Rev Up Compute Node Reboots

Reboots 2x to 5x Faster (WIP)

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

Paul Selwood
Principal Fellow Supercomputing
UK Met Office
Exeter, England - United Kingdom
paul.selwood@metoffice.gov.uk

ABSTRACT

Recognizing that reboot-related downtime directly impacts HPC ROI by reducing usable compute minutes, "Rev Up Compute Node Reboots: 2x to 5x Faster" outlines a methodical approach to analyzing and significantly optimizing reboot times. By instrumenting each phase—shutdown, boot, and post-boot personalization—the paper reveals how to pinpoint delays and streamline the process.

Using DevOps(1) practices for safe, repeatable changes, the team accelerated reboots by offloading install tasks, parallelizing service startups, and fine-tuning orchestration. On Cray System Management (CSM) based systems, these efforts have cut the average reboot time from 35.5 to 10.4 minutes, boosting system responsiveness, availability, and operational efficiency.

INTRODUCTION

A supercomputer's return on investment (ROI) is frequently measured in terms of total job execution minutes across its operational lifespan. These compute minutes are the currency of productivity, directly correlating with the system's scientific, engineering, or analytical output. However, this metric is significantly impacted by periods of downtime—especially those incurred during node reboots. These reboots, while sometimes routine, are often critical for integrating essential software dependencies, applying security configurations, performing system maintenance, installing firmware or kernel updates, and recovering from unplanned incidents such as hardware faults or kernel panics.

The value of these compute minutes is particularly high during the early phases of a system's deployment. At this stage, the system typically represents a leading-edge platform with capabilities that far surpass its predecessors. The scientific and commercial communities are often eager to leverage these capabilities to gain early performance insights or solve complex problems previously out of reach. As such, any interruption to service—planned or unplanned—represents not only a technical hurdle but a loss of strategic opportunity.

To ensure system availability, operations teams are often bound by uptime service-level objectives (SLOs) or formal service-level agreements (SLAs) with stakeholders, institutional partners, or commercial customers. These commitments serve as both accountability mechanisms and operational benchmarks, reinforcing the importance of efficient, reliable system behavior—particularly during transitional states like reboot cycles.

One especially important operational requirement is the ability to reboot compute nodes quickly, especially when switching between system images that vary in their security hardening or software stack. This capability is crucial in multi-tenant environments or in classified computing workflows where nodes must transition securely and swiftly between enclaves.

To address this, the example system's procurement contract included a key performance requirement: each compute node must be capable of rebooting within 15 minutes—originally targeted at 10 minutes—under typical operational conditions. Meeting this constraint demanded a multi-disciplinary effort. System engineers collected and analyzed thousands of reboot data points across various scenarios, accounting for differences in hardware initialization, BIOS and firmware delays, network boot infrastructure, parallelized provisioning methods, and OS-level service startup sequences.

Based on their findings, the team developed a targeted plan for optimization. This included streamlining early bootloader stages, reducing or parallelizing systemd service dependencies, optimizing image distribution mechanisms, and implementing faster security policy verification routines. In cases where kernel customizations or tuning offered material improvements, bespoke patches were applied and validated through regression testing.

As a result, the reboot process was significantly improved—meeting the contractual requirement and reinforcing the HPC system's overall resilience and responsiveness. These optimizations not only reduced downtime but also enhanced operational agility, allowing for rapid deployment of updated security images, reduced mean time to repair (MTTR), and increased confidence in fulfilling service-level commitments.

CCS CONCEPTS

• Computer Systems Organization
• Availability
• Maintainability and Maintenance
• Reliability
• Maintaining Software

KEYWORDS

Supercomputer ROI, Compute minutes, Node reboots, Downtime mitigation, System availability, Service-level objectives (SLOs), Service-level agreements (SLAs), Reboot performance, Reboot optimization, Image provisioning, Boot orchestration, Firmware initialization, BIOS delays, systemd service optimization, OS boot performance, Parallel provisioning, Operational agility, Image distribution, Early bootloader optimization, Transitional system states, Reboot time benchmarking, Infrastructure orchestration

1 Reboot Measurement Methodology

To systematically analyze and reduce the reboot time of a compute node within an HPC supercomputer, a dedicated automation workflow was developed. This workflow was responsible for capturing, parsing, and correlating key metrics across multiple system components immediately following a reboot event. The primary sources of data include:

- Console Output Logs – Captures real-time messages during shutdown and early boot stages from the system console.

- CFS Logs – Provides detailed output from the execution of Ansible at boot time from then Configuration Framework Service (CFS).

- Job Scheduler Node Status – Offers visibility into the readiness of the node to run jobs. The finish line in the measurement is when the node registers as ready to run jobs.

These logs were automatically collected and parsed using custom scripts immediately after each reboot cycle. The resulting time-stamped metrics were imported into a Jupyter notebook environment for interactive analysis and visualization. Graphs were generated to visually highlight the duration and order of each stage, enabling both high-level insights and granular inspection of potential delays.

To streamline and standardize the analysis, the reboot process was decomposed into the following three primary stages:

Shutdown

Encompasses all activities from the termination of running jobs to the complete power-off of the node.

Key steps include:

- Job termination initiated by workload manager
- Synchronization of shared file system data
- Operating System shutdown processes
- Physical or virtual power-off of the compute node

Boot

Covers the initial bring-up of the node until the basic OS environment is available for logins.

Key steps include:

- Power-on and hardware POST (Power-On Self-Test)
- DHCP negotiation and PXE boot process (if applicable)
- OS bootstrapping via kernel/initrd loading
- Execution of systemd units up to the point where sshd becomes active and the node is accessible via SSH

Personalization

Includes post-boot system customization and health validation routines prior to marking the node as ready to run user application jobs via the workload manager.

Key steps include:

- Continuation of systemd service initialization beyond SSH readiness
- Application of site-specific configuration via CFS personalization
- Node Health Check (NHC) routines
- Start-up and registration of the PBS (Portable Batch System) MOM (Message Oriented Middleware) service

This structured breakdown allowed for a more detailed understanding of where time was being spent during the node reboot process and helped to identify specific bottlenecks within each stage.

2 Testing Procedures

This section enumerates the approach to testing reboot times, ensuring repeatable outcomes and quantitative improvement.

2.1 Preparation

To prevent experimental changes from impacting production images and configurations prior to formal validation, a rigorous isolation process was implemented during test preparation. All modifications—whether to system services, packages, or boot logic—were committed to dedicated feature branches in the relevant Git repositories. This included both the customer-maintained infrastructure repositories (e.g., configuration management, site overlays, orchestration tooling) as well as the upstream HPE-provided repositories (e.g., csm-config, cfs-config,

and system software layers maintained by HPE). This branching strategy enabled safe parallel development and provided a mechanism for peer review and CI/CD pipeline integration before promotion to production branches.

On the configuration side, separate Configuration Framework Service (CFS) definitions were created for both **build-time** and **run-time** phases. For build-time, a new CFS layer definition was added or cloned from the baseline (e.g., site-config-build) and customized with additional Ansible layers, including those responsible for experimental changes. These layers were version-locked via SHA to ensure reproducibility and traceability. For run-time, a corresponding site-config-runtime CFS definition was created, which reflected the intended node-level configuration during system boot and operation. These configurations were registered in the CSM environment and tagged with unique identifiers to ensure they were not mistakenly used in production workflows.

Using these isolated CFS configurations, a separate bootable image was built using the Image Management Service (IMS). This image build was tied to the custom build-time CFS definition and included any modified packages, services, or node configurations under evaluation. The image was registered independently in IMS and assigned to a dedicated Boot Orchestration Service (BOS) session template, which mapped to the experimental image, CFS config, and test node(s).

By explicitly separating all components—Git branches, CFS configurations, IMS images, and BOS templates—the test environment maintained strict isolation from production. This architecture not only ensured test repeatability and safety but also enabled quick rollback and diffs between experimental and baseline states. It also aligned with DevSecOps principles by enabling automated validation in pre-production contexts prior to any system-wide promotion.

A single CI pipeline was created to deploy and integrate source code feature branches with the latest running configuration, produce configurations and image artifacts.

[Insert Image of Jenkins Blue Ocean Pipeline]

2.2 Reboot Testing

To continuously track and evaluate reboot performance across system updates, a controlled testing setup was implemented using Cray System Management (CSM). This setup ensures that changes in system configurations or software updates do not inadvertently introduce regressions in reboot times, which can impact overall system availability and job throughput.

The testing methodology involves two dedicated compute nodes within the system, reserved specifically for reboot timing tests. One of these nodes runs the current production configuration and serves as the baseline for performance comparison. The second node is configured with the new or experimental configuration under evaluation. This parallel setup allows for direct, side-by-side comparisons between known-good and new configurations, making it easier to detect performance deviations introduced by software or system changes.

A fully automated pipeline orchestrates the reboot testing process. This pipeline initiates a reboot of both test nodes using the Cray Boot Orchestration Services (BOS), a component of CSM responsible for managing system boot operations. Timing begins precisely when the BOS session for each node starts. BOS inherently records both the start and end timestamps for each reboot session, which provides initial timing data.

However, since the metric of interest is the amount of time a node is actually unavailable to execute workload manager jobs, the "stop time" is determined not solely from BOS, but from the moment the job scheduler (e.g., Slurm or PBS) reports the node as back in a usable state (typically "idle" or "ready"). This approach reflects the real-world impact on HPC operations, as it measures the full duration from reboot initiation to job-ready state.

Throughout the reboot cycle, the pipeline monitors the BOS session status, waiting until the session is marked as "completed" or until a timeout threshold of one hour is reached. If the session is not complete within the timeout, it is logged for further investigation. This timeout mechanism prevents the pipeline from stalling indefinitely due to failed or hung reboots.

This automated testing framework enables consistent, repeatable measurement of reboot times, supporting ongoing system tuning and validation of changes. By maintaining visibility into reboot performance over time, the team can quickly identify regressions, optimize configurations, and ensure a responsive, high-availability environment for HPC workloads.

[Insert image of Jenkins Blue Ocean Pipeline]

3 Analysis

This section describes the techniques used to retrieve data from the reboot, parse it, and aggregate it into a simplified view. Data is automatically fetched and processed once the BOS session reports the status is “Completed”.

3.1 Console Logs

Console logs are downloaded directly from the BMC file system. The prerequisite for this is having a root ssh credential to the BMC endpoint. For Cray EX hardware, console logs are located at /var/log/[node number]. Below is an example command for fetching console logs.

Example 3.1.1

```
# scp x1000c0s0b0:/var/log/n0/current
```

Once downloaded, the log is parsed for certain start and stop regex lines to determine the duration of certain stages. A helper function was created to make this easy to understand from the code.

Figure 3.1.1 - Example console log parser

```
echo "BOS_Session_Id,Test_Type,Stage,Minutes" >"$this_csv_file"
{
    get_shutdown_time
    get_duration_between "BIOS" \
        "PeiCore.Entry" \
        "NBP filename is ipxe.efi"
    get_duration_between "DVS_generate_node_map" \
        "DVS: loaded dvsproc module." \
        "DVS: node map generated."
    get_duration_between "Dracut" \
        "wicked: net0: Request to acquire DHCPv4 lease with UUID" \
        "mount is: mount -t squashfs -o loop /tmp/cps/rootfs /new_root/lower"
    get_duration_between "iPXE" \
        "NBP filename is ipxe.efi" \
        "wicked: net0: Request to acquire DHCPv4 lease with UUID"
    get_duration_between "Login-Prompt" \
        "mount is: mount -t squashfs -o loop /tmp/cps/rootfs /new_root/lower" \
        " login:"
} >>"$this_csv_file"
```

In the above example, BIOS timing is measured from when “PeiCore.Entry” appears and stops when “NBP filename is ipxe.efi” is displayed.

These durations are rolled up to minutes and stored in a separate CSV file per node and later aggregated. The following is an example of the output.

The following image is a random example from the middle of the optimization effort. It is not reflective of the final result.

Figure 3.1.2 - Resulting reboot stage metrics in minutes

```
BOS_Session_Id,Test_Type,Stage,Minutes
b13.11-new-windom,New_Windom,Shutdown,4.18
b13.11-new-windom,New_Windom,BIOS,1.81
b13.11-new-windom,New_Windom,DVS_generate_node_map,.16
b13.11-new-windom,New_Windom,Dracut,1.00
b13.11-new-windom,New_Windom,iPXE,1.68
b13.11-new-windom,New_Windom,Login-Prompt,1.00
b13.11-new-windom,New_Windom,Total_Preboot,5.50
```

From the image above, it shows that shutdown took 4.18 minutes and boot from POST to user login prompt took 5.5 minutes.

3.2 Run-Time CFS/Ansible Logs

After the node boots to the OS, systemd runs the services that are included in the image. As part of that, CFS-state-reporter runs prompting CFS to run personalization configurations. BOS tracks the status of this as part of its session handling, meaning a BOS session does not complete until the CFS session and any retries have completed. Additionally, CFS is configured to retry up to three times by default in the event of a failure.

The following is an example command showing how to retrieve the CFS Ansible logs after the personalization stage completes.

Example 3.2.1

```
# Retrieve the CFS session ids from the BOS
session detail

Cfs_jobs=$(cray bos session describe
[session_id] --format json | jq -r
‘.status.session.job’)

# For each CFS session, retrieve the Ansible log

For cfs_job in $Cfs_jobs; do
kubectl -n services logs job/$(cray cfs sessions
describe batcher-123456-789 -format json | jq -r
’.status.session.job’)) -c ansible >
results/”$cfs_job”.log
done
```

Fortunately, Ansible provides a fair amount of profiling data in these logs. Logs are parsed using regex expressions and exported to CSV files resulting in two files per CFS session, one for all of the task times for a very detailed view, and another of the rollup playbook times.

The following function is an example parser for CFS logs, written in Bash.

Figure 3.2.1 - Ansible playbook and task time parser

```
# Accepts a cfs log file and outputs a csv file with the following columns:
function playbook_timing_csv() {
  local this_cfs_log="$1"
  local this_test_type="$2"
  local this_bos_session_id="$3"
  local this_csv_file="${this_cfs_log}_playbooks.csv"
  echo "BOS_Session_Id,Test_Type,Playbook,Repo,Seconds" >"$this_csv_file"
  cat "$this_cfs_log" |
    grep -E "Running|total" |
    sed 's|Running ||g' |
    sed 's| from repo |,|g' |
    sed 's|total [-]*||g' |
    tr -d '\n' |
    sed 's|https://api-gw-service-nmn.local/vcs/cray/||g' |
    sed -r "s| ([0-9]*\.[0-9]*)s|,\1\n|g" |
    sed 's|.git||g' |
    sed -r "s|(.*)|$this_bos_session_id,$this_test_type,\1|g" >>"$this_csv_file"
}
```

The results look like:

Figure 3.2.2 - Ansible playbook execution times in seconds

```
BOS_Session_Id,Test_Type,Playbook,Repo,Seconds
b13.11-new-antero,New_Antero,shs_cassini_install.yml,slingshot-host-software-
config-management,1.34
b13.11-new-antero,New_Antero,cos-compute.yml,cos-config-management,78.43
b13.11-new-antero,New_Antero,sma-ldms-compute.yml,sma-config-management,37.31
b13.11-new-antero,New_Antero,cos-compute-last.yml,cos-config-management,6.85
b13.11-new-antero,New_Antero,site.yml,custo-post-boot,20.86
b13.11-new-antero,New_Antero,site.yml,mo-config-management,74.25
```

3.3 Job Scheduler Status

The timestamp of the node registration for job duty is retrieved directly from the job scheduler.

The following example is how to do that with PBS Pro.

Example 3.3.1

```
pbsnodes -v “$this_node” -F json | jq -r
“.nodes.${this_node}.last_state_change_time”
```

Using Slurm, a similar command could be used.

Example 3.3.2
```
scontrol show node “$this_node” --json | jq -r
'.nodes[0].slurmd_start_time'
```

3.4 Aggregate Metrics

To simplify visibility, rollup metrics are further parsed from the results to provide a higher-level summary.

Figure 3.4.1 - Rollup reboot metrics

```
BOS_Session_Id              b13.11-new-windom
Test_Type                   New_Windom
Start                       2024-02-27T00:27:34
Stop                        2024-02-27T00:43:03
Minutes                     15.48
Preboot_Minutes             5.66
Num_CFS_Sessions            1
CFS_Sessions_Minutes        ;5.21
CFS_Sessions_Start_Times    ;2024-02-27T00:37:50
CFS_Sessions_Stop_Times     ;2024-02-27T00:43:03
```

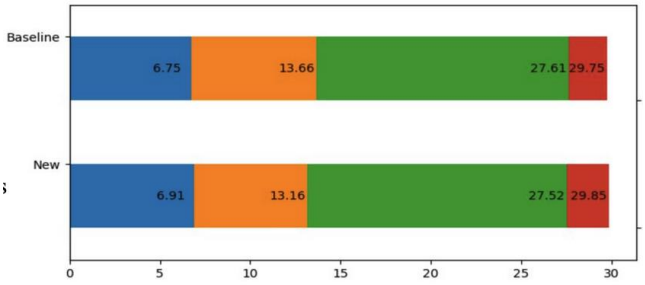
In this example, the full reboot time was 15.48 minutes from the time the reboot BOS command was issued to the time the node reported back into the job scheduler for duty. The underlying detail is also available. Given pre-os boot time was 5.66 minutes and the CFS configuration post-os was 5.21, it is reasonably inferred that shutdown was 4.81 minutes which is confirmed by looking at the pre-os csv file.

3.5 Notebook Dashboards

Lastly, a Jupyter notebook instruments the metrics to show the detailed breakdown of all stages of boot, including the execution time of each CFS Ansible layer and early substages of the reboot.

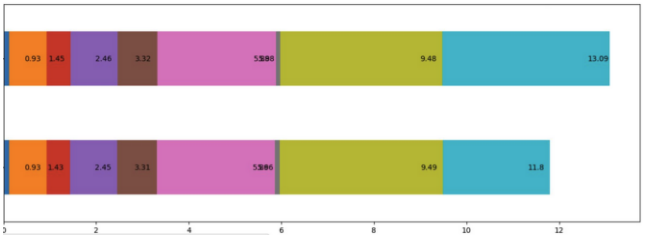
Here’s an example from very early in the boot optimization effort showing approximately 30-minute reboot times.

Figure 3.5.1 - Graphs comparing reboot times of two nodes. In this specific example the experiment yielded no improvement.



This example, again, from early in the optimization effort, shows CFS / Ansible execution times. Each color is a different layer or Ansible playbook, which all together, took more than 13 minutes to execute.

Figure 3.5.2 - Graphs highlighting execution times of each layer in the CFS personalization session. In this particular example, the experiment yielded an improvement of approximately 90 seconds.



4.0 Remediations

Armed with the data from numerous reboots, the following conclusions were drawn

4.0.1 **Shutdown** times vary, 2-10 minutes, because of the shared filesystem data being written back to the source as the job is terminated. This is considered as part of the job and not part of the reboot. This can be further confirmed by recording when the node unregisters from the scheduler.

As a test control, a BOS tuning is applied. The BOS option “max-power-off-wait-time" to 120 seconds where BOS will send a force reboot to the BMC if the OS isn’t down yet.

4.0.2 **Hardware POST** is slower on nodes with more components, from 2 to 4.5 minutes on the Cray EX hardware present during these efforts. Updating firmware provides options to decrease the console verbosity and therefore speed up the process.

4.0.3 **iPXE NIC Selection** can take up to a minute to select the right MAC address and send to the bootscript service for identification and selection of the appropriate boot configuration. Typically, the first onboard NIC is tried, and then the second before moving onto each expansion card network port. Between 10 to 60 seconds can be saved by modifying the iPXE script to detect the machine type being booted and start with the correct NIC before cycling through to others.

4.0.3 **System management** service batch requests into delayed chunks for power off and run-time configuration execution. Fortunately, these values can be tuned downward without issue at this scale, saving about a minute of delay.

4.1 Run-Time Configuration

Run-Time configuration is, by far, the largest opportunity to optimize reboot timing. This is reduced in three primary ways.

4.1.1 Move install activity into the image build.

For the most part, this was a matter of flipping the boolean in the entrypoint of the playbook and providing a default response when the variable is missing. This is the recommended approach for installing packages or providing configuration that is not a secret credential. The following is an example of what this looks like as of CSM 1.4.1.

Example 4.1.1
when: cray_cfs_image | default(false) | bool

Additionally, there are many install actions that end by trying to start a systemd service. Instead, enable the service since systemd is not actively running in the chroot environment of an image build.

4.1.2 Move run-time configuration not requiring protected secrets into systemd units. A new playbook is created for this which rsyncs the playbook files located in a CFS pod, into the image, creates a local ansible configuration and then wraps each playbook entrypoint in a systemd unit. Dependencies are defined within the systemd units where necessary.

Below is a portion of this playbook. The full repo is here(1).

Figure 4.1.1 - Snippet from an Ansible playbook which installs Ansible and copies in all playbooks into the image during build.

```
---
- name: Install Ansible
  ansible.builtin.package:
    name: ansible
    state: present
- name: Make local playbooks directory
  ansible.builtin.file:
    path: '{{ playbooks_dir }}'
    state: directory
    mode: '0640'
- name: Make systemd init script directory
  ansible.builtin.file:
    path: '{{ personalization_init_dir }}'
    state: directory
    mode: '0640'
- name: Rsync all playbooks to IMS host
  shell: |
    set -euo pipefail
    IMS_HOST=$(grep "ansible_host" /inventory/hosts/01-cfs-generated.yaml | se
    rsync -avz /inventory/ -e "ssh -i /etc/ansible/ssh/id_image" "$IMS_HOST":'
  args:
    creates: '{{ playbooks_dir }}/layer0'
  delegate_to: localhost
```

Dependencies are defined in the vars.yaml file of the playbook and used to parameterize the configuration of dependencies in each systemd unit.

Figure 4.1.2 - Ansible snippet from a playbook showing how systemd units for each CFS layer are enabled and ordered where appropriate or necessary.

```
  analytics-config-management:
    playbook: site.yml
    enabled: true
  pbs-config-management:
    playbook: site.yml
    enabled: true
    dependencies:
      - personalization_custo-post-boot_ansible
      - pre-pbs-start-nhc
  cos-config-manage:
    playbook: cos-compute-last.yml
    enabled: false
  custo-post-boot:
    playbook: site.yml
    enabled: true
    dependencies:
      - personalization_slingshot-host-software-config-management
      - personalization_cpe-config-management_ansible
```

As a result, we save about 18 minutes of time. Ansible configuration runs very early in the Linux boot process instead of a minute afterward via CFS. Additionally, each task saves seconds of ssh packaging overhead as they run locally instead.

4.1.3 To ease troubleshooting we have CFS personalization check the status of systemd units to continue centralizing the visibility of configuration execution status. If the systemd unit fails, the CFS session will fail.

Figure 4.1.3 - Ansible snippet showing how CFS personalization will check for a successful localized systemd orchestrated CFS layer and fail CFS if systemd fails.

```
---
- name: Get local personalization systemd status for PBS config
  ansible.builtin.systemd:
    name: personalization_pbs-config-management_ansible.service
  register: personalization_pbs_config_service
  until: personalization_pbs_config_service.status.ActiveState == 'active'
  retries: 12
  delay: 10
  no_log: true
- name: Fail if the personalization systemd status for pbs-config is not active.
  ansible.builtin.assert:
    that: [personalization_pbs_config_service.status.ActiveState == 'active']
    fail_msg: Systemd personalization_pbs-config-management_service must be 'active'.
      Check the log on the node for details, e.g. journalctl -xeu personalization_pbs
    success_msg: Systemd personalization_pbs-config-management_service is 'active'.
```

Again, the link to the Github repository is in the list of references at the bottom.

5.0 Final Results

The final acceptance test plan required testing one of each hardware type across four different environments. When the effort started, the average time of a successful boot was around 35 minutes.

The following table shows the 16 individual node reboot tests conducted during system integration acceptance testing (SIAT).

Table 5.0.1 - Individual Node Reboot Times

System	Node 1	Node 2	Node 3	Node 4
Quad A	10.40	10.23	10.26	10.68
Quad B	12.53	12.30	11.11	10.93
Quad C	10.46	10.45	11.48	11.46
Quad D	11.65	12.28	10.00	10.40

The mean of these results is 10.4 minutes.

Below is the comparison from start to finish.

Table 5.0.2 - Final Comparative Results

Stage	**Before – 12.2023**	**After – 07.2024**
Shutdown	8.9	2.3
Boot	6.2	5.5
Configuration	20.4	2.6

Total	35.5	10.4

Calculating from the new value of 10.4 minutes, **there is a 341% reduction**, e.g. 35.5 / 10.4 = 3.41.

6.0 Future Considerations

In the future, boot times can be reduced further by implementing a few experimental ideas

- **Kexec instead of hardware reboot**
  Kexec is a Linux feature that allows you to load and boot into a new kernel directly from the currently running system without going through the BIOS or firmware reboot process. Implementing it could save 4-10 minutes on hardware POST activity. Currently, Slingshot drivers do not support it, but this may change.

- **Run-time configuration becomes containerized or virtualized**
  Another option is to boot compute nodes more permanently into hypervisor-based or container serving operating systems. Then, deploy job execution environments as virtual appliances or in containers. When the job completes, these resources can be stopped without rebooting. This has the potential to save 2+ minutes of Ansible or bash execution during boot. It can also provide a measure of security, isolating secrets and protected configuration into Linux namespaces.

With the above innovations, reboot times could theoretically come down to less than five minutes.

ACKNOWLEDGMENTS

ChatGPT was used, as a convenience, to expand verbiage in some sections for better flow and clarity. Every point provided was originally initially written by the authors. Additionally, Grammarly was used to correct grammar and revise some words and phrases for enhanced brevity and clarity.

Thank you to Larry Kaplan, Harold Longley, and Isa Wazirzada at HPE for their reviews and feedback of early drafts.

REFERENCES

[1] What is DevOps? https://education.hpe.com/ww/en/training/docs/cds/h0ds6s.pdf