has been installed.

Though we expect that in most cases nothing will be broken by doing a security update, which is often a minor update, it could theoretically happen. In this case, we could think of several ways to deal with such an issue. First, we can try to fix the broken application, for instance by reinstalling it, if necessary with a small patch to make it compatible with the updated package. If the problem is wider than just one or a few applications, CernVM-FS offers an option that allows clients to mount older versions of the repository. This basically gives us the opportunity to let clients choose between (absolute) reproducibility and security. We could also dump the older version into an archive or container image, which can then be used by clients to still run these older versions on their own systems. Ultimately, if an update really breaks the software stack, we can use the rollback mechanism of the CernVM-FS repository to undo the change (i.e., the update) and revert to an earlier version.

7.4 | Mitigating availability and integrity risks

Though the EESSI software stack is targeted at different kinds of systems, we see HPC clusters as one of the main targets of the stack. The scientific software stack is one of the key components of HPC clusters, and, therefore, switching to a remotely hosted software stack that is managed by several (external) people may feel like a risk to the administrators of these HPC clusters: how can they be sure that the stack is always available to their users, and how do they know that the stack does not contain malicious software?

7.4.1 | Availability of the stack

There are several features and infrastructure levels that will help in ensuring that the software stack is always available. First of all, CernVM-FS is set up to deal with this by design: by having several Stratum 1 servers and levels of cache, the system will be resilient against the loss of one or several Stratum 1 servers. Clients will automatically switch to another

one in such cases, and may even still use their local cache or Squid proxies in case all the Stratum 1 servers would be down.

The publicly available EESSI Stratum 1 servers will all be running on a variety of reliably hosted infrastructure, which can for instance be university resources or public cloud providers, and at least a few members of the EESSI core team will have access to all of them. In case of issues, we can quickly diagnose and resolve them. And, as we use DNS aliases for all our Stratum 1 servers in the client configuration files, we can easily and quickly take out a Stratum 1 server, or even replace it by a new machine, should the need arise for doing so.

Finally, every site that wants to use EESSI can easily set up a local 'private' Stratum 1 server, and add this server only to the configuration of their cluster nodes. This allows them to have a full copy of the software stack as a part of their infrastructure, which means that it will still be fully available in case of a problem with outgoing network connectivity.

7.4.2 | Integrity of the stack

Besides the security aspects that we have discussed in Section 7.3, another important security-related aspect of a collaboratively managed stack is making sure that no malicious binaries get added to the stack. Here we rely both on features of CernVM-FS, policies that we use for building and adding software, and transparency.

CernVM-FS itself provides several integrity features that make sure that the contents of a repository cannot be changed between the Stratum 0 and the client: the latter will always verify the signature and hashes of the data it retrieves, and any changes made to this data will lead to a failure. This means that when a malicious binary would get injected into the repository on, for instance, a Stratum 1 (which has to be a bit more open, by design), it will not be possible to actually execute this malicious binary on any CernVM-FS client. As the Stratum 0 is in some way more vulnerable, access to this machine will be severely restricted. Yubikey devices will be used to store the master keys of the repository, and changes in the repository will be actively monitored.

Additionally, we need to control both the software build environment and ingestion process. We will apply industry standard practices to secure the deployment process, including signing of software builds prior to ingestion. We also want to automate as many steps as possible in this process, with humans approving certain actions on various levels. For instance, software builds are stored as tarballs, which will only be ingested to the repository after inspection and approval of one of the maintainers of the repository.

Furthermore, the deployment process should be as transparent as possible by making the logs of the software build process and actions that were taken to ingest the software to the repository publicly available. This allows anyone to verify how the software was built and added to the repository.

Though all of this will minimise the chances of deliberately injecting malicious binaries into the repository, the possibility itself can never be completely ruled out: ultimately, even one of the scientific applications itself may have malicious code in it, which would mean that every line of source code would have to be inspected before actually building it. This problem is not specific to EESSI however, but applies to the broader community as well. We do intend to make an effort to address this however, by looking into available scanning tools for both source code and binaries, and integrate such a tool into our deployment pipeline.

7.5 | Providing a diverse set of applications

Providing a diverse set of applications for multiple combinations of compilers, mathematical libraries and MPI implementations is a particularly challenging problem. A number of recent developments may help to mitigate some of these issues:

- Recent EasyBuild toolchains make use of FlexiBLAS[60] which allows run-time exchangeable backends for BLAS and LAPACK.
- There are compatibility initiatives for both Open MPI and MPICH which should facilitate replacing the MPI implementations shipped with EESSI with a local ABI-compatible alternative. The features necessary to enable this (e.g., overriding library search paths) are already integrated and tested in EESSI.

These alone help reduce some of the multiplicative factors but do not address the fact that the potential space of scientific applications is huge (and growing, as developers are increasingly open with their software). Saying that we intend to cover all possible requirements of all scientists is not realistic, instead we aim to enable scientists to easily leverage and extend EESSI for their own use cases. EESSI is being designed from the outset with this in mind since we already have the limitation that there is plenty of important research software that we cannot distribute due to licensing restrictions. A number of partners have also expressed interest in creating their own infrastructure (using the EESSI blueprint) which extends EESSI for use cases within private organisations.

The task of *delivering* such a software stack to end users is capably handled by CernVM-FS, with large deployments used by CERN, NERSC and CSCS. It provides multiple layers of distribution at the infrastructure level, and arbitrary levels of caching on the system level. For example, you can have a local squid proxy for your organisation which caches information for local systems; and then each individual system can then have a global cache complemented by node-level caches. It is also possible to pre-populate CernVM-FS caches and to export (a subset of) EESSI so that no caching, or indeed even CernVM-FS, is required at all at runtime.

7.6 | Providing a large set of applications on different platforms

As discussed in Section 7.5, the potential application space of EESSI is large. In addition to this, applications must be ported to the various hardware platforms that EESSI will support. It is to be expected that EESSI users will be eager to have new architectures supported and new versions of their required software available. This ultimately raises the question of the long-term maintenance of the EESSI stack. The EESSI project can provide automation and tools to *facilitate* supporting new hardware platforms and new software versions, but EESSI at its core is a community project that requires human effort leveraging community knowledge to do effective maintenance.

Key contributors to EESSI are likely to be HPC hosting sites, since they are keen to have complete stacks of high performance scientific software on their platforms, and scientific software developers, since EESSI offers a means to automatically distribute their latest software to a large number of systems. This is evidenced by the current pool of contributors to EESSI which match these profiles. Contributors are, of course, most likely to focus on the platforms and applications which are most relevant to them. Through its automated workflows EESSI will encourage them to also consider other platforms, leveraging the fact that the long-term return on investment through EESSI is greater than working on any single platform.

Ongoing support for any particular hardware platform or scientific application will likely hinge on the community willingness to continue supporting these. This implies that EESSI will have to consider an approach to sunsetting support for both platforms and software. Strategies on how this could be approached align with the discussion in Section 7.3 on how to provide access to older releases of EESSI.

7.7 | Recompiling software

EESSI aims to give the majority of scientists a high quality *out-of-the-box* experience but it is inevitable that some users will need to either rebuild an existing package with special requirements, install a new version or install a missing application. As mentioned in Section 7.5, EESSI is being designed with this in mind. As is standard with EasyBuild, each software installation is shipped with extensive reproducibility information. Users can use this information to customise a rebuild of a software package, or tweak it for a new application version.

Documentation is also under development that will guide users in how to use EESSI as the base-layer for their own software stacks. This is a priority issue for EESSI as we would like to see EESSI being used to provide toolchains and dependencies in the continuous integration workflows of application developers.

7.8 | Supporting different types of interconnects

EESSI addresses the support of interconnects via the network communication libraries libfabric[61] and UCX.[62] These are the low-level communication libraries that are in turn utilised by, for example, MPI implementations. EESSI

leverages the fact that these have been widely adopted in the HPC domain, and together these cover the vast majority of interconnects on which EESSI is likely to be used. EESSI installations of libfabric and UCX are *fat*, meaning that they are configured with the widest possible interconnect support. Both frameworks can handle runtime detection of interconnect hardware, that is, they will never try to use hardware that is not there, even if they are built fat. Runtime *optimisation* for the available hardware can also be made via appropriate environment variables.

Specifically with respect to MPI, EESSI is being designed such that it will be possible to swap out the MPI implementation with a local ABI-compatible alternative at runtime, which is supported by Open MPI and the MPICH ABI Compatibility Initiative.[**] This means that as long as a site can provide an interconnect-enabled ABI-compatible MPI installation, their hardware can be supported by EESSI.

7.9 | Supporting different types of accelerators

Supporting accelerators, such as GPUs, can be challenging since there can be many generations of hardware requiring different compilation options, and minimum local driver versions may be required for software to function correctly. In addition, accelerator software stacks may come with restrictive licensing which can limit what EESSI is *allowed* to distribute.

EESSI is aware of the potential issues but can only address them as specific instances arise. For example, for NVIDIA hardware:

- without a special exemption we may not be allowed to distribute the CUDA compilers, but since these are binary installations we can integrate support for externally provided CUDA installations such that they can be found by EESSI software;
- CUDA applications can be compiled as 'fat binaries', that is, with support for all architectures supported by the specific CUDA version;
- CUDA compatibility libraries can be installed in user space which allow users to access features from newer versions of CUDA without requiring a kernel driver update, so EESSI can relax the driver requirement to a minimum driver version (i.e., *not* a specific driver version).

In general, accelerator installations are an optional component of the EESSI stack and we can leverage this in order to sanity check that the hardware can in fact be utilised correctly. When using scripts that enable specific accelerator support, we can perform additional configuration steps and also add verification checks to ensure that the software is working correctly for the available hardware.

8 | RELATED WORK

Beyond tools like EasyBuild and Spack which are often used for providing a central software stack on HPC systems (see Section 2.3), there are a couple of other recent projects that aim to ease the burden of installing scientific software on HPC systems. We briefly discuss these projects in this section.

8.1 | The Compute Canada software stack

The EESSI project is inspired by the work done by the Compute Canada consortium, where a similar setup using largely the same open-source tools was constructed to provide a unified software environment for Canada's National Advanced Computing Centers.[40] This consists of a stack of software installations for the different hardware platforms that are available in the Compute Canada infrastructure, which are installed with EasyBuild on top of a compatibility layer that is

[**]See https://www.mpich.org/abi/

constructed using Gentoo Prefix, and which is distributed via CernVM-FS. The result is a familiar software environment that is available for researchers using the Compute Canada infrastructure, regardless of the specific system they are using.

The goals of EESSI are significantly more ambitious however. We aim to provide a software stack across different countries and continents, and support different processor families (Intel, AMD, Arm, POWER, and eventually also RISC-V), while the Compute Canada software environment currently only covers Intel and AMD processors (`x86_64` ISA). In addition, EESSI uses an additional tool not used by Compute Canada to automatically select the set of software installations that is best suited for the client system (see Section 3.3).

Furthermore, while the work done by Compute Canada has a strong focus on the needs of the Canadian research community, EESSI aims to cater to the broader research community, and foster collaboration across nations both within Europe and across the globe. This will require different design decisions to be made compared to the Canadian unified software environment, and to put policies in place to ensure that efficiently managing a software stack that is used by many HPC sites around the world is fostered. We will, for example, set up a mechanism to let the broader research community propose and contribute software installations to include into the EESSI CernVM-FS repository.

During the early stages of the EESSI project, we discussed the possibility of repurposing the Compute Canada software stack for the broader research community, by supporting a broader set of hardware platforms and allowing others to influence the design decisions that have been made so far. Both parties quickly concluded that this would be difficult in practice, and that letting EESSI start afresh with the broader scope in mind, taking into account the lessons learned by the Compute Canada consortium, would be the best way forward.

8.2 | Extreme-scale scientific software stack (E4S)

The Extreme-scale Scientific Software Stack (E4S) project[63] has high-level goals which are similar to those of EESSI, but a very different approach is taken to reach those goals. In E4S, the intent is to facilitate from-source installations via Spack and to provide pre-built binary packages and containers for specific system architectures. The key difference between E4S and EESSI is that with EESSI the software installations themselves are distributed, which get downloaded in the background on-demand as the installations are accessed by CernVM-FS (see Section 3.1), rather than using packages or container images that need to be installed or downloaded as is done in E4S. This results in a more user-friendly experience for researchers who are using the provided software installations.

In addition, the software installations provided by EESSI can be leveraged on a wide range of operating systems, thanks to the compatibility layer (see Section 3.2), while the pre-built binary packages for the E4S software stack are specific to a particular (version of an) operating system.

The containers provided by E4S can be quite large: for version 21.11 the available (compressed) container images are over 50 gigabytes in size.[††]

In contrast, since the specific files required to use software installations provided through EESSI are only downloaded on-demand, the required network bandwidth is significantly reduced (see Section 3.1).

There is a strong focus in the E4S software stack on the needs of the Exascale Computing Project (ECP),[‡‡]which is reflected in the software applications and libraries that are included in it.[§§] In EESSI, we intend to be (even) more ambitious, and cater to a wider variety of scientific domains (see Section 7.5).

8.3 | The OpenHPC project

OpenHPC[64] is a community project focused on facilitating the deployment and maintenance of HPC systems. It provides pre-built packages (RPMs) for a reference collection of open-source HPC software components, which encompasses both system software like resource managers (Slurm, OpenPBS) and administrative tools (ClusterShell, Warewulf),

[††]See https://hub.docker.com/u/ecpe4s

[‡‡]See https://www.exascaleproject.org

[§§]See https://e4s-project.github.io/Resources/ProductInfo.html

as well as a variety of tools and libraries that are often prevalent in an HPC software stack. The latter includes compiler suites, MPI libraries (Open MPI, MPICH, MVAPICH2), performance tools (Score-P, Scalasca, etc.), various special-purpose libraries (Boost, OpenBLAS, PETSc, etc.), software installation tools (EasyBuild, Spack) and so forth. Corresponding environment module files are included (where relevant) to ensure easy access to these different tools and libraries, which are consumed through the Lmod[15,26] environment modules tool that is also provided. Extensive documentation that covers the integration between different components is available to help with configuring the various components.

The collection of software packages provided by OpenHPC is relatively limited, mainly due to the effort that is required to maintain and test them across the different supported operating systems (CentOS and OpenSUSE) and hardware platforms (`x86_64` and `aarch64`), and also to thoroughly document them. For largely similar reasons, the OpenHPC packages include generically optimised binary software, which can significantly impact performance as discussed in Section 1.3, a tradeoff that is often made when using a traditional package manager, as we discussed in Section 2.2. Expanding the focus to providing pre-built packages that were optimised for different specific generations of Intel, AMD, and Arm processors would require substantially more effort in this context, and is considered to be out of scope for the OpenHPC project.

The EESSI software stack largely circumvents these issues by using a different software distribution mechanism, and by using a compatibility layer to support a broad range of operating systems and versions. This way, the scope of supported software applications, tools, and libraries can be expanded significantly, and software installations that are optimised for specific generations of microprocessors can be provided.

9 | FUTURE WORK

While work on EESSI has only started in early 2020, the core functionality – distributing stacks of performance-optimised scientific software to any client running on different operating systems and on different processor architectures – is already working in the proof-of-concept EESSI pilot repository, and being used for demonstration, testing and development.

Through regular testing and discussions with potential users, we have identified the following improvements to reach production-ready status:

- automate the procedure to add new software packages to the EESSI stack (see Section 3.4 for a description of the current procedure);
- document how to build and install software packages which may not be distributed via EESSI for technical or legal reasons;
- introduce monitoring capabilities for our services, particularly, the filesystem layer;
- work with hardware providers, such as NVIDIA, to understand how we can support accelerator run-times (this is already technically possible, but licensing is unclear);
- more extensive testing of the compatibility and software layer is required, including verifying correct functionality, validating the output produced by scientific software applications, evaluating performance and so forth.;
- deploying additional software installations must be done in an automated and secure way, to prevent supply chain attacks by malicious entities.

In addition, there are several possibilities to further improve the EESSI solution to reach an even wider audience or enable customisations that support specific features of the hardware or the software environment at a client machine by

- verifying the ability to override the EESSI installations with site or vendor provided alternatives, for example, to override MPI installations;
- adopting toolchains that use FlexiBLAS which allows run-time exchangeable backends for BLAS and LAPACK;
- providing full support for the macOS operating system;
- providing alternative views of the software stack and documenting how to create a custom view.

10 | CONCLUSION

Scientific software is the engine of today's innovations in research and used by many researchers every day to analyse huge amounts of data, to train neural networks or to simulate complex phenomena. Because of changes in the landscape of computational science – the (re-)emergence of different microprocessor families, the increasing variety of accelerators and the expansion to additional scientific domains – installing scientific software correctly and ensuring that it performs well is a growing problem which is even further exacerbated by stagnating funding for teams managing the software stacks.

EESSI – a grassroots community project of mostly HPC system administrators and support team members – is addressing these challenges by providing a common ready-to-use stack of scientific software installations that can be used easily on a variety of platforms, ranging from personal workstations to cloud environments and supercomputers, without making compromises regarding performance. We presented the flexible design of the EESSI solution which integrates well-known production-ready open source tools and services. The flexible design allows EESSI to support a wide variety of use cases which benefit not only researchers in using software anywhere, but also developers of scientific software because they can more easily build against curated stacks of base libraries. Last but not least, EESSI enables a whole new level of collaboration among (HPC) system administrators. We have also demonstrated that software provided by EESSI can deliver application performance comparable to on-premises installed and optimised installations.

While the EESSI stack is already being used to simplify setting up realistic training resources (cf. LearnHPC[65]), and is available for testing on some HPC clusters, more work is needed to make it production-ready. The EESSI project nurtures a culture of welcoming contributions from anyone and also contributing back improvements upstream to tools and services it uses as building blocks.

ACKNOWLEDGEMENTS

We would like to thank the many different partners that have supported the EESSI project so far: Dell Technologies for bringing together the university partners that kick-started the project; Ghent University in Belgium, the Jülich Supercomputing Centre in Germany, the NESSI project partners in Norway including the University of Oslo and the University of Bergen, the University of Groningen, Vrije Universiteit Amsterdam, and the SURF consortium in the Netherlands for their significant contributions in the development and testing of the proof-of-concept pilot repository; the HPC.NRW competence network in Germany for their feedback and contributions; Compute Canada for the extensive technical discussions providing insight and sharing expertise related to their unified software environment; Amazon Web Services (AWS) and Microsoft Azure for supporting the project by providing sponsored credits in their cloud environments to facilitate development, training and demonstration activities in EESSI; the Fenix Research Infrastructure project for providing cloud resources; the Oregon State University Open Source Lab (OSU OSL) for providing access to POWER9 resources.

In addition, we would like to thank Adam Huffman for proofreading this article and providing valuable feedback, as well as the anonymous reviewers for their thorough and constructive reviews, which have resulted in a much improved article.

We acknowledge PRACE for awarding access to the Fenix Infrastructure resources, which are partially funded from the European Union's Horizon 2020 Research and Innovation Programme through the ICEI project under the grant agreement No. 800858. Alan O'Cais acknowledges support from the European Union's Horizon 2020 research and innovation program, under grant agreement No. 676531 (project E-CAM).

Finally, we would like to thank the developers and user communities of the various open source software projects that we use in the EESSI project, including (but not limited to) CernVM-FS, Gentoo Prefix, EasyBuild, Lmod, archspec, Singularity, ReFrame, Cluster-in-the-Cloud, Magic Castle, Ansible, and Terraform. We look forward to keep on working together with these projects and their respective communities, and contributing back bug fixes, additional features, and other enhancements.

AUTHOR CONTRIBUTIONS

Bob Dröge: Conceptualization (lead), Writing – original draft (equal), Writing – review and editing (equal), Methodology (equal), Software (lead). **Victor Holanda Rusu**: Conceptualization (equal), Writing – original draft (equal), Writing – review and editing (equal), Software (equal), Methodology (equal). **Kenneth Hoste**: Conceptualization (lead), Writing – original draft (equal), Writing – review and editing (equal), Methodology (equal), Software (lead). **Caspar van**

Leeuwen: Conceptualization (equal), Writing – original draft (equal), Writing – review and editing (equal), Software (equal), Methodology (equal). **Alan O'Cais:** Formal analysis (lead), Conceptualization (equal), Writing – original draft (equal), Writing – review and editing (equal), Software (equal); Methodology (equal). **Thomas Röblitz:** Conceptualization (supporting), Writing - original draft (lead), Writing - review and editing (lead).

DATA AVAILABILITY STATEMENT

The input data [PRACE-UEABS benchmark suite version 2.1 (Test Case B)] for the performance evaluation in this study is openly available at https://repository.prace-ri.eu/git/UEABS/ueabs/#gromacs. The data generated through the performance evaluation is made available as supplementary material.

ORCID

Bob Dröge https://orcid.org/0000-0002-8279-868X
Kenneth Hoste https://orcid.org/0000-0001-8034-648X
Caspar van Leeuwen https://orcid.org/0000-0003-4407-6675
Alan O'Cais https://orcid.org/0000-0002-8254-8752
Thomas Röblitz https://orcid.org/0000-0001-8366-6868

REFERENCES

1. Dubois PF, Epperly T, Kumfert G. Why Johnny can't build. *Comput Sci Eng*. 2003;5:83-88.
2. Website for the European Environment for Scientific Software Installations (EESSI) project. https://www.eessi-hpc.org
3. Documentation for the European Environment for Scientific Software Installations (EESSI) project. https://eessi.github.io/docs
4. August M, Brost G, Hsiung C, Schiffleger A. Cray X-MP: the birth of a supercomputer. *Computer*. 1989;22(1):45-52. doi:10.1109/2.19822
5. Krizhevsky A, Sutskever I, Hinton GE. ImageNet classification with deep convolutional neural networks. *Communications of the ACM*. 2017;60(6):84-90. doi:10.1145/3065386
6. Tucker T, Marra M, Friedman JM. Massively parallel sequencing: the next big thing in genetic medicine. *Am J Human Genet*. 2009;85(2):142-154. doi:10.1016/j.ajhg.2009.06.022
7. Ison J, Rapacki K, Ménager H, et al. Tools and data services registry: a community effort to document bioinformatics resources. *Nucl Acids Res*. 2016;44(D1):D38-D47. doi:10.1093/nar/gkv1116
8. Wilson G, Aruliah D, Brown CT, et al. Best practices for scientific computing. *PLoS Biol*. 2014;12:e1001745. doi:10.1371/journal.pbio.1001745
9. TOP500. https://top500.org/
10. Sato M, Ishikawa Y, Tomita H et al. Co-design for A64FX manycore processor and "Fugaku". Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis SC'20; 2020:1-15; Atlanta, Georgia. doi:10.1109/SC41405.2020.00051
11. Kovač M, Reinhardt D, Jesorsky O, Traub M, Denis JM, Notton P. European Processor Initiative (EPI)—An approach for a future automotive eHPC semiconductor platform. In: Langheim J, ed. *Electronic Components and Systems for Automotive Applications*. Springer International Publishing; 2019:185-195. doi:10.1007/978-3-030-14156-1_15
12. Evaluation of accelerated and non-accelerated benchmarks. https://prace-ri.eu/wp-content/uploads/5IP-D7.5.pdf
13. Abraham MJ, Murtola T, Schulz R, Páll S, Smith JC, Hess B, Lindahl E. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*. 2015;1-2:19-25. doi:10.1016/j.softx.2015.06.001
14. Hoste K, Timmerman J, Georges A, De Weirdt S. EasyBuild: building software with ease. Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis; November 10, 2012:572-582; IEEE.
15. Geimer M, Hoste K, McLay R. Modern scientific software management using easybuild and lmod. Proceedings of the 2014 1st International Workshop on HPC User Support Tools; November 21, 2014:41-51; IEEE.
16. Gamblin T, LeGendre MP, Collette MR et al. The Spack package manager: bringing order to HPC software chaos. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis SC'15; 2015:1-12; Austin, Texas. doi:10.1145/2807591.2807623
17. Gabriel E, Fagg GE, Bosilca G et al. Open MPI: goals, concept, and design of a next generation MPI implementation. Proceedings of the 11th European PVM/MPI Users' Group Meeting; 2004:97-104; Budapest, Hungary. doi:10.1007/978-3-540-30218-6_19
18. MPICH. https://www.mpich.org/
19. MPI Forum. https://www.mpi-forum.org/
20. OpenBLAS – an optimized BLAS library. https://www.openblas.net/
21. Van Zee FG, van de Geijn RA. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Transactions on Mathematical Software (TOMS)*. 2015;41(3):1-33. doi:10.1145/2764454
22. Intel oneAPI math kernel library. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html
23. BLAS (Basic Linear Algebra Subprograms). http://www.netlib.org/blas/
24. LAPACK - linear algebra package. http://www.netlib.org/lapack/
25. Furlani JL. Providing a flexible user environment. Proceedings of the fifth Large Installation System Administration (LISA V); 1991:141-152.

26. McLay R. Lmod: environmental modules system; 2022. http://www.tacc.utexas.edu/tacc-projects/lmod
27. Anaconda, Inc. Conda package management system. https://conda.io
28. Grüning B, Dale R, Sjödin A, et al. Bioconda: sustainable and comprehensive software distribution for the life sciences. *Nature Methods*. 2018;15(7):475-476.
29. Kurtzer GM, Sochat V, Bauer MW. Singularity: scientific containers for mobility of compute. *PLoS One*. 2017;12(5):1-20. doi:10.1371/journal.pone.0177459
30. Blomer J, Aguado-Sánchez C, Buncic P, Harutyunyan A. Distributing LHC application software and conditions databases using the CernVM file system. *J Phys Conf Ser*. 2011;331(4):042003. doi:10.1088/1742-6596/331/4/042003
31. Condurache C, Collier I. CernVM-FS – beyond LHC computing. *J Phys Conf Ser*. 2014;513(3):032020. doi:10.1088/1742-6596/513/3/032020
32. CERN. The large hadron collider; 2018. https://home.cern/science/accelerators/large-hadron-collider
33. The GNU C library (glibc). https://www.gnu.org/software/libc/
34. Sandberg R, Goldberg D, Kleiman S, Walsh D, Lyon B. Design and implementation of the sun network filesystem. Proceedings of the Summer USENIX conference; 1985:119-130.
35. Howard JH, Kazar ML, Menees SG, et al. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*. 1988;6(1):51-81. doi:10.1145/35037.35059
36. Free icons designed by Smashicons | Flaticon. https://www.flaticon.com/authors/smashicons
37. Gentoo Linux project: prefix. https://wiki.gentoo.org/wiki/Project:Prefix
38. NixOS: reproducible builds and deployments. https://nixos.org/
39. GNU Guix. https://guix.gnu.org/
40. Boissonneault M, Oldeman BE, Taylor RP. Providing a unified software environment for Canada's national advanced computing centers. Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines PEARC '19; 2019:1-6; Association for Computing Machinery, New York, NY. doi:10.1145/3332186.3332210
41. McLay R, Schultz KW, Barth WL, Minyard T. Best practices for the deployment and management of production HPC clusters. Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis SC'11; 2011:1-11. doi:10.1145/2063348.2063360
42. Culpo M, Becker G, Gutierrez CEA, Hoste K, Gamblin T. archspec: a library for detecting, labeling, and reasoning about microarchitectures. Proceedings of the 2020 2nd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC); 2020:45-52. doi:10.1109/CANOPIEHPC51917.2020.00011
43. FUSE implementation for overlayfs; 2021. https://github.com/containers/fuse-overlayfs/
44. Ansible is simple IT automation. https://www.ansible.com/
45. Giardine B, Riemer C, Hardison RC, et al. Galaxy: a platform for interactive large-scale genome analysis. *Genome Res*. 2005;15(10):1451-1455. doi:10.1101/gr.4086505
46. Terraform by HashiCorp. https://www.terraform.io/
47. GNU Binutils. https://www.gnu.org/software/binutils/
48. GCC. The GNU compiler collection. https://gcc.gnu.org/
49. Karakasis V, Manitaras T, Rusu VH, et al. Enabling continuous testing of HPC systems using ReFrame. In: Juckeland G, Chandrasekaran S, eds. *Tools and Techniques for High Performance Computing*. Springer International Publishing; 2020:49-68.
50. Open MPI: open source high performance computing. https://www.open-mpi.org/
51. ABI compatibility initiative - MPICH. https://wiki.mpich.org/mpich/index.php/ABI_Compatibility_Initiative
52. EESSI actions GitHub marketplace. https://github.com/marketplace/actions/eessi
53. ComputeCanada/magic_castle: terraform modules to replicate a compute Canada cluster in the cloud. https://github.com/ComputeCanada/magic_castle
54. Cluster in the cloud documentation. https://cluster-in-the-cloud.readthedocs.io
55. Di Tommaso P, Chatzou M, Floden EW, Barja PP, Palumbo E, Notredame C. Nextflow enables reproducible computational workflows. *Nature Biotechnol*. 2017;35(4):316-319. doi:10.1038/nbt.3820
56. Mölder F, Jablonski K, Letcher B, et al. Sustainable data analysis with Snakemake [version 2; peer review: 2 approved]. *F1000Research*. 2021;10:33. doi:10.12688/f1000research.29032.2
57. Agile software development – Wikipedia. https://en.wikipedia.org/wiki/Agile_software_development
58. MVAPICH: benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/
59. Horizon Europe. https://ec.europa.eu/info/research-and-innovation/funding/funding-opportunities/funding-programmes-and-open-calls/horizon-europe_en
60. Köhler M, Saak J. FlexiBLAS – A flexible BLAS library with runtime exchangeable backends. LAPACK working note; 2014.
61. ofiwg / libfabric GitHub. https://github.com/ofiwg/libfabric
62. Shamis P, Venkata MG, Lopez MG et al. UCX: an open source framework for HPC network APIs and beyond. Proceedings of the 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects; 2015:40-43; IEEE. doi:10.1109/HOTI.2015.13
63. The extreme-scale scientific software stack. https://e4s-project.github.io
64. OpenHPC. http://www.openhpc.community/
65. LearnHPC – scalable HPC training. http://www.learnhpc.eu
66. Configuration – JUSUF user documentation documentation. https://apps.fz-juelich.de/jsc/hps/jusuf/cluster/configuration.html
67. UEABS / ueabs GitLab. https://repository.prace-ri.eu/git/UEABS/ueabs/#gromacs

68. FlexiBLAS – A BLAS and LAPACK wrapper library with runtime exchangable backends. https://www.mpi-magdeburg.mpg.de/projects/flexiblas
69. Todd Gamblin spack. http://scalability-llnl.github.io/spack/
70. cvmfsexec. https://github.com/cvmfs/cvmfsexec
71. ReFrame. https://reframe-hpc.readthedocs.io/en/stable/
72. Nextflow. https://www.nextflow.io/
73. Snakemake. https://snakemake.readthedocs.io/

SUPPORTING INFORMATION

Additional supporting information may be found online in the Supporting Information section at the end of this article.

How to cite this article: Dröge B, Holanda Rusu V, Hoste K, van Leeuwen C, O'Cais A, Röblitz T. EESSI: A cross-platform ready-to-use optimised scientific software stack. *Softw Pract Exper*. 2023;53(1):176-210. doi: 10.1002/spe.3075

APPENDIX A. STRUCTURE OF THE PILOT REPOSITORY

```
/
- cvmfs
    |- pilot.eessi-hpc.org
       |-- host_injections -> /opt/eessi
       |-- latest -> 2021.06
       |-- 2021.06
           |-- compat
           |    |-- linux
           |         |-- aarch64
           |         |-- ppc64le
           |         |-- x86_64
           |              |-- bin
           |              |-- etc
           |              |-- lib64
           |              |-- usr
           |-- init
           |-- software
                |-- linux
                     |-- aarch64
                     |    |-- generic
                     |    |-- graviton2
                     |-- ppc64le
                     |    |-- generic
                     |    |-- power9le
                     |-- x86_64
                          |-- amd
                     |    |-- zen2
                     |    |-- zen3
                     |-- generic
                     |-- intel
                          |-- haswell
                          |-- skylake_avx512
                              |-- modules
                              |-- software
```

```
|-- EasyBuild
|    |-- 4.4.1
|-- GROMACS
|    |-- 2020.1-foss-2020a-Python-3.8.2
|    |-- 2020.4-foss-2020a-Python-3.8.2
|-- TensorFlow
     |-- 2.3.1-foss-2020a-Python-3.8.2
```

Note here that the `host_injections` directory is a *variable* symlink which can be configured in the CernVM-FS configuration file to point to any location in the host (e.g., to a location on a shared file system).

APPENDIX B. DETAILS OF PERFORMANCE BENCHMARKS

The test system used for performance comparison was the JUSUF[66] system at Juelich Supercomputing Centre. For our test case we used the standard compute nodes, of which there are 144, each with two AMD EPYC 7742 CPUs (with 64 cores, so 128 cores per node) and InfiniBand HDR100 (Connect-X6) interconnect.

The test case used in the benchmarks is taken from the PRACE Unified European Applications Benchmark Suite for GROMACS[67] and uses the `lignocellulose-rf` dataset ('Test Case B'). This dataset is intended for Tier-0 systems. In all cases, the options provided to the `gmx_mpi` executable when running the tests were:

```
mdrun -s lignocellulose-rf.tpr -maxh 0.50 -resethway -noconfout -nsteps 20000 -g
logfile -dlb yes
```

For the EESSI installation, the performance-relevant options given to CMake when building GROMACS for the AMD Zen2 micro-architecture were:

```
-DCMAKE_SYSROOT=/cvmfs/pilot.eessi-hpc.org/2021.06/compat/linux/x86_64
-DCMAKE_C_COMPILER='/cvmfs/.../x86_64/amd/zen2/software/OpenMPI/4.0.3-GCC-9.3.0/bin/mpicc'
-DCMAKE_C_FLAGS='-O2 -ftree-vectorize -march=native -fno-math-errno -fopenmp'
-DCMAKE_CXX_COMPILER='/cvmfs/.../x86_64/amd/zen2/software/OpenMPI/4.0.3-GCC-9.3.0/bin/mpicxx'
-DCMAKE_CXX_FLAGS='-O2 -ftree-vectorize -march=native -fno-math-errno -fopenmp'
-DCMAKE_Fortran_COMPILER='/cvmfs/.../x86_64/amd/zen2/software/OpenMPI/4.0.3-GCC-9.3.0/bin/mpifort'
-DCMAKE_Fortran_FLAGS='-O2 -ftree-vectorize -march=native -fno-math-errno -fopenmp'
-DCMAKE_FIND_USE_PACKAGE_REGISTRY=FALSE
-DGMX_DOUBLE=OFF
-DGMX_GPU=OFF
-DGMX_PREFER_STATIC_LIBS=OFF
-DGMX_EXTERNAL_BLAS=ON
-DGMX_EXTERNAL_LAPACK=ON
-DGMX_OPENMP=ON
-DGMX_BLAS_USER="/cvmfs/.../x86_64/amd/zen2/software/OpenBLAS/0.3.9-GCC-9.3.0/lib/libopenblas.a"
-DGMX_LAPACK_USER="/cvmfs/.../x86_64/amd/zen2/software/OpenBLAS/0.3.9-GCC-9.3.0/lib/libopenblas.a"
-DBUILD_SHARED_LIBS=ON
```

From this snippet, we can see that the compiler used was GCC 9.3.0 with Open MPI 4.0.3 and the OpenBLAS 0.3.9 math libraries. This toolchain was used by EESSI and were also used to build any required dependencies. GROMACS was compiled with compute kernels optimised for the appropriate micro-architectures, using the auto micro-architecture detection option (`-DGMX_SIMD=auto`) from their CMake configuration.

For the JUSUF system installation the performance-relevant options given to CMake when building GROMACS were:

```
-DCMAKE_INSTALL_PREFIX=/p/software/jusuf/stages/2020/software/GROMACS/2020.4-gpsmkl-2020
-DCMAKE_C_COMPILER='mpicc'
-DCMAKE_C_FLAGS='-O2 -ftree-vectorize -march=native -fno-math-errno -fopenmp'
-DCMAKE_CXX_COMPILER='mpicxx'
-DCMAKE_CXX_FLAGS='-O2 -ftree-vectorize -march=native -fno-math-errno -fopenmp'
-DCMAKE_Fortran_COMPILER='mpif90'
-DCMAKE_Fortran_FLAGS='-O2 -ftree-vectorize -march=native -fno-math-errno -fopenmp'
-DGMX_DOUBLE=OFF
-DMKL_LIBRARIES="\${MKLROOT}/.../libmkl_intel_ilp64.so;...;\${MKLROOT}/.../libmkl_core.so"
-DGMX_CUDA_TARGET_SM="60;70;80"
-DGMX_GPU=ON
-DCUDA_TOOLKIT_ROOT_DIR=/p/software/jusuf/stages/2020/software/CUDA/11.0
-DGMX_PREFER_STATIC_LIBS=OFF
-DGMX_EXTERNAL_BLAS=ON
-DGMX_EXTERNAL_LAPACK=ON
-DGMX_OPENMP=ON
-DGMX_FFT_LIBRARY=mkl
-DMKL_INCLUDE_DIR="\$EBROOTIMKL/mkl/include"
-DBUILD_SHARED_LIBS=ON
```

The primary difference in compilation here is the toolchain used. In this case, the compiler used was also GCC 9.3.0 but with ParaStation MPI 5.4.7 (the vendor supported MPI implementation) and the Intel MKL 2020.2.254 math libraries (which EESSI cannot distribute without licence clarification). Again, this toolchain was also used to compile any dependencies. We note that in this case GROMACS was compiled with GPU-support (since JUSUF also has GPU nodes), but GPUs were not included in our tests: all results are for CPU-only executions.

APPENDIX C. GLOSSARY

- MPI: Message passing interface, a standard for message passing on parallel computing architectures.[19] The standard has been implemented in various libraries, such as Open MPI, MPICH, Intel MPI, MVAPICH and so forth.
- ABI: Application binary interface, an interface between two binary program modules. An ABI defines how data structures and computational routines are accessed in machine code.
- ISA: Instruction set architecture, an abstract model of the instructions, data types, registers and so forth that are supported by a processor. Examples of instruction sets are x86, ARM, Power, MIPS and RISC-V. Note that, for example, x86 is often called a *family* of instruction sets, since there are many extensions to the original instruction set (such as AVX, AVX2 and AVX512 vector instructions).
- Microarchitecture: The implementation of an ISA in actual hardware. Examples are Intel Skylake and Cascade Lake, AMD Zen 2 and Zen 3, ARM Cortex-A78, Nvidia Ampere, AMD Graphics Core Next.
- Environment modules: Allow an end user to dynamically change their environment by loading module files, thereby setting the correct environment to use a particular (version of an) application. This enables offering and using multiple versions of the same software on a shared (multi-tenant) system.

APPENDIX D. PROJECTS/TOOLS OVERVIEW

- Open MPI: A library that implements the MPI standard.[17,50]
- MPICH: Another library that implements the MPI standard.[18]

- BLAS: Basic Linear Algebra Subprograms, a collection of linear algebra routines that provide common vector-vector, matrix-vector and matrix-matrix operations.[23] BLAS (sometimes referred to as Netlib BLAS, to distinguish it from other BLAS implementations) is a reference implementation of the specification set by the BLAS Technical Forum: it defines how BLAS routines should behave, but is not necessarily as well optimised as other BLAS implementations might be.
- LAPACK: Linear Algebra PACKage, a collection of routines for solving high level linear algebra problems, such as solving systems of simultaneous linear equations, eigenvalue problems, singular value decomposition, matrix factorizations and so forth.[24] LAPACK routines perform as much of their computation as possible through calls to BLAS library functions.
- FlexiBLAS: A BLAS and LAPACK wrapper library that allows switching between different BLAS and LAPACK implementations at runtime.[60,68]
- glibc: The GNU C library.[33] This provides the core libraries for the GNU/Linux systems. It provides critical APIs for basic functionalities such as opening files, allocating memory, creating threads and so forth.
- binutils: A collection of programs that facilitates compiling and linking programs.[47] The main ones are `ld` (the GNU linker) and `as` (the GNU assembler).
- EasyBuild: A software build and installation framework targeted at providing scientific software installations on HPC systems.[14]
- Spack: A software build and installation framework targeted at providing scientific software installations on HPC systems.[69]
- Gentoo Prefix: A variant of the Gentoo Linux OS that supports installing Gentoo in a non-standard location (a prefix).[37]
- Singularity: A container system for HPC.[29]
- NFS: Network File System is a widely used distributed file system protocol originally developed by Sun Microsystems.[34] It allows a user on a client computer to access files over a computer network similar to how local storage is accessed.
- AFS: Andrew File System is a distributed file system developed by Carnegie Mellon University as part of the Andrew project.[35] It allows a user on a client computer to access files over a computer network similar to how local storage is accessed.
- CernVM-FS: A read-only, globally distributed filesystem that is optimised for distributing software.[30]
- cvmfsexec: A package that allows mounting cvmfs as an unprivileged user, without the cvmfs package being installed by a system administrator.[70]
- archspec: A Python package that aims to detect various aspects of a system, such as the CPU microarchitecture, and that models (and can be queried for) compatibility relationships between CPU microarchitectures.[42]
- Lmod: Lua (environment) modules. Environment modules provide a convenient way to dynamically change the user's environment through module files. This is typically used to support multiple versions of the same software on the same system.[15,26,41]
- ReFrame: A software testing framework aimed at testing HPC software. As such, it can interact with various batch schedulers to launch (and test) multi-node parallel applications.[49,71]
- Terraform: An open-source tool designed to provision computing infrastructure.[46]
- Ansible: An open-source tool that allows automation of (cloud) provisioning, configuration management and application deployment.[44]
- Magic Castle: A software project that allows creating an virtual HPC infrastructure in the cloud, using Terraform.[53] It mounts the ComputeCanada or EESSI software stack through CernVM-FS.
- Cluster in the Cloud: A software project that allows creating a virtual HPC cluster in the cloud, using Terraform.[54]
- Nextflow: A framework for scalable and reproducible scientific workflows using software containers.[55,72] It abstracts the software pipeline logic from the execution layer, so that it can be executed on multiple platforms (HPC job schedulers, cloud platforms, etc.) without changing the pipeline. Parallelization is implicitly defined by the input and output declaration of the processes that make up the pipeline.

- Snakemake: A workflow management system to create reproducible and scalable data analyses.[56,73] A single workflow definition can automatically scale to server, cluster, grid and cloud environments, without requiring modifications.
- OSU Micro Benchmarks: Suite of MPI performance benchmarks developed by Ohio State University.[58]