Managing the SMW as a git Branch

Douglas M. Jacobsen[1], Randy Kleinman[2] and Harold Longley[2]

Abstract— **Modern software engineering/DevOps techniques applied to Cray XC Series systems management enable higher fidelity translation from test to production environments, reduce administration costs by avoiding duplicate efforts, and increase reliability. This can be done by using a git repository to track and manage all system configurations on the SMW(s), and then adapt a gitflow-like development methodology.**

Using git improves change management with peer review of changes, identification of who made a change, knowing why a change was made, facilitating easier reversion of bad changes, and also enables a workflow using the ideas from gitflow with individual branches for preparing, maintaining, and recording releases. Using branches enables multiple contributors to work on different types of change without getting unexpected changes from another feature area.

We will describe a process to extract/abstract configuration data, manage it using git, and then re-apply changes to the SMW(s), similar to NERSC (National Energy Research Scientific Computing Center).

I. INTRODUCTION

The increasing complexity of High Performance Computing systems combined with rapidly evolving user requirements are creating new challenges in systems management techniques to ensure that new features and bug corrections can be integrated into the system without introducing unnecessary regressions or downtime. The recent availability of new software engineering tools and DevOps techniques and the application of these tools to the field of system engineering provide some solutions for solving these issues.

The Cray Linux Environment (CLE) and the Configuration Management Framework (CMF) ecosystem introduced with CLE 6.0 provide the capability to describe the entirety of the system configuration in machine parseable configuration files and standard RPM software packages. Because of this it is trivial to use Software Configuration Management (SCM) tools like svn or git to manage these documents and track them over time. Used effectively, the SCM tool can then provide a given set of configurations all the benefits that SCM tools have always provided: history and provenance of changes, collaboration from multiple contributors, and centralized management of data. The challenge then is not deciding to use these tools, it is determining a way to use these tools that derives the greatest benefit.

NERSC's goals in implementing a new source control system on top of Cray's CLE/CMF configuration system were to:

- provide a mechanism for NERSC Systems Engineers and Cray On-site Engineers to directly collaborate on the configuration of the system(s)
- centralize and unify the configuration of the four Cray XC Series systems operated at NERSC (two production and two Test and Development systems)
- perform all development and testing of configurations and system software capabilities on the Test systems and have a reliable mechanism for deploying those same configurations and capabilities to the Production systems
- capability to use test systems for speculative development while retaining ability to return to production configuration
- control and track all pertinent configurations external to the SMW to ensure we can rebuild the system rapidly in the case of loss of the SMW

The critical needed capability is to be able to reliably move an SMW from one fully known state to another – and back again (if desired). This is a particularly powerful concept when there is a test system and a production system available. The test system is configured and managed using the SCM tools and repositories to create and plan future work. Then using the self-same SCM repositories to configure and manage the production system, the production system can be put into the identical state. This creates a rational testing and development capability required to meet the objectives above.

Git[1] was selected as our SCM tool of choice both because of its technical attributes but also because it enables a highly-collaborative development-focused workflow. Git is a highly distributed configuration management tool, wherein each copy of a repository is (usually) self-sufficient in all regards. In git terminology a change to the repository is called a "commit" while the repository is a collection of commits. Each branch is a specific arrangement of commits through time. A new branch is created from some commit reference of its parent branch. Merging two branches results in a "merge commit" on the target branch which is used to

This manuscript has been authored by an author at Lawrence Berkeley National Laboratory under Contract No. DE-AC02-05CH11231 with the U.S. Department of Energy. The U.S. Government retains, and the publisher, by accepting the article for publication, acknowledges, that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for U.S. Government purposes.

This paper has been accepted and will be published as an article in a special issue of Concurrency and Computation Practice and Experience on the Cray User Group 2018.

[1]D. M. Jacobsen is a Computer Systems Engineer at the National Energy Research Scientific Computing Center (NERSC), Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, California, U.S.A. `dmjacobsen@lbl.gov`

[2]R. Kleinman and H. Longley are Software Engineers at Cray Inc., 901 Fifth Avenue, Suite 1000, Seattle, Washington, U.S.A. `rkleinman@cray.com`, `htg@cray.com`

resolve any conflicts between the branches and create a reference indicating that all the changes in the merged commits have been resolved. Git is particularly adept at branching and managing multiple concurrent lines of development, which is a key feature in the context of this work. Finally, a number of tools [2], [3], [4], that provide social software management and allow policies to be enforced on specific branches or patterns of branch names, provide peer review tools, and enable higher-order branching complexities (such as forking), have emerged which allows a site to enforce an exquisite level of control in how and when certain types of operations can be performed and by whom. The term "gitflow"[5] refers to a particular workflow that organized git branching can enable. However, in this paper, we refer to "gitflow" in a more general sense, which is the workflow that a particular site adopts based on their needs and what git and git-management tools enable.

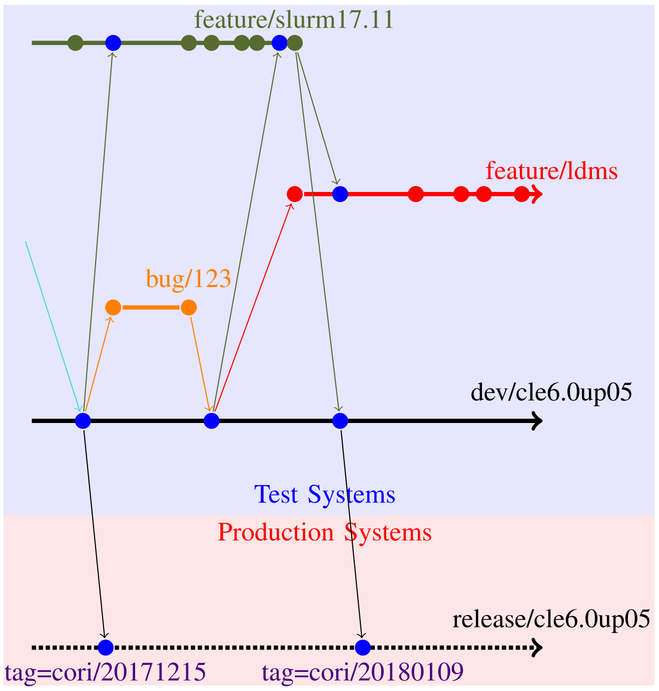

Fig. 1. SMW-gitflow. The horizontal lines represent branches of a git repository. The circles represent commits, non-blue are normal development commits, blue are merge commits. Lines between branches illustrate merge directionality.

As shown in figure 1, an example SMW-gitflow allows parallel development of new features and bug fixes, while all contributing, over time, to some specific mainline development branch. The mainline development branch never gets any direct updates, these are all done in specialized feature or bug development branches. All the development work is performed on the test and development system. In the most conservative development model, all the configurations on the SMW can be replaced to match the contents of a specific git branch. From there system configuration and boot artifacts can be created, and then the test SMW configurations can be restored to another git branch. This allows a developer to try forward-looking or risky work that can move the configuration far afield from the mainline development without any real risk to the system or its role as a model for the production machine.

Once the development branch has reached the needed feature and testing milestones the development branch can be merged into the release branch. The release branch *only* ever gets updates from the development branch, so there is surety that any testing performed on the development branch is predictive of the release branch. The separation of the two allows some level of chaos and reversions to occur in development that is not exposed to release. The production systems only use the release branches for configuring the SMW. In this model the production system is simply an endpoint while the test and development system gets all the care and focus of the system developers.

II. METHODS

The basic concept is that SMW configurations, in particular the source material needed to produce configuration sets, boot images, and configurations required to boot and manage the system are stored in git. None of the Cray-provided system management software is changed or modified to integrate these, however some additional scripts are used, with a front-end called "smwflow"[6], to ease the integration between the git repositories and the SMW.

The methods in this paper are primarily focused on the NERSC method for merging and abstracting multiple system configurations into a single abstracted configuration repository. It should be noted that Cray does provide a mechanism starting with CLE6.0UP04 to prepare config sets for export or re-import sets to and from an SCM repository using the `cfgset_export_perms.py` and `cfgset_restore_perms.py` scripts found in `/opt/cray/imps/default/etc`[7].

A. Managed Configurations

The SMW has a number of different types of configurations that can be considered for git-based management. The primary focus for NERSC up to this point has been storing and managing:

- all the configurations required to generate the global and CLE config sets [8], these include:
  - site-local Ansible plays and roles [7], [9]
  - config.yaml documents *not* generated by the cfgset tool (i.e., non-Cray config.yaml documents)
  - dist files (primarily non-Cray netroot preload files)
  - files/roles/SimpleSync data
  - configuration worksheets
- the RPMs in the site-local zypper repos (using git-lfs [10])
- site-local recipe definitions
- site-local package collections
- xtbootsys boot automation scripts

Other important items we are working to get integrated:

- Hardware Supervisory System (HSS) configuration files stored in `/opt/cray/hss/default/etc` which are described in [11], [12], [13], and [14] and enumerated here:
  - blade_json.sedc
  - bm.ini
  - bootsys.ini
  - cab_json.sedc
  - sm.ini
  - xtbounce.ini
  - xtcli.ini
  - xtdiscover.ini
  - xtnlrd.ini
  - xtpcimon.ini
  - xtpmd.ini
  - xtpmd_plugins.ini
  - xtpowerd.ini
  - xtremoted/xtremoted.ini
- /etc/opt/cray/bootlog_profiler/bootprofiler.ini
- /etc/opt/cray/capmc/
- /etc/opt/cray/dumpsys/
- /etc/opt/cray/edumpsys/ *NEW in up06*
- /etc/opt/cray/esd/esd.ini *NEW in up06*
- /etc/opt/cray/llm/
- /etc/opt/cray/modules/Base-opts.default.local
- /var/opt/cray/certificate_authority/
- /etc/nginx/conf.d/cray/xtremoted.conf
- /etc/nginx/conf.d/cray/esd.conf *NEW in up06*
- Other SMW localizations (e.g., LDAP/PAM configuration, site administration scripts, etc)

B. Organization of Git Repositories

NERSC has organized configurations into three different git repositories: `nersc-cle6` for most of the "shareable" text configuration items; `nersc-zypper`, a git-lfs[10]-enabled repository for storing RPMs; and `imps-secured`, which is a SMW-private repository storing all the root-equivalent credentials. The primary reason for separating `nersc-cle6` and `nersc-zypper` is to ease git cloning and time/effort since more staff are focused on the configurations rather than the binary content. The reason for separating `nersc-cle6` and `imps-secured` is to provide a security separation for particularly sensitive information.

The layout of the `nersc-cle6` repo is shown in table I. The layout of the `imps-secured` repo is similar except it does not place files in the `imps` subdirectory.

The `nersc-zypper` repository has a very simple layout with one directory per managed zypper repository. Within each there is an "RPMS" directory and an "SRPMS" directory. The content of the "RPMS" directory are used to populate the target zypper repository, whereas the source RPMs stored in the "SRPMS" directory are used both to aid in determining provenance of the binary RPMS as well as providing needed source materials to rebuild in the future if necessary. The "smwflow" script is able to selectively install RPMS into the zypper repository to allow for machine-specific RPMS if necessary. Owing to the extreme inefficiency of git in managing binary content, we use git-lfs [10] to manage all RPM content. This allows `nersc-zypper` to participate in the branching scheme, thus enabling polymorphic transformations of the zypper repositories across systems, without incurring extreme download or filesystem performance issues when changing branches or cloning.

C. Configuration Set Construction

As can be seen in table I most of the content of the `nersc-cle6` repository is dedicated to populating and constructing the CLE and global configuration sets. It is important to note that any configuration *sources* are stored in the git repository, not content that can be auto-generated by the Cray-provided CMF, such as the `dist/compute-preload.cray` file, `files/roles/common/etc/hosts`, and almost all `config/cray_*.yaml` files. By focusing the git repository on the storage and management of the source files, rather than attempting to transfer data backwards from the configset unnecessarily into git, the error rate is reduced as well as fewer opportunities for issues to arise as newer versions of CMF are introduced that may handle generated content differently than previous versions.

To construct a new configuration set the following steps are used by the "smwflow" tool:

1) Create a new empty config set. This will initialize the config set and produce initial versions of all autogenerated content. For example:
```
smw# cfgset create --no-scripts \
     --mode=prepare --type=<type> <name>
```
2) Setup worksheets. To maximize the amount of shared, non-redundant content, the "smwflow" tool selects and builds worksheets from a variety of sources in the git repositories. This process is covered in detail in section II-D. Once the worksheets are prepared the config set is updated with the content of the worksheets. For example:
```
smw# cfgset update --no-scripts \
     --mode=prepare \
     -w '/path/to/worksheets/*.yaml' \
     <name>
```
3) Build config/*yaml files using sames rules as described for worksheets in section II-D. No cfgset operations performed.
4) Copy files/* (SimpleSync/Roles) content to config set using same selection rules used for worksheets. The permissions.dat file is used to set custom ownership/permissions in the config set where required. No cfgset operations performed.
5) Copy dist/* content to config set using same selection rules used for worksheets. No cfgset operations performed.
6) Copy ansible/* content to config set using same selection rules used for worksheets. Content is destructively rsync'd in place (i.e., system-specific content may replace generic content). Recommend using system-specific variables for differentiation, not system-specific ansible plays/roles. No cfgset operations performed.
7) If global config set, set up the NIMS maps.

TABLE I
NERSC-CLE6 GIT REPOSITORY

Configuration	Description
imps/cle_ansible	Ansible plays and roles for CLE config set
imps/cle_config	config YAML files to directly inject into cfgset/config
imps/cle_dist	site-local files to populate cfgset/dist, notably local netroot preload definitions
imps/cle_files	roles and SimpleSync files to populate cfgset/files
imps/cle_files/permissions.dat	non-standard owner/group/permissions
imps/cle_worksheets	Jinja2 templated worksheets for CLE config sets
imps/cle_worksheets_vars	variable definitions for templated CLE worksheets
imps/<system>_cle_files	roles/SimpleSync specific to the named system
imps/<system>_cle_files/permissions.dat	non-standard owner/group/permissions for the named system
imps/<system>_cle_worksheets	entire cle worksheets specific to the named system
imps/global_config	YAML files to directly inject into global cfgset/config
imps/global_worksheets	Jinja2 templated worksheets for global config sets
imps/global_worksheets_vars	variable definitions for templated global worksheets
imps/<system>_global_config	system-specific YAML files to directly inject into global cfgset/config
imps/<system>_global_worksheets	other system-specific global worksheets
imps/image_recipes.local.json	local image recipe definitions
imps/package_collections.local.json	local package collection definitions

8) Inject config set construction metadata into `cfgset/config/nersc_cfgset.yaml`. Metadata includes the location, branches, and commits of the `nersc-cle6` and `imps-secured` repositories.
9) Perform final config set update, allowing postscripts to run. This is required to produce a bootable config set that has needed platform files (global) or `/etc/hosts` (cle), for example. Runs `cfgset update --mode=prepare <name>`. It is important that `xtalive` have a positive response and all nodes are enabled during this step.
10) Perform config set validation routines. This runs `cfgset validate <name>`, as well as parsing both the input worksheets and the config-set-produced worksheets and comparing all keys and values to validate that the worksheets were correctly processed.
11) Any final site-local preparation steps needed. At NERSC this is used to build the ssh configuration and inject those files into cfgset/files/* in order to automatically construct ssh_known_hosts (and so on) using knowledge of all systems (by parsing all cray_net_worksheets under management).

It is important to emphasize the role of worksheets and the configurator at this point. Worksheets are provided as the primary vehicle of configuration input because we have found those to be relatively stable over several versions of CLE 6.0. The key feature however is that the configurator reads worksheets in as inputs and produces config files directly from them. It then proceeds to use the config files to produce the worksheets found in the config set. By comparing the git-managed input worksheets to the configurator output worksheets and ensuring that the data within is identical, we can be certain that the configurator has correctly interpreted the git input data. If the git-managed input worksheets differ from the output worksheets, config set construction fails and produces an error.

The final "cfgset update" operation performed intentionally performs no worksheet operations to ensure that all the needed configurations are correctly added to the config/*yaml files, even if those are not directly represented in the worksheets. All "cfgset" operations in the construction process use "mode=prepare" to ensure non-interactive configurator behavior. This is necessary owing to the requirement that all boot artifacts can be generated with no human intervention.

D. Configset Object Selection and Construction for Multiple Systems

To support multiple systems within the same git repository two different but related strategies are used. First, for each object type of interest (e.g., ansible, config, dist, files, worksheets) a number of paths are searched to find the most appropriate values. These paths range from a common path (e.g., cle_worksheets) to system-specific (e.g., cori_cle_worksheets). The second strategy for supporting multiple systems is to use Jinja2 templating [15] inject system-specific variables into worksheets stored in the common area.

When selecting which objects to use to populate the config set a number of paths are searched by combining the object type (e.g., "worksheet"), config set type, system name, and repository. From lowest to highest priority:

1) Shared content,
`nersc-cle6/imps/<type>_<object>`, e.g.,
`nersc-cle6/imps/cle_worksheets`
2) System generation content,
`nersc-cle6/imps/<gen>_<type>_<object>`
e.g., *`nersc-cle6/imps/n8_cle_worksheets`*
3) System-specific content,
`nersc-cle6/imps/<system>_<type>_<object>`,
e.g., *`nersc-cle6/imps/cori_cle_worksheets`*
4) System-specific secured content,
`imps-secured/<type>_<object>`, e.g.,
`imps-secured/cle_worksheets`

Ideally, as much content as possible should be in the shared directory. The cray_net_worksheet.yaml and cray_node_groups_worksheet.yaml files will always be in the system-specific directory.

To drive as much content as possible into the `nersc-cle6` repository, and as much as possible into the common directory, some of the almost-identical worksheets were converted to Jinja2 templates, reading system-specific variables from `nersc-cle6/imps/<type>_worksheets_vars/<system>[_secrets].yaml` or from similar paths in `imps-secured`. The "secrets" YAML files are encrypted with the Ansible Vault, using SMW-specific keys. The Ansible Vault [16] provides AES-256 encryption of sensitive data using a pre-shared key. Each SMW (and thus system) has a separate Ansible Vault key, which is used to secure data stored in git and within the config set to only allow secrets to be accessed on the SMW or as root on the system. Only worksheets support Jinja2 templates - all other content is simply discovered using the path hierarchy described above.

E. *Gitflow / Branching Strategy*

As shown in figure 1, all development is performed in feature and bugfix branches. These feature and bugfix branches are merged into the primary development branches via Pull Requests in NERSC's internal BitBucket. This affords us the opportunity to perform code reviews, discuss testing already performed on the proposed changes, and deployment strategies prior to merging the code into the mainline. Once the code is considered stable and ready for use on production systems, the changes are merged into the release branch, again via Pull Request and requiring review. The changes when going from development to release is typically a roll-up of many commits, potentially resulting in dozens or hundreds of changes, and so the review is to ensure that all features planned for upcoming release are included and as a final verification.

We typically use the development branches as the mainline target for our test systems. However, as has been discussed in this document, the test system SMWs are frequently put onto different branches in support of development activities. We can even load the release branch onto the test system SMW if desired.

Multiple development and release branches can be maintained simultaneously. These support updates from one version of CLE to the next. Thus during a time period of transition from cle6.0up04 to cle6.0up05, we would support branches *develop/cle6.0up04*, *develop/cle6.0up05*, *release/cle6.0up04*, and *release/cle6.0up05*. Any features that are merged into one of the develop branches needs to be pulled into all the other supported develop branches. This can either be done by the author of the feature/bugfix changes by performing multiple Pull Requests into each of, for example, *develop/cle6.0up04* and *develop/cle6.0up05*, or the maintainers of the develop branches can merge changes directly from one to another, depending on the need and the level of difficulty of the merge (typically no difficulty). Similar to before, changes to the release branches are done by merging the matching develop to release branches only.

When booting the production systems (or initiating a rolling update) we tag the HEAD of the release branch used to boot the system. This eases finding critical release commits over time, and makes it very easy to determine when a particular feature actually went into production.

F. *Getting Started from Your Current SMW(s)*

Starting from scratch takes some effort to properly bring in all the configurations, and, if multiple systems are to be managed, to properly separate out the common from system-specific content. To start, one first needs to initialize their own version of the `nersc-cle6` and `nersc-zypper` repositories. Using your test system SMW as the reference, copy the local `package_collections.local.json` and `image_recipes.local.json` to the config git repository. Next copy all of the global worksheets to `global_worksheets`, and cle worksheets to `cle_worksheets`. Copy the cray_image_groups_config.yaml to `cle_config`. Copy the CLE config set ansible directory to `cle_ansible` in the git repo. Finally, start examining the CLE config set files directory, determining which content needs to be stored in git (e.g., Lustre FGR settings, RSIP configurations, munge keys, etc).

At this point attempt to generate a new config set and boot the test system with it. Assuming all works as expected, you'll need to start adding the production system. The initial setup of this is a significant amount of work to identify exactly where the systems diverge, and minimize those as much as possible.

To keep things simple, NERSC uses the identical image recipes, package collections, and site-local Ansible plays on all systems. Image recipes and package collections are not differentiated by system in any way, at all. For ansible, many of our plays look to include variable files named `{{nersc.machineName}}.yaml` or `{{nersc.machineName}}_secrets.yaml` (ansible-vault) systems. The plays and roles are then modified to use those. The "nersc.machineName" is defined by adding a machine-specific config YAML file to the CLE config set.

To determine which worksheets can be shared and which must be made machine-specific, you can compare two worksheets using `smwflow diff_local_worksheets file1.yaml file2.yaml`. In this way you can figure out the best strategy for sharing as many of the worksheets as possible, turning some into static shared content, others into templates, and fully specializing some worksheets.

G. *Updating Configurations in git*

Updating the git configurations is mostly trivial, however it is important that real administrator user accounts perform the git operations, not root or crayadm, as those obscure which real user made some specific changes.

H. Updating SMW Configurations / Changing the SMW Branch

The SMW copies of the git repositories are root owned clones of `nersc-cle6` and `nersc-zypper` that are kept in `smw:/var/opt/cray/disk/1/software/git`. These clones have a default "origin" remote of the repo stored on the NERSC private BitBucket server using the root ssh key to get read-only (pull-only) access.

To change the git branch that is being used on the smw, one needs to first update the SMW copies, for example, here is the manual way:

```
smw# cd /var/opt/cray/disk/1/software/git
smw# cd nersc-cle6
smw# git pull
smw# git checkout feature/slurm-17.11
smw# cd ../nersc-zypper
smw# git pull
smw# git checkout feature/slurm-17.11
```

Manually moving all repositories to the correct branch can be finicky and potentially generate mistakes. Instead of manually running all the git commands directly on the SMW root-owned copies of the git repositories, one can automatically update the repos, check out the appropriate branch and potentially perform additional validation steps using the smwflow command:

```
smw# smwflow checkout <branch>
```

"smwflow" is the software middle-ware prepared by NERSC for the purpose of managing the SMW-gitflow process on the SMW.

Next the configurations stored in those repositories need to be applied to the SMW. We use scripts which require no human intervention to actually do this. This allows either a person to run the scripts manually to perform a custom set of changes or for the commands to be executed by an automation agent for standard rebuild and regression tests.

The first step is to replace the SMW-wide configurations required for building images the git repositories by running:

```
smw# smwflow update image_inputs
```

The image inputs update replaces:

1) local package definitions
2) local recipe definitions
3) local zypper repos

No Cray-provided recipe definitions, package collection definitions, or zypper repositories are managed by git or smwflow. Those are exclusively managed by the SMW software installation and normal Cray patching methodologies.

The next step is to update the global config set and construct a new CLE config set. It is possible to rebuild the global config set from scratch using this process, but the NIMS maps will be destroyed in the process (which may be OK with you). Before attempting either global or CLE config set construction/updates ensure that `xtalive` is fully responsive and that no nodes in the system are disabled (check `xtshow_disabled` for any unexpected output. Failure to do this can result in unbootable nodes later on that don't make it into the global platform definition files. This is because `xthwinv` is called as part of the cfgset update process.

To verify the global config set matches the current content of git:

```
smw# smwflow verify_cfgset --type global global
```

To update the global config set:

```
smw# smwflow update_cfgset --type global global
```

To create a new config set:

```
smw# smwflow create_cfgset --type global global
```

If a new global config set is constructed, it will be necessary to build a new p0 NIMS map. The NERSC-style map (orthogonal NIMS groups of admin, service, compute, login) can be built using:

```
smw# smwflow create_nimsmap p0
```

The map creation process relies on all tier2 and login (or other re-purposed compute nodes) already being re-purposed, and the "tier2_nodes" and "login_nodes" node groups being populated in the `<system>_cle_worksheets` portion of the `nersc-cle6` git repo.

After performing the image_input updates and global config set updates one can then run the `image create ...` or `imgbuilder ...` commands as required to create images. Warning: in many live update situations you'll want to avoid automatically mapping the new images, to ensure that the new image(s) is/are only put in the critical path after pushing the image root to the boot server in the case of netboot images, or other more detailed cases discussed later in section III-B.5.

In parallel with the image builds, the CLE config set can be constructed. It is *possible* to update CLE config sets, but preferred to create new ones in NERSC SMW-gitflow scheme. This reason for this is to reduce confusion in the case of rolling updates, wherein the system may be booted with a diversity of images and configuration set combinations. If new images and new CLE config sets are always constructed anew from known commit-level/branches in git, it becomes possible to map the system changes as a series of state transitions, which eases debugging and communication of change.

To build a new CLE config set:

```
smw# smwflow create_cfgset --type cle \
     <system>.<branch>.$(date +%Y%m%d%H%M%S)
```

At this point all of the currently tracked and managed configurations by NERSC are updated. Future additions of the HSS configuration changes may require the system to be shut down to safely replace them. Changes to the global config set ansible plays may require cray-ansible to be re-run on the SMW, or possibly for the SMW to be rebooted to get fully configured.

This section has discussed in some detail what configurations are updated and how. This however, usually is a simple set of steps:

```
smw# smwflow checkout release/cle6.0up05
smw# smwflow update image_inputs
smw# smwflow verify_cfgset --type global global
```

```
smw# smwflow create_cfgset --type cle <name>
```

I. Performing CLE Software Patch and Site-Local Updates on the Production System(s)

A basic system software update from a minor CLE or SMW patch or adding some site-local RPMs or modifying Ansible or other configurations is greatly simplified in the SMW-gitflow scheme. This is owing to the fact that the configuration and patches are tested on the test system much earlier than they get to the production system. Any changes to system-specific content for the patches in git, need to be done for all system definitions before the changes can be merged to the development branch. This ensures that the development branch is system-agnostic, and that any ordering of system boots can be supported.

The day before the planned maintenance day the development branch is merged into the release branch. Next the CLE/SMW patches can be installed on the production SMW and the site-local zypper repos are regenerated from the updated release branch from `nersc-zypper`. Finally images can be built but not mapped.

After the system has been shut down during the planned maintenance, then the images can be mapped (using saved output from `imgbuilder -s --map --dry-run`. The global and CLE config sets need to be constructed or updated, and then mapped into the NIMS map.

Finally boot the system with full confidence that everything will work as it did on the test machine, since the configurations are precisely the same as was created and tested in the test environment.

J. Performing CLE Software System Updates

When performing a system update, or fresh re-installation, you can use the existing git repositories to greatly speed the update/installation process. This is accomplished by doing all the hard work on the test system(s) in advance of the production system(s) (as usual), by preparing all the worksheet specializations and modified image recipes on the test system. By doing this most of the steps associated with preparing the global and CLE config sets can be skipped almost entirely for the production system(s).

The basic process on the test SMW looks like:

1) Perform SMW Installation in a snapshot
2) Construct a new global config set using the “smwflow” tool. Iterate with git until global config set converges (i.e., validates with “smwflow” with no issues). You’ll need to modify the template version information (ignore the DO NOT MODIFY warnings).
3) Reboot SMW into update snapshot
4) Perform any needed xtdiscovery/rtr steps needed to (re)initialize the SMW
5) Perform any needed xtzap steps to get the HSS system working properly.
6) Modify local recipes (usually just changing SLES and Cray ”up” level indicators in the recipe names. Iterate on building images until the formatting of the recipes and package collections stabilizes.
7) Construct a CLE config set using the “smwflow” tool. Iterate with git until CLE config set converges (i.e., validates with “smwflow” with no issues). You’ll need to modify the template version information (ignore the DO NOT MODIFY warnings).
8) Review all the config set documentation in the install or upgrade section of [11] making any recommended changes.
9) Modify all system-specific worksheets to work with the update.
10) Map the images
11) Boot the system

The basic process on a production system is very similar, except that there should be no need to iterate for convergence. As shown in figure 2, the git branching strategy for managing a system update is uses one development and one release branch per system software version. For example, the TDS system(s) would use branch dev/cle6.0up05 for SMW software installations for cle6.0up05, and branch dev/cle6.0up06 for SMW software installations of cle6.0up06. Similarly the production systems use release/cle6.0up05 and release/cle6.0up06 for SMW software installations of cle6.0up05 and cle6.0up06 respectively. This is done because the git configuration repositories are tuned to configure the relevant software versions, which may imply needed differences in a minority of the configurations (usually). In general commits on feature or bugfix branches can can be merged into either branch (or cherry-picked depending on the context) so long as the developer is careful to merge frequently. As usual the release branches should only get merged from the development branches to ensure that all changes are fully tested and that no commits (other than merge commits) exist solely on the release branches.

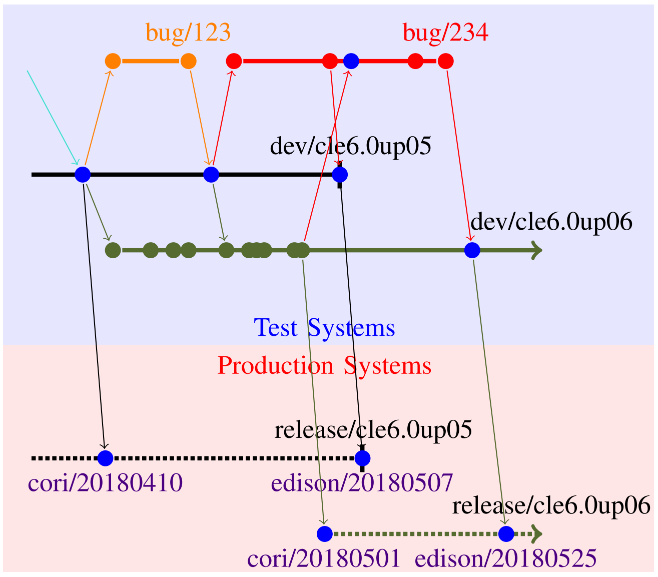

Fig. 2. SMW-gitflow for CLE version updates. The circles represent commits, non-blue are normal development commits, blue are merge commits. Lines between branches illustrate merge directionality.

III. DISCUSSION

A. NERSC Experiences

NERSC has been using an evolving version of these techniques since late 2016, and has successfully updated from cle6.0up02 → cle6.0up03 → cle6.0up04 → cle6.0up05 using a form of what has been presented here (though the full suite of tools was only used for the cle6.0up04 patch maintenances (many) and the cle6.0up04 → cle6.0up05 transition). NERSC has derived a great deal of value from having the edison and cori system configurations fully converged. In many ways it is possible to view them as the same system now, such that most bugs or issues are present on both systems, and a correction of one implements the correction for both. This is despite the fact that these systems are vastly different architectures – edison is an Ivybridge system using RSIP and older networking configurations, while cori is a heterogeneous Haswell and Knights Landing system with DataWarp and using a new Software Defined Networking scheme that NERSC has been developing as well as a variety of other differences. Thus the techniques described here are flexible enough to account for rather large differences in scale, architecture, and configuration.

NERSC runs four Cray XC systems, two production systems (edison and cori), and two test and development systems (alva and gerty). All four systems share the configurations in the git repositories, and a single branch encompasses all four systems. We have found that, in practice, running tests on our larger test system, gerty, is quite predictive of success on either production system. To that end we have specialized the roles of each test system. The alva test system is more focused on experimental and forward looking software installations, such as building new development branches for the latest CLE update level, or attempting new configurations of SMW software. The gerty system on the other hand is more focused on the production state, or sampling configurations based on same SMW/CLE/SLES patch level as the production systems.

Because the test systems can be re-spun so quickly into production configurations, we have found that it dramatically lowers the cost of allowing several different development and administrative activities to proceed simultaneously on the test systems. This has proved invaluable for allowing administrators to experiment on the system to gain experience.

Finally, we have found that we can cut the length of most software-only maintenances. Because the configurations can be tested ahead of time and transferred from the test systems to the production systems via git and smwflow, we can be confident that any bootable configuration tested on gerty is very likely to boot on edison and cori. By staging all the changes the day before and following the procedures described here, we have found that most software maintenances last just 3-4 hours now instead of the longer all day affairs we used to experience. CLE updates are sped up using these techniques but still tend to be a whole-day maintenance.

B. NERSC Future Work

1) Adding More Configurations: In the coming months, NERSC plans to add capabilities to store the HSS configuration files in the same configuration repositories. To support multiple systems, the diversity of configuration values will need to be reviewed to determine the best strategy to try to consolidate the configuration across systems as much as possible. It is likely that we will make use of the Jinja2 templating once again for these. The other complication will be determining when it is safe to deploy these configurations – potentially only when the machine is shut down.

Another configuration target of interest is to consolidate and manage the Sonexion puppet and Lustre server configurations to help improve how the system is managed and ensure that desired values are exposed to the system management team. An initial target is simply to make adjustments to the ssh configuration to meet local security policies.

The final component that we are missing is local configurations and management scripts on the SMW itself. The SMW is largely manually configured with management scripts in /usr/local, not managed by RPM installations. A clear direction here is to start populating the `global_ansible` with plays to be executed by the SMW. The only complication is that `global_ansible/roles` is included in non-SMW ansible roles search path, thus some care will be required to avoid unintended side-effects.

2) System Management Automation: The design objectives of this project required that no human intervention be required for any of the SMW configuration, branch change, boot artifact creation, or even boot processes. This capability allows an automation agent to be used to perform many of the functions described in this document. We have already implemented a proof-of-concept capability to automatically change branches, rebuild localized software packages from remote branch HEADs, rebuild images, and reboot both an Air Cooled XC system as well as the eLogin nodes attached to it using smwflow constructed images and configsets using the Jenkins Continuous Integration software package [17]. This level of automation reduces about nine hours of human work to just over two hours, and could easily enable automated regression testing. The short term goal will be to run builds and tests on up to five branches per day on our software development with full slurm [18] regression test suite and our local reframe [19] regression test suite to validate user functionality. The longer term goal is to begin implementing some level of administrative automation on the production systems, especially attempting to implement limited forms of rolling updates.

3) Secure Commits: One capability we are interested in exploring is further securing our git repositories by using GPG-key signing commits. This would allow us to verify that all modifications originated from our team.

4) Automated Configuration Pre-Computation: The move to always rebuilding configuration sets by using the smwflow tool gives the opportunity to further automate some aspects of the system configuration. As of this writing we have already implemented ssh and LDMS configuration postscripts to smwflow which create new SimpleSync or Ansible content used for configuring those services. Looking more forward, we envision integrating the node group, image_group, and image construction scheme, to directly execute some particularly slow ansible scripts ahead of time in the image chroot and populate the results in SimpleSync. This could be a mechanism for retaining an extremely flexible configuration management system while speeding boot times.

5) Enabling Rolling Updates: Enabling rolling updates of site-local software and configurations is a central objective for the systems engineering team at NERSC. Using the smwflow tools the system over time can now be decomposed to a series of known state transitions. If a given update only requires rebooting non-service nodes like computes and elogins, we may be able to automate the rebooting of nodes to the new configuration state either using CNAT[12], Slurm[18], or automation agents on the SMW to manipulate elogin nodes. One major weakness of rolling updates that the known-state tracking may be able to help with is monitoring the updating system. We cannot disable system monitoring for several days while we wait for a rolling update to complete. Instead, if, when a node boots, it were to communicate which image and CLE config set it booted with (now possible with the smwflow metatdata injected into the config set), then the monitoring system can adapt as specific high-value nodes reboot and potentially services move from one node to another.

6) Boot Performance Tracking: Since the configuration git repository is tracking the state of the system over time it would be advantageous to collect `bootprofiler` data for each configuration set booted. Summaries of these could be committed back into the configuration repository to aid in boot timing bisection to track both which commit levels and patch levels might be related to regressions or enhancements in boot performance.

7) Room for Improvement: The CMF tools allow for exquisite control over the system, however there are some things which could ease integration with external SCM tools like git. The managed json documents, in particular, are not well suited to git management without a little post-processing to ensure that changes over time are properly tracked when merging distant branches. Additionally, it would be advantageous if the `recipe` and `pkgcoll` tool had options to operate on an administrator's copy of the configuration git repository.

Probably the biggest improvement would be if we could standardize a method for injecting configurations to avoid some of the excess scripts that are required to dance configuration data back and forth between the git repository and the SMW. One possibility would be for the SMW to simply read configurations directly out of the git repository. In a limited way this is possible with image recipes and package collections already by overriding the "local edits" files specified in `/etc/opt/cray/imps/imps.json`, however until this is combined with some of the other enhancements, it would be of limited value.

IV. CONCLUSION

The SMW-centralized configuration management in CLE 6.0 is a key feature that enables the work described in this paper. By further organizing and abstracting the configurations of multiple systems into a set of coordinated git repositories, and a minimal software layer to integrate those data onto the SMWs, we can effectively co-manage multiple, diverse systems of varying requirements and capabilities. Using git's enormously flexible branching capabilities and formulating the SMW configurations in such a way that the *entire* configuration of all managed systems is encapsulated by a branch, an administrator is able to mutate a system far afield from its initial configuration and immediately revert back to the production configuration. When applied to a test system, rewriting the SMW configuration from a git branch allows the test system to be used for a variety of development activities, in parallel, while still retaining its value as a model for the production system. Providing these essentially sandboxed branches on a test system affords administrators an opportunity to explore and gain experience with a test system from development and system management perspectives which improves both the configuration and staff capabilities for the production environment. The use of shared configurations between the test and production systems reduces cost by avoiding duplication of efforts, and ensures faithful transfer of the configurations to the production system. Taking advantage of social git services like Bitbucket Server[2] or GitLab[4] enables Peer Review of proposed changes which further increases staff knowledge of changes and provides a feedback mechanism to help improve the configurations, as well as providing mechanisms to interface with Continuous Integration/Continuous Deployment (CI/CD) services like Jenkins CI [17]. Managing the SMW as a git branch is an enabling technology for DevOps methodologies on Cray Systems and will lead to continuous integration of software and configurations increasing system availability, system stability, and system capability for users.

REFERENCES

[1] Hamano, J. and Torvalds, L. (2005): git. Available at http://git-scm.com

[2] Atlassian. (2018): Bitbucket Server. Available at https://www.atlassian.com/software/bitbucket/server

[3] GitHub, Inc. (2018): GitHub Enterprise. Available at https://enterprise.github.com/home

[4] GitLab. (2018): GitLab. Available at https://about.gitlab.com/

[5] Driessen, V. (2010): A Successful Git branching model. Available at http://nvie.com/posts/a-successful-git-branching-model/

[6] Jacobsen, DM (2018): smwflow: gitflow tools for Cray Systems Management. Available at https://github.com/NERSC/smwflow

[7] Cray, Inc. (2018): XC^{TM}Series Ansible Play Writing Guide (CLE 6.0.UP06 S-2582) Available at https://pubs.cray.com/

[8] Cray, Inc. (2018): XC^{TM}Series System Configurator User Guide (CLE 6.0.UP06 S-2560) Available at https://pubs.cray.com/

[9] Redhat (2012): Ansible. Available at https://www.ansible.com/

[10] git-lfs (2014): Git Large File Storage. Available at https://git-lfs.github.com/

[11] Cray, Inc. (2018): XC^{TM}Series System Software Installation and Configuration Guide (CLE 6.0.UP06 S-2559) Available at https://pubs.cray.com/

[12] Cray, Inc. (2018): XC^{TM}Series System Administration Guide (CLE 6.0.UP06 S-2393) Available at https://pubs.cray.com/

[13] Cray, Inc. (2018): XC^{TM}Series Power Management Administration Guide (CLE 6.0.UP06 S-0043 Available at https://pubs.cray.com/

[14] Cray, Inc. (2018): XC^{TM}Series System Environment Data Collections (SEDC) Administration Guide (CLE 6.0.UP06 S-2491) Available at https://pubs.cray.com/

[15] Ronacher, A. (2008): Jinja2. Available at http://jinja.pocoo.org/

[16] Redhat (2018): Ansible Vault documentation. Available at http://docs.ansible.com/ansible/devel/user_guide/vault.html

[17] Armenise, V., (2015) Continuous delivery with Jenkins: Jenkins solutions to implement continuous delivery. RELENG '15 Proceedings of the Third International Workshop on Release Engineering. Available at https://jenkins.io/

[18] Jette, M., Yoo, AB., and Grondona, M., (2002) SLURM: Simple Linux Utility for Resource Management, In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003. Available at http://schedmd.com

[19] Karakasis, V., Rusu, VH., Jocksch, A., Piccinali, J-G., and Peretti-Pezzi, G. (2017) ReFrame: A regression framework for checking the health of large HPC systems. Cray User Group 2017. Available at https://github.com/eth-cscs/reframe