Continuous Deployment Automation in Supercomputer Operations: Techniques, Experiences and Challenges

Nicholas P. Cardo, Matteo Chesi, Miguel Gila
Swiss National Supercomputing Centre (CSCS)
Lugano, Switzerland
Email: (cardo|chesi|gila)@cscs.ch

***Abstract*—Continuous Deployment (CD) is a software development process that pursues the objective of immediately deploying software in production to users as soon as it is developed. CD is part of the DevOps methodology and has been widely adopted in the industry including large web technology companies like Facebook, Inc. and Amazon.com, Inc. to release new features to their public.**

The Swiss National Supercomputing Centre (CSCS) has an interest in adopting CD for the software platform of their supercomputer Piz Daint. The initial implementation provides an automated Continuous Deployment Process (CDP) that is capable of deploying new compute node configurations, including changes to the Programming Environment. Changes can be propagated across the whole system in a relatively short time thereby minimising the use of resources and the possible impact on user workloads.

In order to implement a more comprehensive CD workflow in HPC Operations further barriers must be dismantled and overcome in future supercomputer architecture designs.

***Index Terms*—Programming Environment; System Updates; CLE; Cray XC; Continuous Deployment;**

I. Introduction

Adopting a CD workflow yields large benefits such as reducing the time to deliver new features and time spent performing manual activities. Achieving this requires a great effort on the transformation of the whole software development process and the complete automation of software testing and deployment phases.

At the Swiss National Supercomputing Centre (CSCS), the HPC Operations Compute Services Group (HPC-OPS-CS) is responsible for maintaining the operational status of Piz Daint and the deployment of the software platform in which the users build and execute their own software applications. In order to provide a reliable up-to-date software platform to the users, HPC-OPS-CS has increased the cadence of software changes and deployment on Piz Daint from the previous monthly scheduled maintenance to a continuous process that evaluates the urgency, criticality and risk of each change to be performed both live or through system reboots.

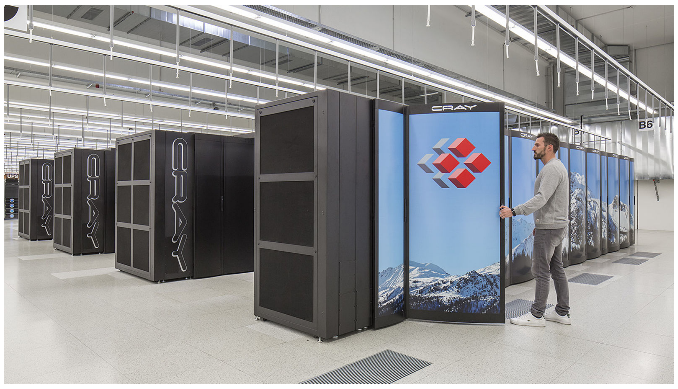
Fig. 1. Piz Daint

Piz Daint is a Cray XC™ 40/50 system with 5704 Hybrid Compute nodes and 1813 Multicore nodes in a single system partition. Each Hybrid Compute Node consists of 1 Intel® Xeon® E5-2690 V3 processor at 2.60 GHz (12 cores), 64 GB of DDR4 memory, and 1 NVIDIA® Tesla® P100 16GB. Each Multicore node consists of 2 Intel® Xeon® E5-2695 V4 processors at 2.10 GHz (18 cores) and 64 or 128 GB of DDR 4 memory. [1]

II. Objectives

One of HPC-OPS-CS goals is the automation of common system operations. In this scope, the effort of increasing software update frequencies would benefit from the adoption of a CD workflow. CD provides an ideal solution to the problem of software feature and change deployments on Piz Daint by removing the need for manual operations. Automation provides the ease of reproducibility with the benefit of reduced manpower to install updates across the system.

Unfortunately, the implementation of CD theory in an HPC environment with supercomputers is more than just a matter of will. CD and other DevOps practices are often associated with cloud computing which is a totally different environment from a supercomputer. Where cloud computing has standardized generally available hardware resources to allow the creation of development or testing environments on demand, a supercomputer like Piz Daint has very specialized hardware, 99% resource occupancy and hours of queue wait time.

With the needs and benefits identified, the following design objectives could be established:

1) Ease of use
2) Reproducability for subsequent updates
3) Reduction of required manpower

A. Ease Of Use

In the end, the solution had to be easy to use. Creating a complicated solution that was difficult to use and understand would only detract from the benefits and deter future use.

If the implementation of the solution is difficult to get operational or difficult to maintain, this too would detract from the benefits and deter future use.

Therefore, "*Ease of Use*" could be quantified by the following:

- Ease of installation and setup
- Easy to understand and use
- Simple to maintain

B. Reproducibility for Subsequent Updates

It is important that each time the same update is performed, the solution responds consistently. The expectation is also that applying an update to something that has already been updated through this process, will complete in the same manner as the first time.

Regardless of the update being applied, the solution should perform similarly and predictably.

Therefore, "*Reproducibility for Subsequent Updates*" can be quantified by the following:

- Highly Reliable
- Reproducibility
- Consistency

C. Reduction of Required Effort

It is expected that the first time an update is performed that an effort is required to set it up. However, with subsequent updates, it is anticipated that this effort would be drastically reduced or even eliminated.

Additionally, the effort required to setup an update should be less than the time required to perform the update manually. Automation should reduce the effort required to complete the update, thereby freeing up human resources for additional tasking.

Therefore, "*Reduction of Required Effort*" could be quantified by the following:

- Quick setup for repetitive updates
- Improved timings for updates compared to manual efforts

III. Design Challenges

As a preliminary step towards CD implementation, HPC-OPS-CS analyzed and compared the theoretical implementation of a CD workflow for their HPC Operations on the:

- Public Cloud
- CSCS OpenStack Private Cloud
- Flagship System Piz Daint

This provided the means to evaluate which of the various platform specific features would affect the CDP and how to best plan for a CD roadmap.

In order to design the CDP, it was necessary to develop automation test-cases in order to validate the process. CSCS implemented a set of quick tests for the Piz Daint hardware and software components as well as a comprehensive application regression test suite based on ReFramei [2]. By leveraging the availability of already developed tests on Piz Daint, a semi-automated workflow to remove and return compute nodes from production service could be developed.

IV. Background

With the introduction of Cray Linux Environment 6.0 (CLE 6), the reconfiguration of service and compute nodes on a Cray XC™ system can be manually initiated by a System Administrator at any time. This can potentially lead to a system having service and/or compute nodes with a configuration out of sync across the entire system.

In most cases, users expect to have a homogeneous system in terms of software and configuration. The expectation is that all compute nodes have similar configuration files, services, software, and filesystems, thereby establishing a consistent known computing environment. However, there are times when it is desired to have multiple configurations active across the system. This occurs when changes are applied to the system in a rolling manner or when configuring and adapting a system after a major change or upgrade.

Cray Scalable Services designates nodes as SoA (Server of Authority), tier1, tier2, or tier3. The System Management Workstation serves as the SoA while the Boot and System DataBase (SDB) nodes serve as tier1 nodes. Tier2 nodes are either service nodes or repurposed compute nodes. Each tier is therefore a client of its predecessor in the boot hierarchy [3]. Prior to CLE 6 update UP07 (CLE 6.0.UP07), the Cray Programming Environment (PE) was deployed by the tier2 nodes using a Cray Data Virtualisation Service (DVS) projected filesystem to the compute nodes making it easy to update the PE across the system. All that was required was to update the image directory on the System Management Workstation (SMW), push the changes with rsync to the boot node, and compute nodes would automatically pick up the new changes after a brief period of time. It is standard operation procedure at CSCS to keep multiple versions of the PE installed on the system, while changing the default is performed less frequently. Users were amenable to this because the same default PE would be in place for long periods of time, but also provided access to newer releases of the PE that was routinely installed shortly after release.

However, with CLE 6.0.UP07, Cray introduced a change to the way the PE is deployed across the the system by utilizing a `squashfs` filesystem. The PE `squashfs` file is generated on the SMW, then pushed to the boot node and projected to the compute nodes by utilizing the same tier2 nodes. The compute nodes then mount the `squashfs` filesystem locally as a loop device. This process provides excellent performance because the DVS layer only has to project a single `squashfs` file instead of a few million small files. Unfortunately, the consequence of this process is that the PE image cannot be updated across the complete system while jobs are running. This is due to the need to run Ansible which effectively unmounts the old PE `squashfs` file and mounts the new one. The end result is that the PE cannot be pushed out to nodes that have jobs running on them without creating problems.

Keeping the configuration consistent across the system and updating the PE has become a very costly manual operation as nodes now need to be drained before the updates can be applied. This becomes increasingly difficult with large-scale systems such as Piz Daint, adding to the challenges of implementing a CD workflow.

V. Design Elements

Since the commissioning of Piz Daint and Piz Dom in November 2016, both systems implemented a set of operations that automatically drain unhealthy nodes and prepare them for corrective actions. Nodes can be drained for a variety of reasons, including hardware problems (i.e. GPU XID) or software problems (i.e. Slurmd spooldir full). This process, called Node LifeCycle Management (NLM), has been used as the foundation for implementing the new CDP. Working from the basic idea that a newly available update release could be handled as a special case of node failure, the fundamentals of the design could be established.

In the NLM workflow, the state of a node allocated by Slurm is determined by the combined use of different features including node state, reservation names and memberships, and drain reasons. [4] NLM is currently a semi-automated process where most of the operations are performed by cron and the Slurm prolog and epilog. However, it also requires manual intervention of an operator or adminstrator to perform the most delicate parts.

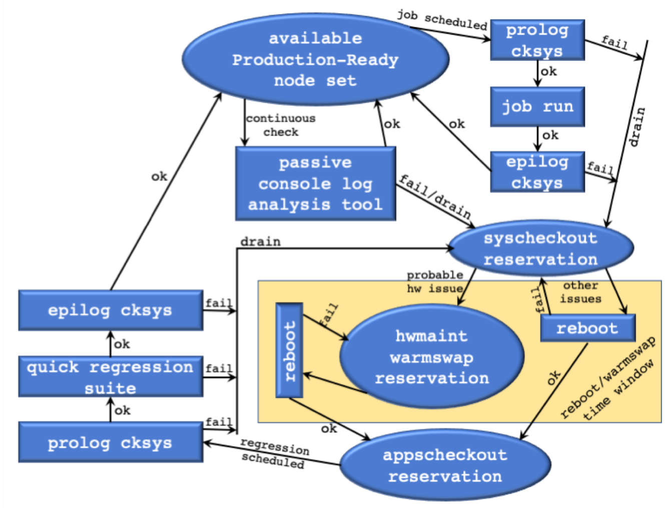

Fig. 2. NLM Workflow

One of these critical operations on Piz Daint is a compute node reboot.

HPC-OPS-CS's experience with managing Piz Daint has shown that in certain situations under specific conditions, what would normally be a standard compute node reboot degenerated into a system-wide incident.

The majority of these incidents were found to be categorized into two specific cases. The first case being the result of the SMW trying to maintain the health of the system at the scale of Piz Daint. The second case is due to the nature of the High Speed Network (HSN), the Cray developed Aries™ [5] interconnect, that requires rerouting and network quiesces when nodes are removed or added to the network.

Because of these criticalities on Piz Daint, node reboots are always manual operations that require an operator or system administrator monitoring and supervising the system. Moreover, reboot operations are restricted to certain time windows to prevent their impact on user performance tests.

For these reasons, it was decided to exclude node reboots from our process design. This is one of the main differences between the CSCS implemented process and a standard implementation on cloud-based systems where spawning and deleting node or container instances is the most common way to deal with software or configuration updates.

The list of design features for the design of the new process are:

- F1 - Exclude Node Reboots
- F2 - Idle Nodes
- F3 - Scalability
- F4 - Publish and Retire Udpates
- F5 - Node Selectability
- F6 - Restrict Simultaneous Updates
- F7 - Implied Boot State
- F8 - Fully Automated
- F9 - Single Reservation

A. *Exclude Node Reboots (F1)*

From HPC-OPS-CS's perspective, excluding node reboots from the process provided the opportunity to minimize the time needed to perform update operations. This is due to physical node reboots being more time consuming than re-spawning a Virtual Machine (VM) or a container. The time saved in service operations is additional compute time available for scientific research.

B. *Idle Nodes (F2)*

Applying an update to a compute node that is actively processing an application could lead to unpredicatable results. Rather than risk applying an update that causes problems for the active application, it has been determined that updates should only be applied to idle nodes.

C. *Scalability (F3)*

Future systems at CSCS could be even larger in scale than the current Flagship System, Piz Daint. The CDP not only has to work at the current scale of Piz Daint but also be capable of running on future, and potentially larger, systems.

D. Publish and Retire Updates (F4)

To achieve reliability and servicability, the capability to publish and retire an update bundle is needed. Conditions may arise where in the middle of the deployment process a problem is detected with the update. Rather than wait for the process to complete incorrectly, a mechanism was required in order to terminate the active update process, revise it, and restart it.

E. Node Selectability (F5)

There may be situations when an update only needs to be applied to select groups of nodes. This may be due to the specific functionality provided by individual nodes as all nodes may not provide equivalent functionality. An added benefit provided with this functionality is the ability to test an update on a few nodes before applying it to the entire system.

F. Restrict Simultaneous Updates (F6)

Care must be taken to not apply the update to the entire system at once. This would result in the system draining and going idle. By limiting the number of nodes that can be simultaneously taken out of service for updating, production operations can be maintained. This holds true even if large quantities of nodes have become idle on their own. Every caution must be taken to prevent the CDP from blocking production operations. The CDP needs to run in a manner that is virtually undetected by the user community.

G. Implied Boot State (F7)

When nodes are booted they are considered to be already updated. If a node is rebooted for any reason, it must return to service without the need for further update operations. If nodes were returned to service without being updated, then they would need to be drained again, wasting valuable compute time.

H. Fully Automated (F8)

The objective of implementing a CDP is to complete the repetitive task of applying updates in an automated manner that is faster, thereby freeing up staff. Automation breeds efficiency by reducing staff overhead.

I. Single Reservation (F9)

Nodes would be identified for updates by placing them into a special Slurm reservation. There are a number of activities as part of the NLM process already in place at CSCS which also utilize Slurm reservations. It is very easy to create numerous node reservations which can lead to confusion. Therefore, the CDP needs to work within a single Slurm reservation. Keeping it simple reduces the chances of confusion, making standard operational activities and troubleshooting a bit easier.

VI. Workflow

The CDP has been named the Node Update (NU) service. It is designed to automatically perform updates on idle nodes, checking them regularly via a cron job or by reserving a node after the end of a scheduled job. This is accomplished by using a specific hook in the slurm epilog script, delivering the feature F2.

In the NU design, an update is provided in a specific directory at a specific path on a shared file system. The directory name is the codename of the update/deployment. The directory contains a script called `update.sh` that will perform the required update for a single node.

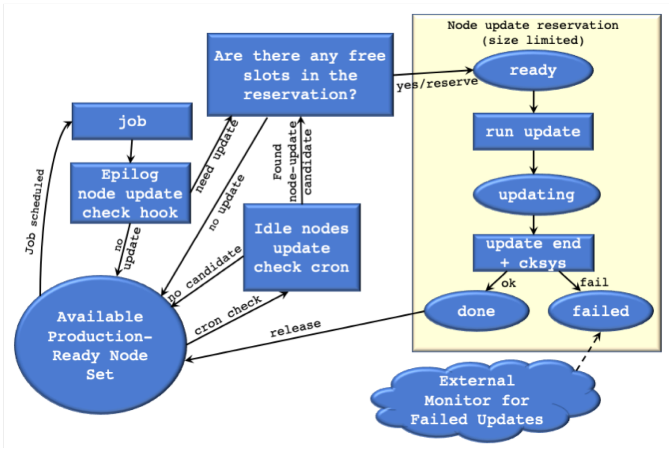
Node update reservation (size limited)
Are there any free slots in the reservation?
yes/reserve
ready
run update
updating
update end + cksys
ok
fail
done
failed
External Monitor for Failed Updates
job
Job scheduled
Epilog node update check hook
no update
need update
no update
Found node-update candidate
Idle nodes update check cron
no candidate
cron check
release
Available Production-Ready Node Set

Fig. 3. NU Workflow

Another file, `active-updates.list` on the shared file system contains the list of active updates associated with the nodes where they must be applied, accomplishing feature F5. Publishing a new update consists of appending a line to `active-updates.list` file. Removing the line disables the update operation, providing feature F4.

The codename of each update bundle has a timestamp format and every node at boot records the boot time with the same format in the local configuration file `local.version`. The timestamp format used is `YYYYmmddHHMMSS` where:

- `YYYY` represents the four digit year
- `mm` represents the two digit month number
- `dd` represents the two digit day of the month
- `HH` represents the two digit hour
- `MM` represents the two digit minutes
- `SS` represents the two digit seconds

After a successful update operation, the `local.version` file's contents will be updated with the codename/timestamp of the newly applied update.

In this manner, the operation of checking the need for an update is a simple comparison between the `local.version` file's contents and the `active-updates.list` file's contents. This check is performed locally on every node.

In order to perform the updates only on idle nodes without any conflict with running jobs, all the update operations are performed when the nodes are both idle and reserved in the

Slurm `node_update` reservation identified by feature F9. The quantity of nodes in the reservation is artificially limited by the NU workflow with a tunable parameter, `RES_LIMIT`. This provided the means for delivering feature F6.

The complete NU workflow described in figure 3 consists of the following states:

- IDLE
- READY
- UPDATING
- DONE
- FAIL

A well choreographed set of guidelines controls the flow of nodes through the progression from one state to the next.

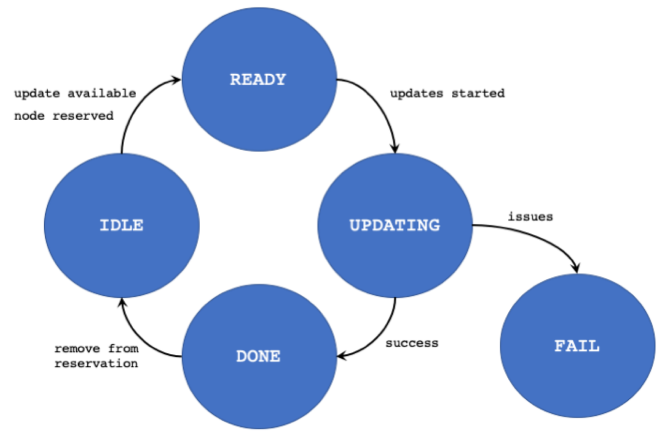

Fig. 4. State Diagram

A. IDLE State

A node is in the IDLE state because it is either waiting for a new allocation or the last job just finished. In the first case, a cron job will check to see if a new update needs to be applied and free slots exist in the `node_update` reservation. The second case is handled by the Slurm epilog script which performs the same actions. If there is a new update available and there are free slots in the Slurm reservation, the node is added to the `node_update` reservation. This action moves the node into the READY state

B. READY State

A node is in the READY state when it is in the `node_update` reservation and there is an update to apply. Every node in this state can start update operations. Once the updates start, the node transitions to the UPDATING state.

C. UPDATING State

A node will perform all available updates sequentially one after the other. If any update operation fails, the node transitions to the FAIL state, otherwise it continues the updates until all are performed. When all the available updates have completed successfully the node transitions to the DONE state. These are the same system checks used in NLM process and if no issues are identified, the node can transition to the DONE state. As a last step before changing state, the `local.version` file is updated.

D. DONE State

Once a node has completed its available update operations, it can be returned to production service. As quickly as possible, the node is removed from the `node_update` reservation and it can transition back to the IDLE state.

E. FAIL State

Any node that has failed to successfuly perform an update operation is placed in the FAIL state. It must stay in the `node_update` reservation and manual intervention is required to resolve the incident.

F. Overcoming Limitations

In the NU workflow, on the contrary to NLM, the node state is mapped to a local file and the interaction with Slurm is limited to the addition and removal of nodes to/from the `node-update` reservation.

In the initial phase of the implementation of the NU workflow it was observed that the simple operation of checking how many free slots are available for the `node_update` reservation could result in scalability issues. The query to determine how many nodes are in the reservation is a simple Slurm command, but if performed by thousands of `epilog` scripts simultaneously, could result in problems with the Slurm daemon.

To solve this problem a new agent was introduced in the workflow called the *Orchestrator*. This new agent is the only one entitled to communicate with the `slurmctld` in order to protect it from too many simultaneous requests and to cache the query results. To protect the *Orchestrator* itself from too many concurrent requests, all the message handling between the nodes and the *Orchestrator* happens via a message bus. This was overcome by using Apache Kafka® as a scalable solution to implement a many to one/one to many message bus.

> *Kafka® is used for building real-time data pipelines and streaming applications. It is scalable, fault-tolerant, wicked fast, and runs in production operations in thousands of companies.* [6]

Kafka® runs on an available service node in the system and required the installation of Python pip.

The introduction of Apache Kafka® as a message bus required that all compute nodes to be capable of acting as Kafka® clients. This meant that each compute node had to be able to produce and consume messages from a Kafka® topic [6]. The standard Kafka® client is based on a Java API, but in the design of the *Orchestrator* and compute node client, it was decided to base the development on the `kafka-python` library available through the python package installer pip.

The *Orchestrator* agent's design has several responsibilities:

- Manage the communication between the compute nodes and the `slurmctld`, aggregating compute nodes request into only a few Slurm commands. A small delay in the communication is enforced in order to allow this aggregation to happen.
- Manage the `node_update` Slurm reservation. Only the *Orchestrator* is entitled to add/remove nodes to/from the reservation, and also human operators are required to use a specific API to manipulate the reservation.
- Schedule tasks on the compute nodes in the `node_update` Slurm reservation in order to advance the NU process workflow.

The creation of the *Orchestrator* agent provided the ability to workaround Slurm scalability issues. At higher scales, the *Orchestrator* can be scaled-out into a distributed agent thanks to the scalability of the Kafka® message bus solution.

Indeed, the optimal solution to this problem would be to enhance Slurm to allow the job scheduler to scale-out with the size of clusters and their user base. Also, equiping it with an cache-endowed API that is resistant to the clients' abuses will also improve scalability issues.

VII. Early Results

Based on the foundation of previous work, HPC-OPS-CS's latest effort towards CD on Piz Daint was the implementation of an automated CDP. This enables the deployment of new compute node configurations, including changes to the Programming Environment on the whole system in just a few hours/days. The approach taken minimized the impact on the use of resources and the possible impact on the users workload.

The deployment of changes on Piz Daint through CDP without disrupting user workloads enabled CSCS to perform Blue-Green Deployments in order to mitigate the risk associated with the validation of any possible software change.

In order to evaluate the actions that such a process will introduce into a running system, a dry-run mode has also been implemented. This mode easily facilitates various test scenarios without impacting the running system.

VIII. Next Steps

With the early successes achieved during development brings the next phase of full-scale production implementation. The flexibility of the solution has shown benefits in other areas which need to be explored.

A. Staged Production Rollout

This work has been developed on Piz Dom, a 64 node Cray XC™ 40/50 Test and Development System (TDS). In the near future the CDP will be evaluated at large scale. This will be accomplished in three stages:

- **First stage:** Deploy on Grand Tave, our production Cray XC™ KNL platform with 164 compute nodes. This system is busier and larger than Piz Dom, which will allow the identification of issues still at a somewhat larger scale.
- **Second stage:** Identify and correct issues identified in the first stage, then deploy the CDP on Piz Daint. By utilizing the dry-run mode, the process can be used to evaluated how the overall system, including the Kafka® implementation, works at scale.
- **Third stage:** Turn off dry-run mode and evaluate how this works in production at scale.

B. Additonal Features

Early testing has identified value in pursuing the incorporation of the steps currently being performed by cron jobs into the CDP. This will add greater flexibility and control for all the work performed within the CDP.

C. Expanded Scope

CSCS is planning to adopt CDs in more general terms and will be investigating the introduction of a site-wide Kafka® platform. This could potentially permit the synchronisation of system-related tasks with other high-level organization-wide operations. For instance, the population of the Slurm database for each of our systems is currently done individually per cluster at specific times via a set of complex cron jobs. This could be replaced by a policy-event driven CDP that links to the CDPs for each system. This will effectively synchronise high-level events, such as account creations, with the population of the Slurm database in moments suitable for the scheduler running on each system or even in coordination with storage and network operations.

IX. Conclusion

CD in HPC environments and on supercomputers can already be a fact for userlevel scientific applications; on the other hand, some hurdles prevent a complete implementation for the low-level software platform of a supercomputer like Piz Daint. Some of the issues are related to traditional HPC and scientific production workflows, while others are related to the constraints of current system technologies and designs.

Following the strong interest in software automation at scale, CSCS built a CD workflow for Piz Daint for compute node configurations and the Programming Environment. However, in order to achieve a more comprehensive result, further barriers must be dismantled and overcome in future supercomputer designs.

One of the design challenges of a CDP on production HPC systems, is the lack of idle resources. Typical HPC systems usually target a utilization of 99%, which makes on-demand, real-time resources not commonly available. Overcoming this required creativity when deciding how to select blocks of resources that could be idled for updates.

HPC systems work to maximize the utilization of the compute resources and minimize any idle resources. The scientific workload not only can scale up in concurrency but also in duration of the runs. The Continuous Deployment process must work in symbiosis with the job scheduler by taking advantage of the bits and pieces of idle time that naturally emerge.

The job scheduler itself is a component that needs continuous improvement in top-scale HPC platforms. CSCS protected the job scheduler from the interaction with too many clients by building an *Orchestrator* agent as a communication interface. Scalability features of this interface could be integrated in the job scheduler to allow future systems and user-base expansions.

Continuous Deployment on standard IT platforms leverage virtualization and containerization techniques. Today, HPC centers use software abstraction as a solution to facilitate software portability in extreme cases. However, in distributed web platforms, software abstraction is usually the foundation for any automation process. Caution towards software abstraction in HPC is due mainly from the possibility of performance loss or unpredictability. This problem can be observed as a minor issue in web production environments, but is a difficult hurdle in HPC. HPC must always chase the highest possible performance of the hardware platform, therefore, strong attention must be paid when integrating any software abstraction layer.

In the scope of our experience with Piz Daint, Cray provides a large portion of the software platform. In order to be able to continuously deploy the platform in its whole, further collaboration is needed from manufacturers in the design of software, firmware, and hardware. In absence of any abstraction layer, great care must be taken in how the software platform is bundled, released, and tested. The effort to make changes and revert them back (rollback) on busy production systems should be taken into consideration by all architects and developers involved in HPC system design.

Another barrier to overcome in future system designs is the reliability of the High Speed Network (HSN). Experience with Piz Daint's Aries network demonstrated excellent performance in standard operation scenarios. However, some design choices limit the reliability of basic standard operations like compute node reboots. [5] This fact influences system operation workflows at CSCS and the implementation of the CDP on Piz Daint.

In future designs, the reliability of the HSN and of operations like node reboots, must be taken under serious consideration. Moreover, in traditional HPC environments where idle time is perceived as a threat to be fought, improving the overall time needed by node reboots could be a key factor for implementing more dynamic automation processes like continuous deployment.

Besides the current system limitations, CSCS is looking forward to bringing this feature from the current initial stages to full production on Piz Daint. CSCS is confident that introducing Continuous Deployment on HPC system platforms like Piz Daint could be an important step forward in the automation of traditional HPC system administration tasks. Doing so could facilitate the transition to a more DevOps oriented style of work within the HPC Operations Team at CSCS.

X. Acknowledgments

This work was supported by the Swiss National Supercomputing Centre (CSCS).

XI. Trademarks

Kafka® is a registered trademark of The Apache Software Foundation.

Intel® and Xeon® are registered trademarks of Intel Corporation.

Aries™ is a trademark of Intel.

XC™ is a trademark of Cray Inc.

References

[1] CSCS, "Piz Daint." http://www.cscs.ch/computers/piz-daint. (Accessed April 2019).

[2] CSCS, "RedFrame." https://reframe-hpc.readthedocs.io. (Accessed April 2019).

[3] Cray Inc., *XC™ Series Boot Troubleshooting Guide (CLE 6.0.UP03) S-2565*, pp. 25–26.

[4] SchmedMD, "Slurm workload manager - documentation." https://slurm.schedmd.com. (Accessed April 2019).

[5] Bob Alverson, Edwin Froese, Larry Kaplan, Duncan Roweth, Cray Inc., "Cray® XC™ Series Network." https://www.cray.com/sites/default/files/resources/CrayXCNetwork.pdf. (Accessed April 2019).

[6] Apache Software Foundation, "Apache Kafka." http://kafka.apache.org. (Accessed April 2019).