There and Back Again: A Case Study of Configuration Management of HPC

Preston M. Smith
Purdue University
psmith@purdue.edu

Jason St. John
Purdue University
jstjohn@purdue.edu

Stephen Lien Harrell
Purdue University
sharrell@purdue.edu

ABSTRACT

Configuration management is a critical tool in the management of large groups of computer systems which are vital to deployment of High Performance Computing (HPC). In this paper, we describe the history, architecture, overarching goals, and outcomes of various configuration management systems utilized in support of HPC at Purdue University. Additionally, we enumerate best practices of configuration management that have been discovered in the strongly iterative HPC environment at Purdue.

KEYWORDS

HPC, System Administration, Large Deployment, Configuration Management, Case Study

ACM Reference format:
Preston M. Smith, Jason St. John, and Stephen Lien Harrell. 2017. There and Back Again: A Case Study of Configuration Management of HPC. In *Proceedings of HPCSYSPROS'17: HPC Systems Professionals Workshop, Denver, CO USA, November 12–17, 2017 (HPCSYSPROS'17),* 7 pages.
https://doi.org/10.1145/3155105.3155110

1 INTRODUCTION

Purdue University is somewhat unique in the academic computing space in that the HPC business model is built around the expectation that a new machine is built and installed every year. Each cluster deployed typically contains between 500 and 600 Intel Xeon nodes, an InfiniBand network, a parallel scratch filesystem, and sometimes accelerators, such as Xeon Phis or Nvidia GPUs. Each cluster is operated for 5 years and then is replaced with a new cluster. This regular turnover has been the catalyst for some unique decisions and time frames with configuration management for HPC at Purdue. Since the beginning of this business model, the administration staff that are responsible for these deployments have to balance rapid deployment of new technology with challenges from staff turnover. These requirements have made the ability to have a clean, consistent, and reproducible configuration management infrastructure crucial to the successful operation of HPC resources at Purdue.

HPCSYSPROS'17, November 12–17, 2017, Denver, CO USA

ACM ISBN 978-1-4503-5128-7/17/11…$15.00
https://doi.org/10.1145/3155105.3155110

1.1 Large Deployment Needs of HPC Clusters

In order to scale the administration of tens, hundreds, or thousands of machines without scaling staff proportionally, management tools are required to enable the system administrator to make changes to groups of systems at once while ensuring that the systems are configured according to a defined standard. In 1997, Finke [18] notes "As the number of Unix machines being supported grows, support groups develop or install more and more tools to automate and streamline the system installation and update process."

Couch [15] describes system configuration management as "the process of maintaining the function of computer networks as holistic entities, in alignment with some previously determined policy". According to Burgess, [12] "In system administration arising from the world of Unix, configuration management is used to refer to the setup, tuning and maintenance of data that affect the behavior of computer systems."

In high-performance cluster computing specifically, a variety of configuration management solutions exist to solve configuration management and deployment problems. These systems both address problems with management and deployment of groups of systems in general, but support the unique software and hardware in use on HPC clusters. ROCKS (2003) [26], OSCAR (2001) [24], SCE (2001) [31], and Systems Installation Suite (SIS - 2002) [17] are cluster management frameworks in use prior to Purdue's adoption of Linux clustering.

Today, many of these options are still available, in addition to Scyld, OpenHPC [27], Warewulf [3], xCAT [19], and commercial tools like Bright Computing [1].

1.2 Brief History of HPC at Purdue

Since the creation of the nation's first computer science department in the early 1960s, Purdue University has provided computing systems to support research. Over the years, the Purdue University Computing Center (PUCC) operated a CDC 6500 [7], a Cyber 205, and Intel Paragon, and the Engineering Computer Network created the first dual-processor support for the VAX 11/780 [20]. In the 1990s, PUCC ran a cluster of IBM RS/6000s and an IBM Scalable Parallel SP system.

Since the early 2000s, Purdue has operated Linux "Beowulf" [29] clusters built from commodity PC hardware. At first, Linux clusters were "recycled" clusters [6] built from former student lab PCs. This model was replaced by the creation of Purdue's Community Cluster Program [25], which supplements centrally funded staff and core infrastructure with faculty funds to purchase the clusters' nodes.

Today, the Community Cluster Program serves approximately 170 faculty partners from 36 departments, representing every academic college at Purdue.

2 MANAGING HPC CLUSTERS: ONE FRAMEWORK AT A TIME

2.1 Master Source (msrc) and Distrib

From 1995 through 2008, Purdue operated an IBM Scalable Parallel (SP2) system. Administration of the IBM SP2 system was performed with the Parallel System Support Programs (PSSP). PSSP is a "higher-level set of support programs and interfaces that enables you to take advantage of the powerful parallel processing features of the SP system." [4] PSSP allowed an administrator to administrate the system, its user accounts, storage, high-speed switch, and resource manager all from a central control workstation (CWS).

Also during the 1990s, Purdue operated an Intel Paragon XP/S Model 10 system, with 142 compute nodes [32], and a compute cluster of eight RS/6000 servers.

Tying all of these systems, and those serving the instructional computing needs of the university together was a suite of tools, developed by PUCC system administrator Kevin Braunsdorf, known collectively as Master Source (msrc) [11]. Msrc and related tools allowed Purdue UNIX and ECN administrators to, as Brausdorf states, practice DevOps "before Gene Kim made it popular" [2].

2.2 STACI

With the advent of Linux-based "Beowulf" clusters [29] at Purdue, the adoption of IBM SP management tools was not feasible, nor was the porting of msrc to the fledgling Linux platform. In 2002, Purdue systems staff evaluated OSCAR and ROCKS as options to support Linux clusters. In the end, systems staff developed an image-based management tool dubbed "STACI" - the Software Toolkit for Automated Cluster Imaging [28].

2.2.1 Goals. STACI set out to be fully automated, cluster and architecture-neutral, handle multiple systems, and not change software under actively running user jobs. The system also enforced separation to prevent multiple admins from interfering with each others' work.

2.2.2 Implementation. A STACI node image was stored in a chroot, with versioning, and relied on the PBS batch system for state. Only nodes not running jobs were re-imaged and rebooted, and nodes awaiting a new image were blocked from accepting any new work by being marked offline in the PBS batch system.

To upgrade an image, the system administrator would utilize a workflow like the following:

- Lock the image with "lock_img".
- Chroot into the image directory ("chroot debian-sarge-i386").
- Make changes as necessary. Edit files, install packages, etc.
- Unlock the image with "unlock_img", incrementing node image version.
- STACI node scanner and rolling upgrade processes begin to process image updates.

2.2.3 Outcomes. On the small (less than 100 nodes) clusters of the time, STACI worked well. It allowed for long-running jobs to run, trivial changes could easily be pushed to nodes in a job-coherent manner, and updates can quickly be pushed from the headnodes to idle nodes.

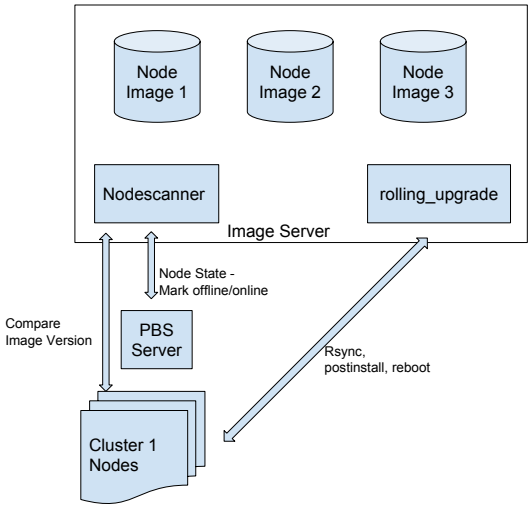

Figure 1: STACI Architecture

However, with scale came challenges: Updating images across an entire cluster would create load issues on the image server - often a cluster login node or PBS server. At the time, limited network bandwidth or disk I/O was often an issue.

With these known constraints, by 2005, when the Community Cluster Program began in earnest with 500+ node clusters, a new, more scalable management toolkit was required.

Although functionally, STACI was left behind, its legacy lives on at Purdue today and it is used as an example of how rolling upgrades of clusters can be done for a subset of critical updates. These type of rolling upgrades can be especially useful when a maintenance is not a viable option within the time-frame needed for an update, for example, in the case of kernel vulnerabilities.

2.3 CFEngine

As the admin team grew, new team members brought experience with new tools, and the Purdue team adopted CFEngine [13] as the successor to STACI.

2.3.1 Goals. By 2005, Purdue's cluster systems continued to grow in scale and in complexity, with new funding secured [5] for Purdue to join the NSF TeraGrid [14], and to operate a Tier-2 facility for the CMS experiment at the Large Hadron Collider at CERN [10]. 2005 also saw the deployment of "Lear", a 512-node cluster.

These projects required many additional services that preferred stateful nodes, and additional control was needed to manage the rapid pace of change to join CMS and the TeraGrid.

CFEngine was used as a configuration management framework, and defined rules to set machines to a known state. Configuration scripts were stored and managed in Subversion, and the administration team could document changes via diff messages mailed out with every commit.

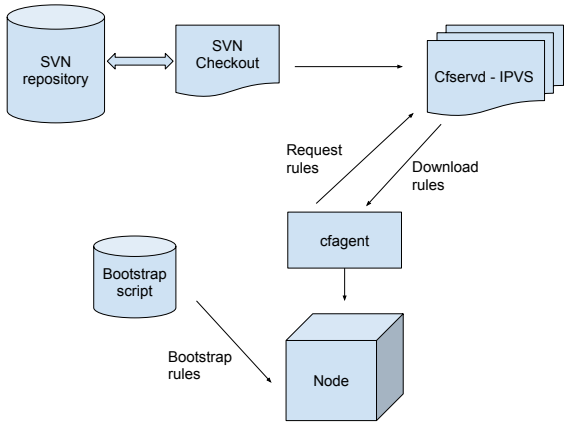

Figure 2: CFEngine Architecture

By May 2008, a challenge was laid out to deploy the "Steele" cluster in one day [30], and the admin team implemented a scalable deployment infrastructure using PXE, TFTP, Kickstart, and CFEngine [16].

2.3.2 Implementation. The automated infrastructure used to deploy Steele used PXE netboot, TFTP, and Kickstart to deploy a base Red Hat operating system. At the end of the kickstart, CFEngine and rules were downloaded from the deployment servers, CFEngine ran in "bootstrap" mode, and the node was rebooted, coming back as a fully-configured Steele node.

Before Steele, CFEngine was served by a single cfservd node, later replaced by a cluster of cfservd machines, made more scalable by IP virtual server *IPVS* [33]. Cfagent processes on nodes and cluster servers were run at boot and hourly via cron jobs.

CFEngine scripts would do all of the things that an admin would do by hand in the STACI workflow: download config files from SVN, edit other config files, start processes, install packages, run scripts, etc. The use of Subversion also allowed post-commit hooks to sanity-check files - for example, the syntax of the "sudoers" file.

To make a change on a cluster's nodes, the system administrator would utilize a workflow like the following:

- Update a local copy of the CFEngine rules repository from Subversion.
- Create appropriate CFEngine rules or add config files to execute the desired action.
- If a change was a test, protect the new rules by making them only executed if a "test" flag is defined.
- Manually test a change for syntax and correct behavior.
- Deploy change to a live cluster by checking chanages into the Subversion repository.
- Changes will propagate to all cluster nodes within an hour.

STACI also remained in some form, used as a helper script integrated with CFEngine to manage rolling reboots of cluster nodes, in the events of kernel upgrades, filesystem changes, etc.

2.3.3 Outcomes. A key characteristic of the CFEngine implementation was that it was completely deterministic. When a CFEngine run completed, the system would be in exactly the state that its policy specified. CFEngine would iterate through its rules as many times as necessary for the configuration to converge.

Another strong point of CFEngine was expressiveness. One team member described it as "The perl of configuration management." To borrow from the Perl community, "There was more than one way to do it.".

This is also a drawback, as administrator-designed CFEngine rulesets were inconsistent due to the environment's organic growth. For example, is there a guideline for when you check in a config file to SVN and distribute it versus just editing the one line you need to change in the file that the OS distributes? Another unanswered question during this time was "should the team write rules that are technology-based with code exceptions for specific systems, or make configuration rules system-based?". The resulting ruleset was difficult to read and therefore hard to change with confidence.

Additionally, unlike STACI, non-HPC infrastructure was managed in CFEngine. DNS, IPVS, HTTP servers, and even managed Linux desktops were integrated into the rulesets which ended up creating unintended consequences. Small tweaks to modules that were used by both the HPC and infrastructure portions of the ruleset could have far-reaching and unexpected outcomes.

2.4 Puppet

By 2012, with the deployment of the "Carter" community cluster, the organic nature of the CFEngine framework had made it a challenge for operating the HPC systems. CFEngine 2 was nearing it's end-of-life so a move to a supported management framework was undertaken. After some evaluation of options, including CFEngine 3 and Puppet, a decision was made to build a new management infrastructure based on Puppet [22].

2.4.1 Goals. The goal of the Puppet environment was a clean-room recreation of everything that makes a group of machines a community cluster. This effort had several key design decisions:

- The new environment imposed a sense of structure and process around the setup from the get-go, not built organically. This structure was set forth in a style guide that focused on readability of the puppet rulesets.
- Everything in the tree was a template, requiring only changes to variables. For example, rather than copying a fully-configured ldap.conf file to each machine that used it, Puppet used Hiera-based key-value lookups to fill in the name of the LDAP server into a template ldap.conf
- The Puppet design assumed no divergence. A cluster is a cluster and any divergence was handled within a specific template implementation.
- To impose structure missing from the CFEngine implementation, each component of the cluster had its own module, split by technology type, for instance, modules existed autofs, LDAP client, Torque batch system, SSH config, shell rc files, etc.
- Modules were organized by individual subsystem components such as a Puppet module for LDAP authentication, Torque resource management, and SSH instead of by cluster where all system configurations came from a single directory

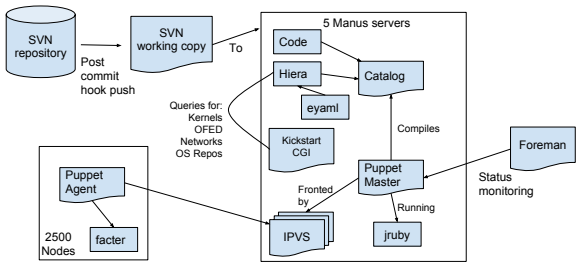

Figure 3: Puppet Architecture

2.4.2 Implementation. The Puppet environment was driven by a Subversion repository, which upon a commit, would push updates to a cluster of five servers, which served as the core of the Puppet infrastructure. These servers ran Puppet masters driven by jruby [9] for speed. These Puppet servers compiled catalogs based on the Puppet repository's Ruby code and Hiera keys. The Puppet masters were all load-balanced behind an IPVS virtual server.

Secrets such as passwords or private keys were held on the Puppet servers - first in GPG-encrypted files and later in encrypted YAML *eyaml* Hiera keys. [21]

Additionally, the Puppet masters also ran replicas of the site LDAP account database and a local copy of the clusters' DNS information.

Each of approximately 2,500 nodes ran a Puppet agent that would call in to the Puppet master servers over a 4-hour splay. A Foreman dashboard provided monitoring of Puppet-managed servers.

To make a change on a Puppet cluster's nodes, the system administrator would utilize a workflow like the following:

- Update a local copy of the Puppet repository from Subversion.
- Create or change a module's code, Hiera values, or template files to execute the desired action.
- If a change was a test, a new module was often created, and the test node's definition was changed to include that new module in place of the module being changed.
- Manually test a change for syntax and correct behavior.
- Deploy change to live cluster by checking changes into the Subversion repository.
- Changes will propagate to all cluster nodes within 4 hours.

2.4.3 Outcomes. Initially, the Puppet infrastructure and methods were a success. A more tightly controlled and less populated infrastructure allowed the systems staff to create a clean implementation of what had been a chaotic mish-mash of rules in CFEngine. However, after most of the transition was complete, some problems started to appear.

In the approximately five years of operation with the Puppet infrastructure, many challenges were encountered. As this was the first attempt to tightly control the configuration management infrastructure, some design choices were made that were non-optimal but were nonetheless adhered to. It is important to note this list is in hindsight. Many of these problems started appearing in the first or second year of implementation; however, they did not manifest themselves as severely until towards the end of the implementation.

- Changes required lots of re-running of Puppet, making it very slow to process changes across a cluster. For example, to execute a maintenance, admins often required 12-16 hours to apply patches. They would often be required to run Puppet multiple times and reboot cluster nodes several times.
- Related to re-running Puppet is the environment's scalability. The Puppet implementation required a non-trivial amount of computation on the Puppet masters, and as an effect, long maintenance days were exacerbated by having to manually throttle the number of machines being changed in a batch, so as to avoid burdening the infrastructure also serving other systems.
- The Puppet agents were noisy and resource-intensive on compute nodes. Running jobs that coincided with a Puppet run saw slowdowns ranging from 20% at 8 nodes to 45% at 64 nodes. In comparison, CFEngine has negligible impact up to 16 nodes. At 32 nodes, there's a 5% drop, and 15% at 64 nodes. With average job sizes at Purdue of just 6-8 nodes, CFEngine's impact was much more tolerable.
- The environment had no support for testing branches. Every change was made in production. Testing had to be done by making new modules and assigning those modules to specific nodes/server.
- The environment saw an explosion in the number of modules. By 2016, there were 194 unique modules in the Puppet tree. This not only affected complexity and maintainability, but also the scalability of Puppet servers and the length of agent runs on compute nodes.
- One of the problems with the maintainability of CFEngine started to occur with this system as well with one monolithic tree being responsible for multiple clusters as well as non-HPC infrastructure. The Puppet implementation was not intended to have general purpose (non-cluster) servers within its scope, but for lack of an option, they went in anyway.
- Problems with broken changes started to have wide-reaching consequences. Widespread outages due to typos or mistakes in the Puppet code were sometimes seen in modules that had little to do with the resource that was impaired.
- Despite the intent to not be divergent, divergence was unavoidable. For example, from cluster to cluster, we may see differences in networks from IP over InfiniBand to converged 10 Gb Ethernet to different OFED stacks. This divergence in a template-based environment created increasing complexity that was difficult to modify with deterministic results. The design goal of using templates and Hiera [23] keys exclusively led to several complex abstractions including network configurations or generation of Apache HTTPD config files from hiera keys. In the case of the network module, when a new cluster was built with a different network configuration than a previous one, the network module required overhaul, to avoid breaking existing clusters.
- Tightly coupling the kickstart infrastructure into Puppet created a dependency that sometimes resulted in a broken

installer. Students were typically responsible for OS deployment and often lacked the broad knowledge or access to change Puppet to allow a machine to be installed.
- Another paradigm that was used and was not successful was additive configuration generation. For instance, the original firewall module used rules from each separate technology template e.g. the httpd module would add a rule to open port 80. The firewall module would take all of the rules from all the templates and create a complete ruleset. Thus, it was difficult to understand the current state of the firewall or how it would be pushed to a specific machine.

2.5 xCAT and GoCD

By mid-2016, the Puppet environment was deemed to be too large and complex to maintain and be operationally credible. While most of the non-HPC infrastructure and older clusters still remain configuration managed with Puppet today, a new system using xCAT was devised to deploy and manage the annual HPC deployments.

It was decided that configurations would be static and a configuration management framework such as Puppet was discarded. In its place, a sparse root directory was checked into source control, and rsync was used to customize blank xCAT images.

2.5.1 Goals. The primary goal was proper continuous integration and continuous delivery (CICD) for testing of stateless system images provisioned via xCAT [19]. After unit testing, new job submissions would be rewritten dynamically using Torque's qsub submit filter to require the new system image for the job. Torque would then request that xCAT re-image the node (via rebooting the node). Once the node returned with the new image, the job would be started.

A secondary goal was limiting the scope of any specific change by changing deployment technologies as well as using new repositories. Although the goals were few, this concept was a large change from the previous vision: stateless nodes versus stateful nodes; and required continuous integration versus few testing options. This meant that much of the organizational knowledge about HPC deployments was suddenly obsolete.

2.5.2 Implementation. CICD was implemented via GoCD [8] running in Docker containers. The team moved from a self-hosted SVN repository server to GitHub. By using GitHub's web hooks, branches and pull requests for changes were linked between the source code system and GoCD. The on-premise Docker cluster hosted the central GoCD server as well as the necessary build agents for xCAT image building. A build agent was also run on the xCAT master, with dedicated hardware nodes for functional, real world testing inside of the Torque environment. This allowed GoCD to test new images and also to deploy completed images.

xCAT successfully deployed full system images containing all of our cluster services and software; however, the modification of Torque's submit filter to require new system images was never implemented as the amount of time to implement CICD was underestimated.

The intended workflow for how an administrator would make changes to the cluster image was as follows:

- Staff would update a local copy of the xCAT repository from the local GitHub instance
- A new Git branch would be created, and generates a pull request.
- GoCD would build a new xCAT image
- GoCD would execute unit tests.
- If the unit tests pass, then the cluster's xCAT image would be updated, and deployed to a set of test nodes set aside.
- The new node image would be pushed to the node via a Torque job submission with a job requirement.
- Torque would request that xCAT re-image the node
- The node would reboot into latest image and begins to accept jobs

2.5.3 Outcomes. Using separate code repositories for each deployment fixed not only the templating complexity of Puppet but also domain separation so that changing rulesets of different infrastructure and clusters would no longer cause unintended consequences. Each cluster was now effectively isolated from each other.

Moving to stateless image-based clusters was also a positive improvement. The "noise" of the very large Puppet catalog runs was no longer present and problems that could be recurrent with stateful nodes went away with a reboot.

CICD, however, did not work well; the process was very brittle and non-deterministic. If a CICD run did not pass successfully, running CICD again would often succeed. As the development of CICD progressed and used in production it became clear that there would have to be many more tests to be written for the many corner cases in the environment. This staff time was not accounted for in the original plan.

Making minor changes was a long, complicated, resource-intensive process (e.g. pull request with full CICD and image rebuild for changing one line in a text file) Subsequent pull requests depended on previous pull requests passing unit testing and being merged into master. These long processing and building times caused simple maintenances to be problematic. A simple typo that CICD did not catch could easily add 12 or more hours to a system maintenance.

The CICD infrastructure was originally functional but became impractical to operate. The complete pipeline originally took only an hour between a merge request being submitted to a completed image being produced. (This is the time between a change getting committed and a new image being available in the xCAT inventory.) After several months with new functionality getting bolted on in an attempt to fix process problems around frontend end image deployments and with changing enterprise infrastructure, the pipeline grew to be nearly seven hours long. This brought all configuration changes to a grinding halt as the systems administration team essentially only got one chance a day to make changes. At this same time, the lead developer behind the CICD infrastructure left the University.

The CPU and Disk usage told us the infrastructure had to be relocated from a few large virtual machines to dedicated hardware. As well, we identified the need to add additional stages to the CICD pipeline to speed up builds. The largest opportunity for time savings would be to have a stage which built a generic OS image which would be rebuilt only as necessary before feeding into the stages

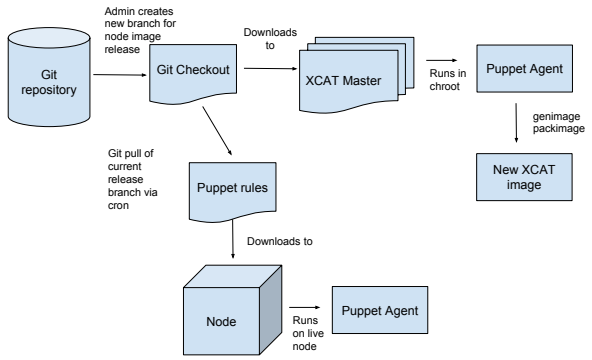

Figure 4: Hybrid xCAT/Puppet Architecture

which built specific compute or frontends images. The development of this stage would have required retooling all the build scripts due to assumptions made in the scripts about the image building process. However, all of these efforts looked untenable with the staff time available and the departure of the team's developer.

2.6 xCAT and Puppet Hybrid

The xCAT/GoCD configuration model was fraught with difficulties, particularly with making minor changes, so a minimal, masterless Puppet instance was added to node images, and the GoCD CICD process was removed.

2.6.1 Goals. One of the main goals of this design was to take the positive outcomes of stateless nodes and restore the ability to make small changes without having to generate a new image and reboot the cluster. Another goal for this deployment was to use existing knowledge both of technology and lessons learned over the years. The scope limitation of the xCAT/GoCD method, rolling reboots from the STACI method, and the technology stack (institutional knowledge) of Puppet were reused.

2.6.2 Implementation. In this system, xCAT PXE boots a full system image containing masterless Puppet. In the image creation Puppet rulesets install and configure cluster services according to node type (e.g. front-end, compute node). Once a node has been booted, Puppet continues to run and manage configurations on the same branch as the live system.

To make a change on a hybrid cluster's nodes, the system administrator would utilize a workflow like the following:

- Update a local copy of the Puppet repository from the local GitHub.
- Create Puppet code to execute the desired change on the cluster's nodes.
- Manually test the candidate code on a node
- When the changes are verified, the new Puppet code is checked into the Git repository in the master branch and, depending on urgency, the current deployment branch.
- Depending on the change one of three things could happen
  - If a change requires an emergency reboot, a new image is created from the master branch and a STACI-style rolling reboot upgrade occurs.
  - If it is a change that requires a reboot or is not an emergency, a cluster-wide downtime is scheduled to upgrade the nodes.
  - If the change is an emergency but does not require a reboot, the masterless Puppet executed via cron jobs on each node pulls from the Git repository's deployment branch and executes changes, in seconds. Changes will propagate to all cluster nodes within an hour. A bad configuration change in this scenario can impact running jobs so this is used as a last resort.
- If it is not already done, the admin then regenerates the cluster's master xCAT image for future nodes.

There are two branches maintained in Git: master and a live deployment branch. All changes go to the master branch, which feeds new image builds and testing that occurs before a maintenance window. To deploy a change (or usually a fix) to the cluster while it is running, commits from master must be cherry-picked to the deployment branch, making changes safe and easy.

Every month, a maintenance is scheduled, where we generate new system images and create a new live deployment branch based on master. Changes are applied directly without going through the GoCD CICD process.

The masterless Puppet instance on each node is designed to be as simple as possible. In nearly every case, Puppet simply installs a configuration file with no templating or variable substitution. The combination of not communicating with a Puppet master and simplified rulesets greatly reduced any job impact from Puppet runs.

2.6.3 Outcomes. Combining xCAT and Puppet gives us the benefit of stateless system images with the ease of management of Puppet. Our Puppet environment executes quickly and consistently, with very minimal impact on running jobs. Additionally, this system has allowed the spin up time for new systems administrators to go from months to weeks as it more closely resembles managing a single server or golden image. With the existing knowledge of Puppet, the STACI method of reboots, and minimized scope, system administrative time-to-solution has decreased and the ability to make small changes has been restored.

3 LESSONS LEARNED

The goal is to create a reproducible, consistent process for managing HPC clusters, so that we can quickly deploy a cluster each year and have systems administrators move on to more directly enabling our faculty's scientific discovery.

Our cluster management tools should be as simple as possible, so that our admin staff can focus on delivering value to Purdue's researchers, rather than on the meta-work of building automation systems. Over 13 years, and five management paradigms, we have reached a number of conclusions:

- There is no right solution—only the right solution for your environment and your business model at the current moment in time.
- Tight integration with many subsystems, such as kickstart, makes things convenient but can also complicate the situation when the tight integration breaks.

- At each step, consider how likely you are to be able to quickly onboard, train, and make productive a new admin of the average skill-set in your market. If your technology stack can only be managed by the person that created it, consider simplifying.
- Templating is a great tool, but it is better to have multiple rulesets doing similar things well with readability, rather than one built to meet all divergent cases that is so complex that it is impossible to enhance without unintended consequences.
- The less technically perfect solution that meets core requirements that is built with a technology that everyone understands is always the better option.
- As a corollary, the less technically perfect solution that is ubiquitous in the hiring marketplace is often the better option.

4 ACKNOWLEDGEMENTS

We'd like to thank all of our colleagues, current and past, for their contributions when creating these systems. Specifically, Mike Shuey, Scott Hicks, Jenett Tillotson, Andy Howard, Rick Irvine, Randy Herban, Spencer Julian, Graham McCullough, Alex Younts and Seth Cook. A special thanks to Paul Peltz and Andy Howard for giving feedback on this paper.

REFERENCES

[1] [n. d.]. Bright Computing | Advanced Linux Cluster Management Software. ([n. d.]). http://www.brightcomputing.com/
[2] [n. d.]. LinkedIn - Kevin S Braunsdorf. ([n. d.]). https://www.linkedin.com/in/ksbraunsdorf/
[3] [n. d.]. Warewulf. ([n. d.]). http://warewulf.lbl.gov/trac
[4] 1998. RS/6000 SP Overview. (1998). http://wwwuser.gwdg.de/~applsw/Parallelrechner/sp_documentation/pssp/admin/spa2mst09.html
[5] 2003. Extensible Terascale Facility (ETF): Indiana-Purdue Grid (IP-grid). (2003). https://www.nsf.gov/awardsearch/showAward?AWD_ID=0338627
[6] 2003. Sun donates $3.6 million for high-performance computer cluster. (2003). http://www.purdue.edu/uns/html4ever/031101.Bottum.Sun.html
[7] 2013. Raising the CDC 6500 from the dead: Paul Allen's engineers attempt toughest computer restoration yet. (2013). https://www.geekwire.com/2013/bringing-cdc-6500-paul-allen-ambitious-computer-restorations-attempted/
[8] 2017. Open Source Continuous Delivery and Automation Server | GoCD. (2017). https://www.gocd.org/
[9] 2017. The Ruby Programming Language on the JVM. (2017). http://jruby.org
[10] Kenneth Bloom. 2008. US CMS Tier-2 computing. In *Journal of Physics: Conference Series*, Vol. 119. IOP Publishing, 052004.
[11] Kevin Braunsdorf. [n. d.]. The pundits tool-chain from the NPC Guild. ([n. d.]). http://www.databits.net/~ksb/
[12] Aaron B. Brown, Joseph L. Hellerstein, and Alexander Keller. 2008. 1.3 - Automating System Administration: Landscape, Approaches and Costs. In *Handbook of Network and System Administration*, Jan Bergstra and Mark Burgess (Eds.). Elsevier, Amsterdam, 43 – 74. http://www.sciencedirect.com/science/article/pii/B9780444521989500057 DOI: 10.1016/B978-044452198-9.50005-7.
[13] Mark Burgess and Oslo College. 1995. Cfengine: a site configuration engine. *in USENIX Computing systems, Vol 8, No. 3* (1995), 309–337. https://www.usenix.org/legacy/publications/compsystems/1995/sum_burgess.pdf
[14] Charlie Catlett. 2002. The philosophy of TeraGrid: building an open, extensible, distributed TeraScale facility. In *Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on*. IEEE, 8–8.
[15] Alva L. Couch. 2008. 1.4 - System Configuration Management. In *Handbook of Network and System Administration*, Jan Bergstra and Mark Burgess (Eds.). Elsevier, Amsterdam, 75 – 133. http://www.sciencedirect.com/science/article/pii/B9780444521989500069 DOI: 10.1016/B978-044452198-9.50006-9.
[16] Donna Cumberland, Randy Herban, Rick Irvine, Michael Shuey, and Mathieu Luisier. 2008. *Rapid Parallel Systems Deployment: Techniques for Overnight Clustering*. Proceedings of the 22nd Large Installation System Administration Conference (LISA '08), USENIX Association, Berkeley, CA, USA. 49–57 pages.
[17] Sean Dague. 2002. System installation suite massive installation for linux. In *Ottawa Linux Symposium*. 93.
[18] Jon Finke. 1997. Automation of Site Configuration Management. In *Proceedings of the 11th Systems Administration Conference*. San Diego, CA, 155–168. https://www.usenix.org/legacy/publications/library/proceedings/lisa97/full_papers/18.finke/18_html/main.html
[19] Egan Ford, Brad Elkin, Scott Denham, Benjamin Khoo, Matt Bohnsack, Chris Turcksin, and Luis Ferreira. 2002. Building a Linux HPC Cluster with xCAT. *IBM Redbook* (2002).
[20] George H. Goble and Michael H. Marsh. 1982. A Dual Processor VAX 11/780. In *Proceedings of the 9th Annual Symposium on Computer Architecture (ISCA '82)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 291–298. http://dl.acm.org/citation.cfm?id=800048.801738
[21] Terri Haber. 2014. Encrypt Your Data Using Hiera-Eyaml. (21 October 2014). https://puppet.com/blog/encrypt-your-data-using-hiera-eyaml
[22] Luke Kanies. 2006. Puppet: Next-generation configuration management. *The USENIX Magazine* 31, 1 (2006), 19–25.
[23] Spencer Krum, William Van Hevelingen, Ben Kero, James Turnbull, and Jeffery McCune. 2013. Hiera: Separating Data from Code. In *Pro Puppet*. Springer, 263–294.
[24] T. G. Mattson. 2001. High performance computing at Intel: the OSCAR software solution stack for cluster computing. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*. 22–25. https://doi.org/10.1109/CCGRID.2001.923170
[25] Gerry McCartney, Thomas Hacker, and Baijin Yang. 2014. Empowering Faculty: A Campus Cyberinfrastructure Strategy for Research Communities. *Educause Review* (2014).
[26] Philip M. Papadopoulos, Mason J. Katz, and Greg Bruno. 2003. NPACI Rocks: tools and techniques for easily deploying manageable Linux clusters. *Concurrency and Computation: Practice and Experience* 15, 7-8 (2003), 707–725. https://doi.org/10.1002/cpe.722
[27] Karl W Schulz, C Reese Baird, David Brayford, Yiannis Georgiou, Gregory M Kurtzer, Derek Simmel, Thomas Sterling, Nirmala Sundararajan, and Eric Van Hensbergen. 2016. Cluster computing with OpenHPC. (2016).
[28] Michael Shuey. 2004. Automatic Software Updates on Heterogeneous Clusters with STACI. (May 2004). http://www.linuxclustersinstitute.org/conferences/archive/2004/PDF/presentations/Shuey.pdf
[29] Thomas L. Sterling, John Salmon, Donald J. Becker, and Daniel F. Savarese. 1999. *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters*. MIT Press, Cambridge, MA, USA.
[30] Steve Tally. 2008. Purdue supercomputer unboxed and built by lunchtime. (5 May 2008). https://news.uns.purdue.edu/x/2008a/080505McCartneyBuild.html
[31] Putchong Uthayopas, Sugree Phatanapherom, Thara Angskun, and Somsak Sriprayoonsakul. 2001. Sce: A fully integrated software tool for beowulf cluster system. In *Proceedings of Linux Clusters: the HPC Revolution*. National Center for Supercomputing Applications (NCSA), University of Illinois, 25–27.
[32] William Whitson. [n. d.]. Getting Started on the Paragon XP/S Supercomputer. ([n. d.]). https://web.archive.org/web/19980127055751/http://www-rcd.cc.purdue.edu:80/Paragon/quick-start.html
[33] Wensong Zhang and Wenzhuo Zhang. 2003. Linux virtual server clusters. *Linux Magazine* 5, 11 (2003).