Continuous Integration and Delivery for HPC

Using Singularity and Jenkins

Zebula Sampedro
University of Colorado Boulder
Research Computing
Boulder, Colorado
sampedro@colorado.edu

Aaron Holt
University of Colorado Boulder
Research Computing
Boulder, Colorado
aaron.holt@colorado.edu

Thomas Hauser
University of Colorado Boulder
Research Computing
Boulder, Colorado
thomas.hauser@colorado.edu

ABSTRACT

Continuous integration, delivery, and deployment (CICD) is widely used in DevOps communities, as it allows for teams of all sizes to deploy rapidly-changing hardware and software resources quickly and confidently. In this paper, we will describe how University of Colorado Boulder Research Computing has adopted these practices on the RMACC Summit supercomputer [17] to allow system engineers and researchers alike to capitalize on the benefits of CICD-centric development workflows. We will introduce the topic of CICD at a high level and describe how such practices can ease common software management challenges for High-Performance Computing (HPC) resources. We will then document the infrastructure deployed for Summit, and explain how software such as Jenkins and Singularity enabled adaptation for an HPC environment. We will conclude with two case studies discussing the use of our CICD infrastructure: one case study from the perspective of a system engineer maintaining user-facing resources, and the other case study from the perspective of a researcher developing, maintaining, and using the MFiX-Exa codebase.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**; **Software libraries and repositories**; *Software testing and debugging*; *Software version control*; *System administration*; Maintaining software;

KEYWORDS

continuous integration, continuous delivery, continuous deployment, Singularity, containers, software builds, software automation, MFIX-Exa

ACM Reference Format:
Zebula Sampedro, Aaron Holt, and Thomas Hauser. 2018. Continuous Integration and Delivery for HPC: Using Singularity and Jenkins. In *PEARC '18: Practice and Experience in Advanced Research Computing, July 22–26, 2018, Pittsburgh, PA, USA.* ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/3219104.3219147

PEARC '18, July 22–26, 2018, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-6446-1/18/07.
https://doi.org/10.1145/3219104.3219147

1 INTRODUCTION

HPC resources have complex and dynamic software needs that are challenging to manage and maintain. Users often want the latest software available for their research or development, which drives the need for frequent installation and updates. Since clusters are most often a shared resource with specialized hardware, software must be managed centrally by a system engineer to ensure that it has been optimized for the resource and functions with existing software in the stack. Administrators currently address these complex software needs with tools such as Lmod environment modules, Easybuild [21], Spack [20], and Puppet [9]. However, even with these tools software management in an HPC environment includes manual and error-prone steps. Newly installed software needs to be tested and benchmarked to ensure users make optimal use of the HPC resource. Users developing HPC software need to build their code and run similar correctness tests and benchmarks. Administrators and developers deploying other user-facing services such as gateways need to build and test those environments as well.

While HPC software management is a complex process, it shares many of the same challenges with enterprise-scale software development which requires reliable updates and validation on a demanding schedule. In the past several years industry leaders have established a set of practices referred to as Continuous Integration and Continuous Delivery (CICD) to address these challenges. CICD is comprised of two phases of a modern software delivery pipeline that fully automate the transformation of code changes into the release of viable production software. In this context *Continuous* emphasizes that the full pipeline is run automatically after each commit. This approach obviates the need for complex merge and test procedures prior to release, because each commit is transformed immediately into releasable software. Automation is critical to these phases as it ensures that each commit is subject to the same procedure to build, test, and release. It eliminates error-prone, manual steps in the software build and validation cycle. Another core tenet of CICD is that software passes through multiple phases of automated test suites, ensuring that all software that passes build is production-ready.

While CICD has traditionally focused on the software development lifecycle, the advent of practices such as *Everything as Code* and new solutions like containers have extended CICD to deliver complete software environments instead of just single applications. The evolved capabilities of modern CICD practices to test, manage, and deploy full application environments has compelling implications for the HPC software stack. Managing the stack as a single unit is no longer necessary as it can be decomposed into smaller, more testable components. University of Colorado Boulder

Research Computing has begun to employ modern CICD practices to simplify the software the management process and increase the frequency and reliability of releases. In this paper, we will describe the CICD architecture currently deployed at CU Boulder, and detail how it enables us to automatically build, manage, and deploy containerized software stacks for use on the RMACC Summit Supercomputer. We will then provide two case studies of how this architecture is being used to improve the software delivery lifecycle both from the perspective of a researcher maintaining the MFiX-Exa codebase, and a system engineer managing user-facing services.

2 RELATED WORK

Managing and automating the configuration of the heterogenous hardware and software environments that comprise HPC resources is commonly handled by configuration management tools like Puppet. CU Boulder Research Computing and other HPC providers have employed this method with great success. Configuration management reduces the complexity of cluster environments by consolidating configuration into a single source, and allows for infrastructure to be defined as code [9, 22, 27].

There have been multiple efforts to realize the benefits of containerized software environments in an HPC context. These solutions most often take the form of HPC workloads containerized with Docker, allowing for the jobs to run in an arbitrary software environment that would not have been possible on the host system [23]. Security concerns with the use of Docker on a shared resource [19] have led to the rise of several containerization solutions built specifically for the needs of HPC, such as Singularity [24], Shifter [18], and CharlieCloud [25].

Jenkins has been utilized by several groups to facilitate automated performance monitoring of both scientific software and hardware resources. Vergara et al. use a CI workflow powered by Jenkins to automate application-level benchmarking to determine the impact of hardware and software changes on code performance [28]. Voss et al. have used Jenkins for a similar purpose of automatically benchmarking the performance of the Stampede2 and Lonestar5 supercomputers [29].

3 ARCHITECTURE

We wanted to develop a CICD infrastructure that would support the development and management of scientific codes and software environments for HPC end-users using container-centric workflows. Containerization was our targeted strategy because it enables us to decompose software environments traditionally managed by hand into testable, distributable components. Even where a configuration management solution like Puppet manages specific software environments, breaking the environment out into smaller, self-contained units enables a higher degree of reliable testing while being more durable with respect to hardware and related environment changes.

Our software, the Singularity specification files, the deployment infrastructure, and even the build pipeline are treated like application code. This approach ensures that all aspects of software management conform to the same levels of rigor as software, and provides greater confidence in the provenance and reliability of changes. As a result of this everything-as-code approach, all changes originate in a version control system and propagate down the pipeline.

Jenkins serves as the primary interface between version control systems and deployment, and it also manages the container build process for each image repository. For each project that we wish to automate we configure a Jenkins job that polls the appropriate repositories for changes, and starts a build when a change is made. The build process clones the necessary repositories, and then builds the image with Singularity. In each of our software repositories, we place a Singularity recipe file [10] alongside application code, mirroring the structure of image repositories in Docker Hub [3] and Singularity Hub.

If the tests run in the Singularity `%test` block [10] pass, then Jenkins proceeds with the build. Otherwise, Jenkins notifies via email that the build has failed. This workflow is illustrated in Figure 1. If the container build and test cycles succeed, Jenkins then uses the Singularity Global Client [11] to register and store the image in Singularity Registry for consumption by downstream services like configuration management or benchmarking jobs.

3.1 Singularity and Singularity Registry

Singularity is a containerization solution created specifically for the needs of scientific software, allowing for mobility of compute and reproducible science [24]. Singularity employs a different security model than Docker which allows for built containers to be run as an unprivileged user and does not require that the user interacts with a privileged daemon process [24]. This security model made Singularity an ideal choice for our infrastructure, as we are automating the build of software environments that are either user-facing or directly executable by the user.

Singularity Registry is a decentralized infrastructure for managing and publishing Singularity images, which was generalized from the Singularity Hub cloud service to allow for deployment at individual institutions [26]. It has been optimized for local image storage and provides features that allow for on-site authentication services and container access control. In our pipeline the registry takes the role of an artifact repository, as seen in Figure 1; it provides a mechanism to store, version, and publish containers built during continuous integration by Jenkins. We have configured our deployment of the registry to use our internal GitLab server [4] as an OAuth provider for authentication.

3.2 Docker Compose

Singularity is an excellent container solution for the user-facing services produced by our CICD infrastructure. The infrastructure itself though is not intended for direct consumption by our users and therefore does not inherit the same set of security concerns that prompted the use of Singularity. As a result, we chose to containerize the infrastructure using Docker, which is a very mature container solution built for microservices [16]. Using Docker for the infrastructure allows us to leverage a large ecosystem of tools and documentation for container orchestration, which helped us to develop a highly portable solution and increased the variety of build and test jobs that Jenkins could support.

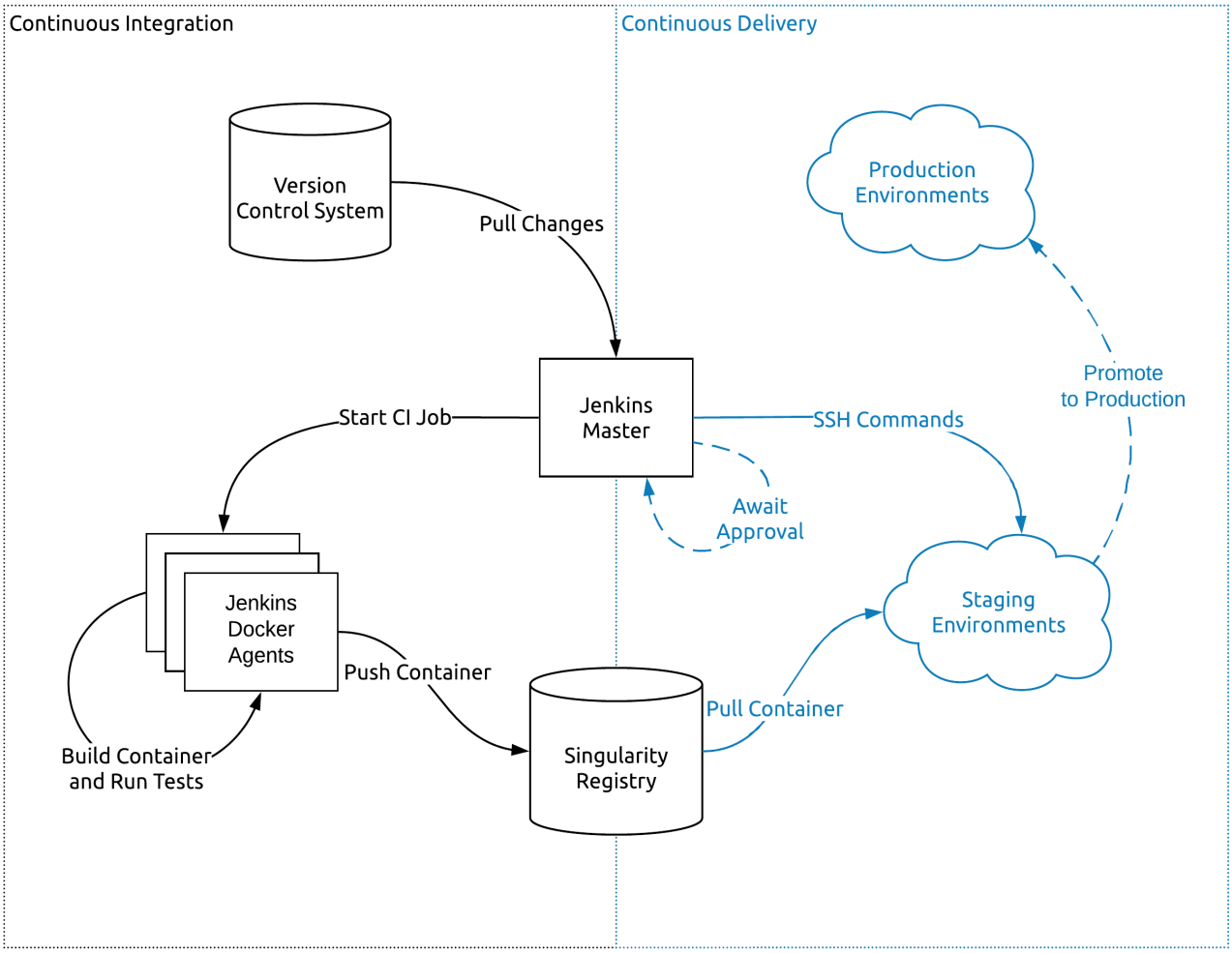

Figure 1: A high-level overview of the pipeline that transforms code changes into deployable software releases.

Compose is an orchestration tool for multi-container Docker applications which defines service configurations in a YAML file, and allows for application management with a simple command line tool [2]. Managing our CICD infrastructure with Compose was a natural fit, as we were already containerizing the Jenkins master and agent software environments with Docker, and we wanted to be able to easily package, distribute, and deploy our infrastructure elsewhere within our resources and potentially to other HPC sites.

We based our Compose configuration off of the one distributed with the Singularity Registry repository [13] and reconfigured some services within it to be used for Jenkins as well as the registry. Dockerfiles for Jenkins agents and configuration for the Jenkins agents and master were added to the registry configuration so that the entire infrastructure could be captured in the same configuration.

3.3 Jenkins

Jenkins is an open-source automation platform [5] that is widely used for building and deploying software. It is highly-extensible via plugins, provides a convenient GUI for creating and editing build processes, and is currently in use at several HPC sites as a general-purpose automation tool [29].

Jenkins supports several different styles of build job configuration and we chose the *Pipeline* method because it is easier to place under version control and is growing in popularity, which is increasing the number plugins and support available. Jenkins pipeline jobs are configured by a specially-formatted Groovy script that defines each step of the build process, and where the step should execute.

The process for cloning a code repository containing a Singularity recipe, building the image, running tests, and publishing via Singularity Registry is consistent between projects, so we have broken the build process out into a *Shared Library*. A Shared Library is a mechanism for separating out components of a Groovy build

pipeline into importable libraries to facilitate reuse. Shared libraries also allow us to version the Singularity build pipeline independently of any project, obviating the need to insert a Jenkinsfile into each project we wish to automate.

Our Jenkins agents run in Docker containers managed by the same Docker Compose configuration as the rest of the infrastructure. Agents are used for container builds and publishing and are thus provisioned with Singularity and the Singularity Registry command-line tool. These agents use the JNLP protocol for communication and registration with the Jenkins master. This approach allows us to scale the number of integration agents using the Docker Compose *scale* command [2] without necessitating a configuration change, as JNLP agents self-register with the master based on a pre-defined secret.

For delivery jobs, we use the Jenkins SSH Plugin which allows for the execution of shell commands on remote hosts [14]. The remote execution allows for increased flexibility in where and how deploy jobs run, as deploy resources can be managed directly from Jenkins without the need to have a remote agent configured and running. Most importantly deployment via the SSH Plugin allows us to defer deployment procedures to systems we already have in place, like Puppet and Slurm.

4 CASE STUDIES

4.1 Deploying User-Facing Software Environments

For many of our users, JupyterHub [6] serves as a primary access point to our cluster resources. JupyterHub is a multi-user proxy and process manager for single-user Jupyter Notebooks, which enable the authoring and execution of interactive code documents [15]. Jupyter Notebooks are capable of supporting many different software stacks and languages and can be extended by way of plugins. While supporting a wide range of Jupyter use cases has increased the productivity of many of our users, it has also necessitated the maintenance of a complex software environment that is sensitive to updates and dependency changes.

We have traditionally managed the Jupyter notebook software environment as part of our environment module stack, which has predominantly been a manual process. Integrating Notebooks into our module stack has also created challenges related to software inter-dependencies, as the Jupyter ecosystem evolves at a more rapid pace than we can accommodate with our module stack. This ultimately leads to an overly rigid set of dependency versions that must be supported. Updating Jupyter and dependent software becomes a tedious process of trial-and-error down the road, discouraging frequent updates and feature additions.

Since many of the changes made to our module stack and the Jupyter notebook integration were performed by hand, we had no mechanism outside of documentation to accurately capture the versions and configuration of the environments we deployed to our cluster. Over time this often led to configuration drift between staging and production environments, and worse still it made it difficult to return to working configurations in the event of a failure.

We wanted to modernize this error-prone, manual process and deliver updates to our users more quickly and confidently. To accomplish this we chose to containerize our Jupyter Notebooks with Singularity. Using containers made it possible to encapsulate a Notebook software environment in a reproducible way, and provide a heightened level of flexibility with respect to how we could compose these environments. We could also use the same image across staging, production, and manually-spawned environments, eliminating configuration drift.

Each container runs a single-user notebook server, and is configured to run either via JupyterHub process management, or manual spawning by the user. The containers are composed of multiple software layers overlaid upon one another during build by the Singularity `FROM` tag. The starting layer is called the *RC Service Base* [12]. This is a CentOS 7 layer configured to allow direct use of important host services from within the container, including SSSD for identity management and Slurm for job scheduling [30]. The RC Services Base is extended by the *Jupyter Base*. This layer installs dependencies for JupyterHub so that Notebooks may be spawned directly by the user or via JupyterHub, and configures the environment so the Notebook servers use locations within standard Singularity bind mount locations for user configuration and process files. These two base layers ensure basic functionality of notebook servers in our system. Further layers that provide additional software to the notebook environment, such as PySpark or R, can build on these base layers. This approach has proved valuable as it provides a reproducible, reliable method for distributing a variety of software environments usable on our cluster resources. Breaking up dependencies also allows for a variety of other software services to be constructed using shared components that have already been verified. This layered approach has made it possible to start distributing containers for use in our tutorials and training offerings with minimal setup and validation.

To ensure proper functionality of each container, we have added environment tests in the Singularity recipe `%test` block for each layer. These tests run before the Jenkins build registers the new container with the registry, and are used mostly for verifying basic software functionality. Once a container passes validation and is pushed to the registry, the Jenkins build job uses the SSH Plugin to pull the container to our JupyterHub staging node for manual verification if the commit was to the current development branch. We hope to expand our automated tests to cover end-to-end workflows in our staging environments, but for now, we are using the `input` step of the Jenkins pipeline plugin to await manual approval before continuing. The pipeline tags the container as the current release candidate after manual approval was received.

Containerizing our user-facing Jupyter notebook software stacks has made it possible for us to deliver more feature-rich environments to our users quickly and confidently. These new environments are no longer susceptible to general resource changes and can be used as building blocks for more user-facing software. The portability of these environments enables us to, with a single container, support multiple different use mechanisms. Automating the delivery and deploy process ensures that each environment change is automatically tested where possible, and imposes structure to manual verification where necessary. Despite the manual verification steps, we have obviated the need for any by-hand changes to staging and production environments, which has improved the accuracy of our updates tremendously.

4.2 MFiX-Exa Benchmark Automation

The multiphase flow with interphase exchanges (MFiX) project [8] is a computational fluid dynamics (CFD) code developed by by the National Energy Technology Laboratory (NETL). MFiX-Exa [7] is a rewrite of the original MFiX built on top of AMREX [1] to enable it to run on future exascale computing systems. Running MFiX-Exa at exascale will allow for industrially relevant problems to be solved in reasonable wall time. With support of DOE grant FE0026298, University of Colorado Boulder Research Computing is contributing to MFiX-Exa by focusing on single node performance aspects of the code.

A typical software engineering paradigm is correctness first, performance later. Exascale software, however, must be designed with performance in mind to be successful. Much like code correctness, the performance of exascale software needs to be monitored while it is developed so that improvements and regressions can be tracked. Similar to other CFD codes, the performance of MFiX-Exa changes based on the problem it is simulating. MFiX-Exa has a 'tiny profiler' built-in which tracks the total time for a run as well as inclusive and exclusive function timings. Using the 'tiny profiler', a variety of test cases are run to measure the performance of MFiX-Exa.

Like many other CFD codes, testing MFiX-Exa for correctness is not a straightforward task. The MFiX-Exa repository includes many smaller simulations which are run in the Singularity `%test` block that runs and verifies small physical simulations. However, the correctness of larger simulations is not certain even if these tests pass. Fortunately, one of the performance benchmarks we chose has a theoretical result which can be plotted and compared to the simulated result. While this comparison adds a manual step to the pipeline, it boosts our confidence in the results.

The MFiX-Exa CICD pipeline starts with the Jenkins Master polling the develop branch for new code commits three times a week. A typical CI pipeline might build the project following every commit, but the allocated compute time for the MFiX-Exa project limits the scope and number of benchmarking jobs that can be run. When new commits are found, Jenkins JNLP agents build MFiX-Exa in a Singularity container. The MFiX-Exa Singularity container consists of two layers: a base layer which includes MFiX-Exa's dependencies and an mfix layer which installs the latest version of MFiX-Exa. MFiX-Exa's dependencies change infrequently, which means the base layer rarely needs to be built. This saves time for most builds as only the mfix layer needs to be built. After the container is built, the tests in the Singularity `%test` block are run. If the tests pass, Jenkins pushes the container to the Singularity registry. Next, we use the Jenkins SSH Plugin to submit a Slurm job which pulls the latest container from Singularity Registry and benchmarks MFiX-Exa. At the end of the benchmarking job, a Python script using Numpy and Matplotlib plots benchmarking results and correctness comparisons where applicable.

Figure 2 shows an example benchmarking result created with the CICD pipeline. Specifically, Figure 2 displays the weak scaling results for the Homogeneous Cooling System (HCS) case versus date and commit. Figures like these help associate performance changes to commits without manually benchmarking and plotting results.

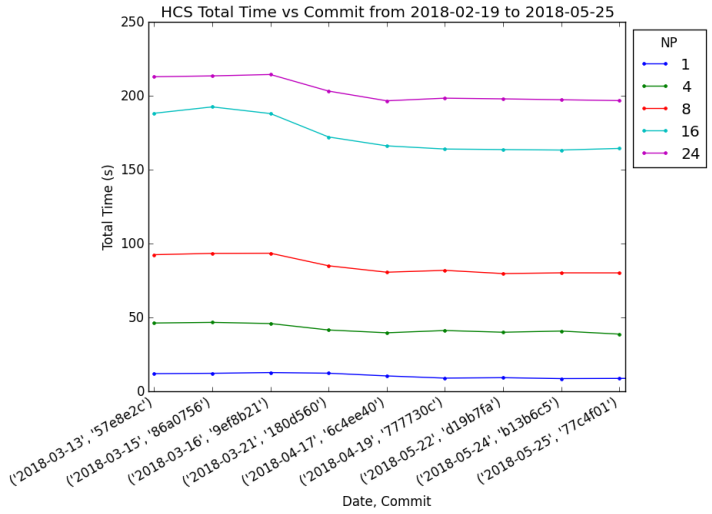

Figure 2: Change in performance of the HCS case over time for a variety of processor counts.

The CICD pipeline solves many reproducibility and portability issues prevalent in HPC software development. When discussing the latest benchmark results, common issues occur such as:

- Comparing results from different commits and branches
- Comparing simulations where different compile flags were used
- Comparing scaling results from simulations with different parameters
- Comparing results from different compilers and MPI implementations
- Comparing results from different operating systems
- Comparing results on different hardware (network, CPU, etc)

The MFiX-Exa Singularity container documents the build process, ensures that the same compiler and MPI implementation are used, tests the code, and can be run to verify the performance on a variety of hardware. The Singularity Registry facilitates sharing containers and stores previous Singularity containers which can be reused to verify previous results. Jenkins automates the process and eliminates most manual steps. While some performance and correctness comparisons are still manual, being able to skim through a few plots greatly reduces the amount of time spent tracking down regressions.

5 CONCLUSIONS

The complexity of HPC software stacks and the growing user demand for new and up-to-date software necessitates that maintainers of HPC resources employ solutions that enable frequent and reliable updates to their resources. The advent of containerization solutions that are viable in an HPC context has provided a mechanism for separating certain types of HPC applications into self-contained, reproducible, and testable environments. In this paper we have demonstrated one approach to this problem that integrates Singularity containers with common automation tools like Jenkins and

Puppet to facilitate the adoption of CICD practices into HPC workflows, increasing the potential for delivering high-quality, reliable software.

ACKNOWLEDGMENTS

This work utilized the Summit supercomputer, which is supported by the National Science Foundation (awards ACI-1532235 and ACI-1532236), the University of Colorado Boulder, and Colorado State University. The Summit supercomputer is a joint effort of the University of Colorado Boulder and Colorado State University.

University of Colorado Boulder Research Computing receives funding to improve MFIX from DOE award FE0026298.

REFERENCES

[1] 2018. AMREX: Block-Structured AMR Software Framework and Applications. https://amrex-codes.github.io/. (2018). [Online; accessed 06-March-2018].
[2] 2018. Docker Compose Documentation. https://docs.docker.com/compose/. (2018). [Online; accessed 06-March-2018].
[3] 2018. Docker Hub. https://hub.docker.com/. (2018). [Online; accessed 06-March-2018].
[4] 2018. GitLab- The only product for the complete DevOps lifecycle. https://about.gitlab.com/. (2018). [Online; accessed 06-March-2018].
[5] 2018. Jenkins. https://jenkins.io/. (2018). [Online; accessed 06-March-2018].
[6] 2018. JupyterHub. https://jupyterhub.readthedocs.io/en/stable/. (2018). [Online; accessed 06-March-2018].
[7] 2018. MFiX-Exa: multiphase flow with interphase exchanges for exascale. https://amrex-codes.github.io/MFIX-Exa/docs_html/Introduction.html. (2018). [Online; accessed 06-March-2018].
[8] 2018. MFiX: multiphase flow with interphase exchanges. https://mfix.netl.doe.gov/. (2018). [Online; accessed 06-March-2018].
[9] 2018. Puppet - Get on the shortest path to better software. https://puppet.com/. (2018). [Online; accessed 06-March-2018].
[10] 2018. Singularity. http://singularity.lbl.gov/. (2018). [Online; accessed 06-March-2018].
[11] 2018. Singularity Global Client. https://singularityhub.github.io/sregistry-cli/. (2018). [Online; accessed 06-March-2018].
[12] 2018. Singularity RC Base Image. https://github.com/ResearchComputing/singularity-slurm-base. (2018). [Online; accessed 06-March-2018].
[13] 2018. Singularity Registry. https://doi.org/10.5281/zenodo.1012531. (2018). [Online; accessed 06-March-2018].
[14] 2018. SSH Plugin - Jenkins. https://wiki.jenkins.io/display/JENKINS/SSH+plugin. (2018). [Online; accessed 06-March-2018].
[15] 2018. The Jupyter Notebook. https://jupyter-notebook.readthedocs.io/en/stable/. (2018). [Online; accessed 06-March-2018].
[16] Charles Anderson. 2015. Docker [software engineering]. *IEEE Software* 32, 3 (2015), 102–c3.
[17] Jonathon Anderson, Patrick J Burns, Daniel Milroy, Peter Ruprecht, Thomas Hauser, and Howard Jay Siegel. 2017. Deploying RMACC summit: an HPC resource for the Rocky Mountain Region. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*. ACM, 8.
[18] Richard Shane Canon and Doug Jacobsen. 2016. Shifter: containers for HPC. *Proceedings of the Cray User Group* (2016).
[19] Theo Combe, Antony Martin, and Roberto Di Pietro. 2016. To Docker or not to Docker: A security perspective. *IEEE Cloud Computing* 3, 5 (2016), 54–62.
[20] Todd Gamblin, Matthew LeGendre, Michael R Collette, Gregory L Lee, Adam Moody, Bronis R de Supinski, and Scott Futral. 2015. The Spack package manager: bringing order to HPC software chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 40.
[21] Markus Geimer, Kenneth Hoste, and Robert McLay. 2014. Modern scientific software management using EasyBuild and Lmod. In *HPC User Support Tools (HUST), 2014 First International Workshop on*. IEEE, 41–51.
[22] Val Hendrix, Doug Benjamin, and Yushu Yao. 2012. Scientific Cluster Deployment and Recovery–Using puppet to simplify cluster management. In *Journal of Physics: Conference Series*, Vol. 396. IOP Publishing, 042027.
[23] Joshua Higgins, Violeta Holmes, and Colin Venters. 2015. Orchestrating docker containers in the HPC environment. In *International Conference on High Performance Computing*. Springer, 506–513.
[24] Gregory M Kurtzer, Vanessa Sochat, and Michael W Bauer. 2017. Singularity: Scientific containers for mobility of compute. *PloS one* 12, 5 (2017), e0177459.
[25] Reid Priedhorsky and Tim Randles. 2017. Charliecloud: Unprivileged containers for user-defined software stacks in hpc. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 36.
[26] Vanessa Sochat. 2017. Singularity Registry: Open Source Registry for Singularity Images. *Journal of Open Source Software* 2, 18 (2017). https://doi.org/doi:10.21105/joss.00426
[27] Sébastien Varrette, Pascal Bouvry, Hyacinthe Cartiaux, and Fotis Georgatos. 2014. Management of an academic hpc cluster: The ul experience. In *High Performance Computing & Simulation (HPCS), 2014 International Conference on*. IEEE, 959–967.
[28] Veronica G Vergara Larrea, Wayne Joubert, and Christopher B Fuson. 2015. *Use of Continuous Integration Tools for Application Performance Monitoring*. Technical Report. Oak Ridge National Laboratory (ORNL); Oak Ridge Leadership Computing Facility (OLCF).
[29] Joseph Voss, Joe A Garcia, W Cyrus Proctor, and R Todd Evans. 2017. Automated System Health and Performance Benchmarking Platform: High Performance Computing Test Harness with Jenkins. In *Proceedings of the HPC Systems Professionals Workshop*. ACM, 1.
[30] Andy B Yoo, Morris A Jette, and Mark Grondona. 2003. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 44–60.