Harnessing Configuration Management & Deployment Pipelines

From Weeks to Hours

Dennis Walker
HPC Solutions Engineering
Hewlett Packard Enterprise
Las Vegas, NV - United States
dennis.walker@hpe.com

Siri Vias Khalsa
HPC Advanced Deployment
Hewlett Packard Enterprise
London, England
sirivias.khalsa@hpe.com

Alex Lovell-Troy
OpenCHAMI
Los Alamos National Laboratory
Washington D.C., United States
alovelltroy@lanl.gov

ABSTRACT

Modern high-performance computing (HPC) environments demand rapid, reliable, and repeatable change management across increasingly complex and distributed systems. This whitepaper presents a pragmatic approach to implementing configuration management, continuous integration (CI), and deployment pipelines tailored for HPC software stacks. It outlines how to version control everything—from switch configs and job schedulers to container images and firmware—and automate the orchestration of changes across multi-zone, multi-site environments.

INTRODUCTION

Ensuring peak reliability, current functionality, and up-to-date security requires cultivating the capability to continuously update and integrate a complex array of dependencies, spanning hardware, firmware, system management software, network configurations, API services, OS distros, job schedulers, AI libraries, and analytics tools. This paper presents simple and contemporary DevOps methodologies designed to automate, validate, and replicate changes effectively across multiple production environments.

By creating these capabilities in-house, an organization empowers its vendors to deliver more quickly. This results in providing current functionality and a better guarantee of operational reliability to its stakeholders. Vendors become empowered to provide software earlier in cases where the urgency warrants bypassing potentially lengthy, deep, and complete integration test suites. Stakeholders can experiment with a greater variety of software stacks, which can lead to innovation.

Modernized HPC configurations are increasingly found spanning multiple sites and zones to guarantee cloud-like availability. In such cases, configuration management, infrastructure-as-code, and deployment automation are the only ways to reliably reproduce the many configuration details necessary to provide a coherent user interface across the topologies.

This basic DevOps framework will streamline the update process, providing system security, current functionality, environment reproducibility, and more reliable supercomputing operations. In addition to the methodology, we will provide a tour of example orchestration and pipelines for Cray EX systems, including CSM, HPCM, Altair PBS, Lustre, Slingshot, and OpenCHAMI.

CCS CONCEPTS

• Software Evolution
• Maintaining Software
• System Administration
• Software Version Control
• Maintaining Software

KEYWORDS

DevOps Engineering, Automation, Continuous Integration, Continuous Delivery, Change Management, SDLC, Configuration Management, Infrastructure-as-Code, HPC, CSM, HPCM, OpenCHAMI, System Management

1 CORNERSTONES OF DEVOPS

To begin, it is necessary to enumerate a few foundational best practices for managing change through automation. These concepts are built upon to provide a simple yet comprehensive approach to ensuring configuration version control and change tracking and automating the deployment of these changes across environments with greater velocity,

LA-UR-25-24127

confidence, and consistency.

1.1 DevOps as Culture

At the core of a DevOps culture is the idea that all change should be encapsulated in source code which can then be rapidly and safely progressed from one step in the pipeline to the next. Each step is validated automatically before passing on to the next. Managing the configuration of each portion of the system through a version control system and discrete workflow pipeline allows each member of the development and operations team to achieve the same confidence in each change as it is validated and subsequently applied. Each iteration becomes faster and safer through a culture of collaborative constant improvement.

While the tooling is important, it is hard to overstate the importance culture plays in applying DevOps to any environment, and especially that of HPC. Applying all of the best practices outlined in the rest of the paper will not guarantee desired results. Without a culture that favors collaboration between those tasked with operating a system and those responsible for development of that system, improvements are ad-hoc works of bespoke artisanship and are likely to be temporary. Further, this collaborative approach needs to be reinforced by workflows and tools that refine over time. Automation and version control with defensive testing and reliable documentation allow both operators and developers to cross-train and safely experiment to improve the overall system.

1.2 Version Controlled Environments

To reliably manage the rising complexity in HPC Supercomputers, version control is not just a best practice—it is the bedrock of compliant, performant, and highly available systems administration. HPC data centers are increasingly elastic, integrating dynamic power control, cooling, firmware settings, boot configurations, custom software stacks, run-time personalization, job scheduler topologies, network partitioning, and storage security for tens of thousands of nodes. Managing such systems without comprehensive version control is a recipe for technical debt, configuration drift, and the inability to identify the root causes of operational incidents.

Version control provides traceability across the entire tech stack and lifecycle. Knowing the system's state over time is vital when operational issues arise. In collaborative environments with multiple admins and DevOps personnel, version control ensures that concurrent work doesn't cause conflicting changes. Structured workflows, such as branching and pull requests, allow safe experimentation, peer review, and controlled promotion to production systems.

Version control also facilitates reproducibility and disaster recovery. HPC centers must be able to reconstitute environments across test, staging, and production systems—sometimes across entirely different hardware. Whether managing Slurm configurations, MPI builds, or site-specific provisioning scripts, having everything versioned ensures consistency and reduces deployment friction.

In regulated environments—especially those subject to NIST SP 800-53r5 or similar standards—maintaining detailed, time-stamped audit logs of changes is mandatory. Every configuration update, package version, or policy adjustment must be attributable to a specific individual and event. A well-maintained version control system like Git and proper commit hygiene and access controls satisfy these auditability requirements by default.

Lastly, some products, like CSM, have most of the tooling recommendations below built-in. This paper advocates for externalizing these capabilities as well to replicate configurations across products and environments, and to better handle upgrades or disaster recovery scenarios.

In summary, version control is not optional in HPC system administration—it is foundational. It ensures compliance, reinforces accountability, supports reproducibility, and empowers collaborative management of some of the world's most complex computing environments. If your system has users depending on uptime, integrity, and security, then everything—from cron jobs to cluster configs—must be versioned.

1.3 Utility of Pipelines

A common first step in applying DevOps principles is to focus on automating changes rather than applying them manually. The code used to apply an automated change can then be used to trigger the next change in a line or to halt operation and back out of changes.

LA-UR-25-24127

Automation through scripting and configuration management is essential for applying infrastructure changes consistently and reliably. Manual processes are prone to human error—misconfigurations, skipped steps, or inconsistent setups—that can lead to downtime or unpredictable behavior. By automating these tasks, teams ensure that environments are reproducible, stable, and aligned with best practices, regardless of who executes the change or how often it's applied.

Pipelining automation also enables repeatability across a growing number of environments. Whether deploying to development, staging, or production, scripted processes allow teams to replay the same changes with consistent results. This eliminates configuration drift, simplifies testing and debugging, and ensures that new environments mirror production as closely as possible—critical for scalability and operational confidence.

This transforms infrastructure management from a manual, error-prone task into a disciplined, versionable, and testable process. It supports faster, safer deployments, strengthens collaboration across teams, and helps organizations respond to change with agility and control—core goals of any DevOps practice.

3 PIPELINING CHANGE ACROSS TEN CSM/HPCM ENVIRONMENTS

In this section, we will share how applying these DevOps principles enabled HPE to manage a sprawl of related environments.

Recently, HPE deployed ten interconnected HPC environments, four HPCM and six CSM, spanning two data centers and four zones. To confirm configurations were consistent and ensure change deployment stability, cloud-based CI/CD pipelines were created to:

- Ensure reproducibility.
- Manage node role assignments and multi-tenancy.
- Ensure compatibility across systems, including IdP integrations, IPAM, network isolation, and file system mounts.
- Enable change tracking.
- Manage software configurations and switches.
- Version-control artifacts, images, and packages.
- Expedite hotfixes.
- Simplify communication.
- Automate manual, repetitive effort.

A new superset configuration parameter schema was developed to apply configuration across many environments using one set of playbooks. It was used to specify IP addresses, NAT gateways, file mounts, node roles, sudoers, LDAP servers, and more.

A combination of bash scripts and Ansible was used. Bash, for its simplicity and Ansible, for its idempotency. Where Bash was used, a common framework was developed to provide idempotency and standardized reporting.

Lastly, Azure CI pipelines were used. They need only two parameters during deployment: the git repository branch to source from, and the destination environments to apply to.

The following sections break out how orchestration was implemented in CSM and HPCM environments respectively. Product capabilities that lend themselves well to version-controlled change automation from a CI pipeline are enumerated. The resulting implementation was used to manage change across many environments, greatly reducing the time necessary to manage each, while also ensuring that configurations were consistent. Optional best practice methods are also presented.

3.1 CI-Friendly Facilities Within CSM

Cray System Management (CSM) provides a robust foundation for modern Continuous Integration (CI) pipelines by offering a fully integrated stack of declarative, version-controlled orchestration tools. Designed with automation and repeatability at its core, CSM supports end-to-end CI workflows through a combination of key technologies. Source code is managed through Gitea with version control, while Nexus serves as a centralized artifact repository for containers, Helm charts, RPMs, and binaries. Squashfs images are efficiently stored via S3-compatible Ceph storage, and system configurations are maintained through Ansible-based CFS. Secrets are securely handled using Vault, while Argo enables scalable CI and automated workflows. All of this is wrapped in a Kubernetes/Helm-based infrastructure, with comprehensive REST APIs ensuring programmability across the stack—making CSM an ideal platform for driving consistent, automated, and secure CI operations.

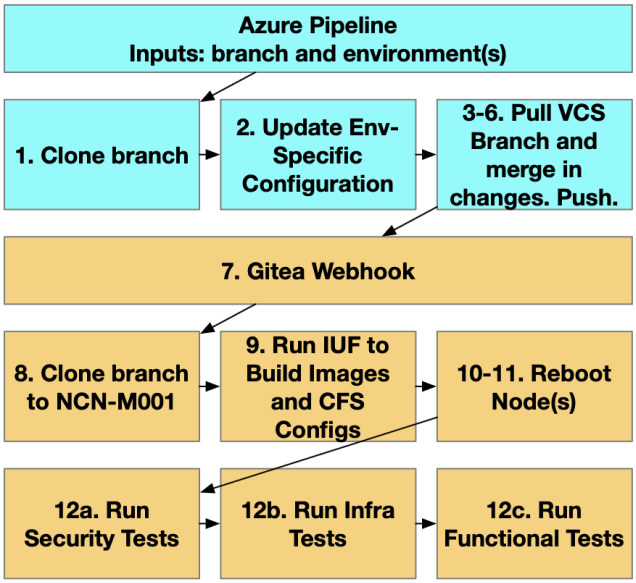
This includes:
- **Gitea** (via VCS) as source code management,

- **Nexus** for artifact warehousing (container registries, helm chart repositories, RPM repositories, and binary/archive file retrieval)
- **S3** (via **Ceph**) for Squashfs image storage
- **Ansible** services (via **CFS**) for centralized configuration management
- **Vault** for secrets / credentials management
- **Argo** for CI and automated workflows
- Comprehensive system management REST APIs
- and a **Kubernetes/Helm** hosting for its management infrastructure

3.2 CI-Orchestrated Compute Updates In CSM

Continuous Integration pipelines need only two parameters during deployment: the git repository branch to source from, and the destination environments to apply to.

Image 3.2.1 - Sequence Diagram of CSM Update Pipeline



For each pipeline:

1. Clones the specified branch into the workspace.

2. Updates any specified environment-specific configuration files based on the provided target environment name parameter.

3. Opens a temporary ssh proxy tunnel into CSM using a limited access service account.

4. Clones the provided feature branch name if it exists in the environment's Gitea instance. If so, an rsync operation mixes in the updated files into the branch, enabling localized change tracking in the Gitea git log.

5. Commits the change, providing a link to the cloud-based repo commit URL in the message. Pushes the branch up to Gitea.

6. Close the pipeline's SSH tunnel to the environment.

7. Within Gitea, a callback webhook hits a custom Flask service running on a management node's Weave interface to kick off local cloning and deployment.

This was done for simplicity. An Argo workflow could have been created instead, which is the recommended approach.

The subsequent steps are performed from orchestration within the environment, with access to a designated NCN, in this case, ncn-m001. An engineer can also perform them separately if confidence in reliability is still settling. Once reliability is known and change is isolated correctly, connecting to the external pipeline is commensurate with the definition and best practice of "Continuous Integration".

8. Clone the VCS repositories to the local file system.

A custom repository contains the IUF and SAT boot prep files. To ensure change isolation, the artifact names are parameterized to include the branch names.

9. Run IUF with limit parameters to create only artifacts, such as IMS images, CFS configuration, and BOS session templates.

10. Reboot select nodes with the new images and configuration.

11. Once the BOS session is complete, run the automated test suite against the node(s) to confirm the expected result.

12. Finally, run a workload on the compute that is exemplary of its normal duty. Validate the result.

3.3 CSM Update Velocity Improvement

In this section, we compare the time to deploy and test a simple change to CSM by hand with the time it takes when assisted with CI pipelines. We then extrapolate the time

savings during development and through a typical production rollout.

Following the steps in 2.1 for CSM, the typical time to build and test a new compute node image in isolation is 8 –16 hours. We'll use 12 hours as the mean amount of time to identify and resolve issues during the test run per environment. This break down as follows:

Table 3.3.1 - Manual execution time to update an image in minutes

Task	Execution Time	Cumulative Time
Stage Ansible in VCS	30	30
Deploy Artifacts in Nexus	30	60
Create CFS Configurations	30	90
Build Images	60	150
Create BOS session template	15	165
Reboot nodes with new config	30	195
Validate changes	15	210
Regression test	30	240

The above table shows that a knowledgeable engineer may spend approximately 4 hours navigating the mechanics of one test run by hand. Additionally, with so many manual steps, encountering setbacks due to mistakes becomes more likely.

This means that any change may take days to develop, no matter how small. And when deploying across 6 environments, it takes days to deploy.

Compare this to the fully automated CI pipeline:

Table 3.3.2 - CI Pipeline to build, boot, and test image changes

Task	Execution Time	Cumulative Time
Stage Ansible in VCS	5	5
Deploy Artifacts in Nexus	1	6
Create CFS Configurations	2	8
Build Images	45	53
Create BOS sessiontemplates	1	54
Reboot nodes with new config	20	74
Validate changes	3	77
Regression test	3	80

The above table shows that feature experiments are deployed, booted, and verified in 1.33 hours using a CI pipeline, **saving 2.66 hours per test run**. Given it is fully Imagautomated, the engineer can work on other items while awaiting feedback. The engineer can reliably start and finish an experiment every day.

Moreover, when deploying to production systems, multiple systems can be deployed in parallel, greatly reducing the time.

To recap, **without a CI pipeline, it may take 4-5 days** to start an experiment, test it, and deploy it to all 6 environments. **With a CI Pipeline, it can realistically be done in 1-2 days.**

3.4 CI-Friendly Facilities in HPCM

HPCM was designed with simplicity in mind. It orchestrates change through imperative commands and scripts. It does not prescribe much compared to CSM. This can still, however, be easily instrumented to drive change in a declarative approach through source code with some additional tooling.

Let's enumerate these options:

- Add local Git in flat files over SSH.
- HPCM provides simple, file-based RPM repository management
- Image storage is the local file system. We version with directory namespacing.
- Add Ansible to manage changes idempotently.
- HPCM stores secret credentials in encrypted, flat files on the admin node.
- Add any external CI tool with access, e.g. Jenkins.
- HPCM has a limited API. It generally assumes you are running commands or scripts on the admin

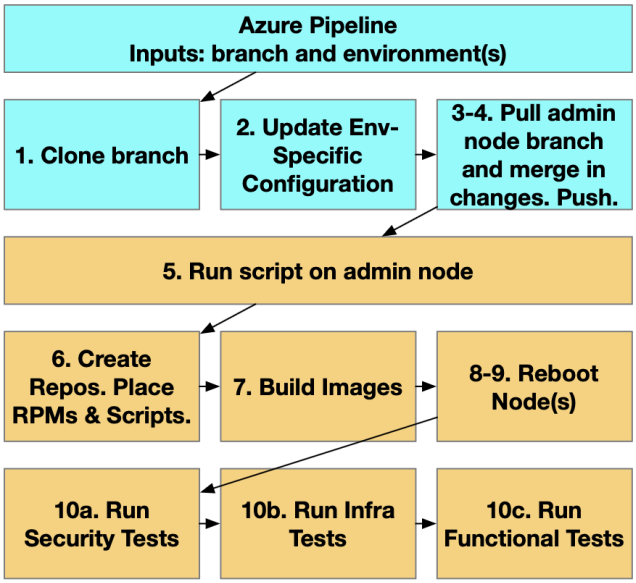
node. Its CLI tooling is well documented and relatively user-friendly.

- Changes to the management infrastructure are rolled out to hotspares, and then the remaining node for admin and leader node types.

3.5 CI-Orchestrated Updates In HPCM

Continuous Integration pipelines need only two parameters during deployment: the git repository branch to source from, and the destination environments to apply to. The pipeline execution follows much of the same process as the CSM pipeline, the differences being the CLI calls and the assumption that the CI pipeline has direct access to the HPCM admin node.

Image 3.5.1 - Sequence Diagram Showing HPCM Compute Image Update



For each pipeline run, the automation
1. Clones the specified branch into the workspace.

2. Updates any specified environment-specific configuration files based on the provided target environment name parameter.

3. Clones the provided feature branch name if it exists in the environment's local git files. If so, an rsync operation mixes in the updated files into the branch, enabling localized change tracking in the git log.

4. Commits the change, providing a link to the cloud-based repo commit URL in the message. Pushes the branch up to the admin node local git.

5. The pipeline then runs a script within the admin node to continue deployment.

The following steps are performed from orchestration within the environment, with access to an admin node. An engineer can also perform them separately if confidence in reliability is still settling. Once reliability is known and change is isolated correctly, connecting to the external pipeline is commensurate with the definition and best practice of "Continuous Integration".

6. Place artifacts. Create repositories and repository groups.

7. Create new image(s).

8. Reboot select nodes with the new images and configuration.

9. Once the reboot is complete, run the automated test suite against the node(s) to confirm the expected result.

10. Finally, run a workload on the compute that is exemplary of its normal duty. Validate the result.

3.6 HPCM Update Velocity Improvement

Here, HPCM environments are used to manage HSN fabrics for storage and PBS pro schedulers.

Table 3.6.1 - Manual steps updating PBS Pro and HPCM

Task	Execution Time	Cumulative Time
Push Ansible to HPCM admin node	15	15
Copy artifacts to HPCM admin node	15	30
Prepare run-time configurations	30	60
Build new images	90	150
Sequentially reboot the HA	45	195

hotspares for PBS and HPCM into the new configuration		
Promote new hotspares to primary VIP and run healthchecks	30	225
Sequentially reboot second nodes for PBS and HPCM	45	255
Validate changes	15	270

The table above shows that each testrun takes 4.5 hours each to navigate the deployment orchestration by hand. Assuming 2-4 test runs per change, that expands to 1-2 days of time to validate simple changes. Rolling this out to all environments requires another 2 days, resulting in 4 days of time spent in repetitive deployment activity for even the simplest change.

Let's compare this to time spent when assisted by CI pipeline automation.

Table 3.6.2 - CI Pipeline execution time when updating PBS Pro and HPCM

Task	Execution Time	Cumulative Time
Push Ansible to HPCM admin node	1	1
Copy artifacts to HPCM admin node	5	5
Prepare run-time configurations	1	1
Build new images	45	52
Reboot the HA hotspares for PBS and HPCM into the new configuration	20	72
Promote new hotspares to primary VIP and run healthchecks	1	73
Sequentially reboot second nodes for PBS and HPCM	20	93
Validate changes	5	98

The table above shows that each testrun, using CI pipelines, takes a little more than 1.5 hours to deploy and validate changes to both PBS pro schedulers and HPCM admin nodes. Given the orchestration is fully automated, the engineer is able to do other work while it runs, resulting in potentially fewer test runs.

Assuming 2-3 test runs per experiment, the total time to validate a configuration change becomes 294 minutes or just under 5 hours. This means, a change to configuration can be started and validated in less than 1 day, and deployed to all environments in the following day.

3.7 Environment Promotion

In a Software Development Life Cycle (SDLC), *environment promotion* is the structured process of moving software changes through a series of environments, each with a distinct purpose. This ensures that code is progressively validated under conditions that resemble production more closely at each step. The journey typically begins in the **development environment**, where individual developers or teams write and test code. This environment is optimized for flexibility and speed, often featuring mocked services or simplified configurations that allow for rapid iteration and unit testing.

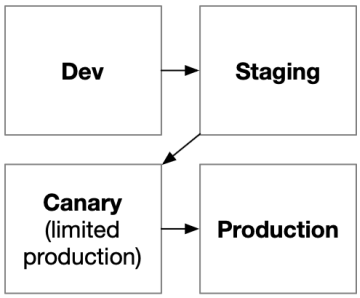
After initial development, the code is promoted to a **staging environment**—also known as *test* or *pre-production*. Staging is designed to closely replicate the production environment in terms of infrastructure, configurations, and sometimes even anonymized real-world data. This is where teams perform integration testing, regression testing, and user acceptance testing (UAT) to ensure that the software works as expected when all system components interact together. It's the final checkpoint before live deployment and is critical for catching issues that aren't apparent during isolated development.

For systems requiring extra caution during release, a **canary environment** is often used before full production rollout. In a canary deployment, the new version of the software is released to a small, controlled portion of users or requests in the actual **production environment**. This allows teams to monitor real-world performance, behavior, and errors at a limited scale. If everything performs as expected, the deployment is gradually expanded to the full production environment, making the release available to all users. This

phased approach minimizes risk and enhances reliability across the release process.

In the customer's infrastructure, four CSM environments and two HPCM environments formed a multizone production. The remaining two CSM and two HPCM environments for change validation, one for development and one for staging.

Image 3.7.1 - Environment promotion sequence



4 SPEED OF ITERATION WITH OPENCHAMI

In this section, we will share how the open-source project, OpenCHAMI, is being developed and deployed with DevOps principles in mind, and how it is being applied at Los Alamos National Laboratory(LANL).

Using OpenCHAMI, the Los Alamos team was able to integrate a cluster of new hardware in less than a week. This was much faster than expected. Similar activities in the past have taken over a month. The cluster in question was composed of new node types with new GPUs and unfamiliar BMCs. Without the ability to effectively test before the hardware arrived, the team needed to quickly identify areas for improvement, test the fixes, apply changes through automation, and move on to the next problem. In this loop of finding and fixing issues, it is common to fix the problem only to see it reoccur when the fix has not been applied to the automation. OpenCHAMI anticipates this and makes each iteration as fast as possible, encouraging the team to leverage it rather than batch changes for later.

The sysadmin team highlighted three strategies responsible for saving iteration time when finding and fixing issues. In this case study, we will explain how the focus on reducing the cost of experimentation impacted the overall speed of integration.

1. Rapid image rebuilds using `image-builder` and Gitlab runners.
2. Fast node boots with `cloud-init`
3. External firmware management tooling with `magellan`

4.1 Introducing OpenCHAMI

OpenCHAMI is an open source alternative to commercial options like CSM and HPCM. Built as a collection of independent components that can be used individually or in combination, it takes a more limited view of system management than larger commercial options. This modular approach lends itself well to pipelines. The subsystems responsible for secure and fast provisioning of HPC nodes are completely independent from the Operating Systems that run on the nodes themselves. Both operate independently of network management which is out of scope for OpenCHAMI at the time of writing.

4.2 OpenCHAMI Installation & Upgrade

In the simplest case, OpenCHAMI can be deployed with a set of defaults in a matter of minutes using docker-compose. Upgrades are performed by updating the docker compose file and restarting the containers. Relevant configuration files are grouped in a single directory and suitable for version control along with the docker compose files.

For production use, sites often find that using Kubernetes or SystemD for scheduling containers fits their operational model better. In those cases, upgrades follow the native pattern and version control is external to OpenCHAMI.

In any case, OpenCHAMI upgrades involve preserving data in CSI volumes while updating version tags of each container and restarting them. By simplifying the responsibilities of the system manager, OpenCHAMI is able to sustain downtime without any loss of operation. For failover that must happen faster than the time it takes to reboot the OpenCHAMI node(s), volumes can be disconnected on one head node and reattached on a separate head node without interruption of service on the compute nodes.

4.3 OpenCHAMI Continuous Integration

OpenCHAMI itself has not yet made its first release. Teams at different sites are structuring experiments that will lead to

best practices recommendations. At LANL, the team is testing two pipeline types.

The first has a single OpenCHAMI control plane for one 650 node cluster. Each change affects the entire cluster at once. The second divides a 1200 node cluster among six control planes in which changes happen to at most 200 nodes at a time. Both have benefits. Managing the single control plane is clearly faster, but using pipelines to manage the six control planes is only about twice as much effort and adds a degree of safety since 5/6ths of the system remains unchanged when attempting a change for the first time, catching errors before users are impacted.

4.4 OpenCHAMI Image Management Pipelines

OpenCHAMI doesn't put any requirements on the Operating System for sites. However, many sites complained about the lack of an open source option for building squashfs system images as quickly and easily as application containers. OpenCHAMI decided to release an open source image-builder for that purpose.

It leverages buildah to create and update images with a docker-like layer model. In the simplest example, it imports an image from an OCI registry or S3 location and then adds or removes content from it through additional layers. In the LANL pipelines, a base image is updated only when necessary, but subsequent layers can be updated much more frequently.

Using the built-in pipelines included with GitLab, a commit that changes a single layer may trigger rebuilds of multiple system images as upper layers are reapplied. If the build of a lower layer is not successful, subsequent builds aren't attempted. Changing upper layers commonly takes seconds.

When testing different versions of AI tooling, the iteration speed was measured in seconds rather than tens of minutes, which is common with larger integrated tooling. The sysadmins could work faster through the automation than they would have been able to handle manually.

4.5 OpenCHAMI Leverages Cloud-Init for Faster Node Personalization

Common in the world of the public cloud, `cloud-init` is relatively unknown to HPC sysadmins. It is a de facto standard with an open source client provided by the team at Canonical that is included with all major linux distributions. Each cloud implements their own metadata server to provide the client with information useful to boot a linux system from an undifferentiated system image. This process of instantiating a generic image into a specific running instance is often called "personalization". In HPC centers, ansible is often the tool of choice for personalization. It has powerful features to make changes to a running node based on discoverable information and information stored in an inventory. Cloud-init does far fewer things than ansible and it only runs as part of boot. This allows it to run much earlier in startup and to run much faster. Switching from ansible to cloud-init on a 650 node test system changed the boot time from ~8 minutes to ~5 minutes.

The sysadmins at LANL used the OpenCHAMI cloud-init metadata server to set configuration data and scripts for a set of nodes at once using logical groupings. All compute nodes received a standard configuration. Nodes with direct access to filesystems received an additional configuration, and nodes allocated for users to log in received a separate additional configuration.

Each group-based configuration could be updated independently via the metadata server API. The sysadmins could each focus on their own area of configuration without impacting other work happening in other areas of the cloud-init server. This allowed each sysadmin to work in parallel without first coordinating with all other changes.

4.6 External Firmware Management w/ Magellan

Bleeding edge hardware is common at LANL where scientists demand ever more from their science systems. When the new cluster arrived on site, not all devices had the same firmware revision, and we had reason to believe that newer firmware wasn't necessarily the best for our users. The sysadmins needed to assess the versions of firmware on all nodes and ensure commonality. As part of integration, the admins needed to run tests on various firmware versions, ensuring commonality again.

The OpenCHAMI team prioritized firmware management as a fully standalone module. The Magellan tool is a single binary that can integrate with microservices but can also be used alone. It is suitable for use with bash scripts and follows the redfish standard for managing the firmware of servers and the devices contained within those servers.

When integrating a new set of servers, a single sysadmin was able to focus solely on firmware while the rest of the team worked on other items. This parallelization of labor helped the whole team get things done faster. Once the sysadmin experiments with firmware clarified the behavior of the BMC in the new servers, the automation could be written to ensure commonality before and after testing.

5 CONCLUSION

Modern HPC systems are no longer well-served by manual, ad-hoc change management. This whitepaper illustrates how embracing configuration management, CI pipelines, and DevOps practices can radically accelerate both the velocity and reliability of change. What once took days or even weeks—manually staging artifacts, building images, rebooting nodes, and validating environments—can now be executed in a matter of hours through automated, version-controlled pipelines. This not only increases team efficiency but also ensures greater consistency, safety, and traceability of every change.

By codifying infrastructure and operational workflows, organizations can iterate faster, reduce human error, and deploy confidently across multiple environments. Whether using integrated platforms like CSM and HPCM or modular solutions like OpenCHAMI, the result is the same: less time navigating change, and more time delivering value. As demonstrated in real-world environments, automated pipelines transform deployment timelines from multi-day efforts into daily cycles—unlocking a new level of agility for even the most complex supercomputing environments.

Below is a summary of concepts and information above.

- DevOps culture in HPC promotes collaboration through codified, testable workflows that streamline operational handoffs.
- Version control underpins auditability, reproducibility, and coordination across large-scale infrastructure teams.
- Automation replaces manual steps with reliable, repeatable scripts and pipelines that eliminate configuration drift.
- CI pipelines reduce deployment and validation time from multiple days to just a few hours—multiplying change velocity.
- CSM provides built-in support for CI-driven change with integrated version control, artifact management, and orchestration.
- HPCM can be CI-enabled using external tools like Ansible and Git for structured, repeatable changes.
- Environment promotion provides a controlled pathway from development to production, minimizing release risk.
- OpenCHAMI delivers lightweight, modular automation ideal for rapid iteration and decentralized infrastructure management.
- When using OpenCHAMI, GitLab pipelines and tools like cloud-init and Magellan allow sysadmins to work in parallel and isolate responsibilities.

The net result is fewer delays, faster feedback, and higher confidence in every change—no matter the environment scale.

ACKNOWLEDGMENTS

ChatGPT was used, as a convenience, to help frame some of the best practice information. Most of what it generated was restructured, reworded, and filled in for detail. Grammarly was used to correct grammar and revise some sentences for enhanced brevity and clarity.

FURTHER READING

- HPE Performance Cluster Manager Upgrade Guide (S-9926-005)
- HPE Performance Cluster Manager Administration Guide (S-0120-007)
- Cray System Management (CSM) Documentation (https://github.com/Cray-HPE/docs-csm)
- OpenCHAMI Documentation (openchami.org)
- OpenCHAMI Cloud Init (https://github.com/OpenCHAMI/cloud-init)
- OpenCHAMI Image Builder (https://github.com/OpenCHAMI/image-builder)
- OpenCHAMI Firmware Management (https://github.com/OpenCHAMI/magellan)
- Buildah Container Images (https://buildah.io/)
- Cloud Init (https://cloud-init.io/)