Use of Continuous Integration Tools for Application Performance Monitoring

Verónica G. Vergara Larrea, Wayne Joubert and Chris Fuson
Oak Ridge Leadership Computing Facility
Oak Ridge National Laboratory
Oak Ridge, TN, USA
Email: {vergaravg,joubert,fusoncb} at ornl.gov

***Abstract*—High performance computing systems are becoming increasingly complex, both in node architecture and in the multiple layers of software stack required to compile and run applications. As a consequence, the likelihood is increasing for application performance regressions to occur as a result of routine upgrades of system software components which interact in complex ways. The purpose of this study is to evaluate the effectiveness of continuous integration tools for application performance monitoring on HPC systems. In addition, this paper also describes a prototype system for application performance monitoring based on Jenkins, a Java-based continuous integration tool. The monitoring system described leverages several features in Jenkins to track application performance results over time. Preliminary results and lessons learned from monitoring applications on Cray systems at the Oak Ridge Leadership Computing Facility are presented.**

***Keywords*-continuous integration; application monitoring; supercomputers**

I. Introduction

High performance computing (HPC) systems are growing in complexity. New types of hardware components continue to be added, such as computational accelerators and non-volatile random-access memory (NVRAM). The software to support this additional complexity is also growing. These hardware and software components can interact in complex ways that are not entirely orthogonal in their coupling. For example, an upgraded library could have a higher memory requirement due to new algorithms which could in turn impact the amount of memory available for message passing interface (MPI) communication buffers. Applications themselves are also complex and are being combined into intricate workflows. Resource allocations on large HPC systems are costly and continue to be competitively awarded; users' ability to accomplish their research goals is negatively impacted when system performance unexpectedly degrades. The causes of such degradations are not always obvious; large, complex HPC systems often lack transparency to allow users and center staff the ability to understand what issues are impacting performance. Furthermore, while systems become larger and more complex, center staff workforce typically remains a nearly-fixed resource; this points to the need for automated means of monitoring system performance.

At the system administration level, tools are currently available to evaluate system health when such action is needed, for example, after a reboot [1]. However, from the user standpoint, the highest priority is to validate the reliability and performance of the system as experienced by user applications. Of paramount importance to users is that applications run reliably, accurately, and efficiently. To test this accurately, there is no substitute for running the applications themselves under realistic test conditions. Additionally, it is important to test auxiliary aspects of the system which directly impact the user experience. These include performance characteristics of software such as system libraries, system performance characteristics that can be tested using kernel benchmarks, and system configuration issues such as default modules which impact the code development experience of users.

In the past, efforts have been made to monitor application performance over time on HPC systems, notably with the NERSC Sustained System Performance metric [2] and the Department of Defense's sustained systems performance monitoring effort [3]. Similarly, the goal of the present work is to select a small set of application codes and tests to run periodically over time, e.g., daily, weekly, monthly or quarterly, to detect performance variances.

Whether for the purpose of application performance monitoring over time, or for the similar task of acceptance testing of new systems, it is not uncommon practice for each center to develop in-house tools customized to the specific requirements of the relevant task (see, e.g., [4]). In recent years, however, HPC systems have leveraged not only commodity hardware but also software tools used by the broader software community. Continuous integration (CI) is an increasingly practiced methodology for software development and delivery. To support CI, multiple GUI-based tools have been developed which have been found effective for managing periodic or on-demand software building, testing and reporting of results. Though targeted to a slightly different workflow than HPC application-centric system health monitoring, these tools offer potential promise for improving productivity for the latter workflows. They also inherit the advantages of off-the-shelf software. Compared to in-house solutions, they are more likely to have a familiar interface that is easily learned by incoming center staff. Furthermore, if well-supported by third parties or the community, there is potentially less need for center staff commitment to support these tools for development, maintenance, debugging, and porting.

The purpose of this paper is to examine popular CI tools to evaluate their appropriateness for use in monitoring application performance over time on HPC systems. The primary focus here is Jenkins [5], for which we describe the implementation of a monitoring system as well as present results and experiences from using this tool.

The remainder of this paper is organized as follows. First, we describe the requirements for an application performance monitoring tool. We then describe the tools evaluated. As a result of this evaluation, we selected Jenkins to use for implementing the application performance monitoring tool. In the following section, we describe the specific implementation of this system and then present results from use of this tool on the Oak Ridge National Laboratory (ORNL) Titan system. As an alternative, we also examine briefly the Splunk data monitoring and analysis tool [7], which does not satisfy all project requirements but has strengths for the reporting component of the targeted workflow. Finally, we present discussion, lessons learned and conclusions.

II. Requirements

The overall objective of this work is to deploy a tool usable by center personnel to monitor the performance of applications and kernels, and validate system configuration over time on an HPC system. This objective can be clarified and made more specific by enumerating a set of user stories to illustrate how such a tool might be used on a routine basis:

- As a center staff member, I want to be able to view the behavior and performance of each of a set of applications over time, so that I can determine quickly whether there has been a performance variance.
- If a variance is observed, I would like to diagnose its cause, e.g., by examining the run logs or correlating the event in time with a known system upgrade, so that remedial action can be taken.
- I would like to easily start, stop, or modify the periodic build and run process for an application (including management of how the tool is run across system reboots) so that the level of reporting can be maintained or adjusted.
- I would like to be able to easily modify existing build/run configurations, e.g., change test cases or core counts for runs, and add new applications to the tool, so that the coverage of system behaviors that are being tested can be improved.
- I would like to be notified, e.g., via email or text message, when a high severity test failure occurs, so that immediate action can be taken.

Needless to say, the tool should make it possible for a user to execute these workflow tasks in a way that is optimal from the standpoint of usability, so that minimal time and effort are needed by a user employing the tool on a routine basis during day-to-day operations. The well-known principles of usability engineering are apropos in this context (see, e.g., [8]), leading to design objectives such as minimizing the number of keystrokes or mouse clicks required to accomplish a given task.

With this backdrop, we now enumerate specific requirements. These fall under two categories: requirements for the tool itself as software, and requirements for how the tool must function as part of the application monitoring workflow described above. It is clear that the continuous integration workflow does not exactly match the HPC system application performance monitoring workflow: for the former, the application code itself is the primary point of failure to be tested, whereas with the latter, the unexpected performance variances from the system software stack and hardware are primarily in focus, leading to different failure response actions. Because of this less-than-perfect match, it is not to be expected that CI tools will necessarily be deeply customized to the requirements for an HPC system application performance monitoring tool. Thus, it may be that all requirements are not met entirely, or that some customization of the tool may be needed, via additional code, use of or development of plugins, or modification of the tool source code itself. In each case, an evaluation must be made to determine whether it is better to use the tool under these conditions or write a better-customized tool entirely from scratch.

In order to evaluate the available continuous integration tools and select a tool for developing the prototype, we created the following list of features that describes the ideal tool's match of the application monitoring workflow:

- INTERFACE: provide a graphical user interface (GUI), with support for command line interface (CLI) as needed.
- JOB MANAGEMENT: allow full job control via the GUI and provide a mechanism to execute jobs on-demand and periodically, with capacity for organizing and grouping tests by objective or by system.
- VISIBILITY: provide an interactive dashboard to view and analyze results in big-picture form with the ability to drill down into specific job results, including run artifacts from a job.
- ANALYSIS: provide highly configurable analysis and plotting capabilities, e.g., regarding status of jobs or custom metrics related to application performance.
- REPORTING: enable generation of reports and graphs in formats suitable for end uses, e.g., presentations or web pages, with flexibility of report style and format.
- NOTIFICATION: allow ability to send customizable notifications about job status or failure conditions via email, text message or instant messaging.
- AVAILABILITY: permit multiple users to access the tool in compliance with center security policies, with

ability to launch jobs on remote systems without requiring multiple authentications and to interact with native HPC scheduler software.
- ARCHIVING: allow storage and easy access of past jobs for later analysis.
- RESILIENCE: perform robustly in case of job or system failures, with capability for restarting.

From a software standpoint, we identified the following requirements to be necessary for the tool that is selected as the basis of the monitoring system:

- AVAILABILITY: should be freely available and open source.
- MAINTENANCE: must be a maintained software project, with a release in the last year or recent changes and bug fixes.
- SUPPORT: should be well-supported, have a large user community, and have an active development team behind it.
- FLEXIBILITY: must be flexible and extensible, e.g., through a plugin API or programmable interface.
- PORTABILITY: must be portable and usable for different targeted systems.
- DEPLOYMENT: should possess easy installation and deployment mechanisms.
- SUITABILITY: should require a minimal amount of customization to meet workflow requirements.

In the next section we present the results from the evaluation performed based on these requirements.

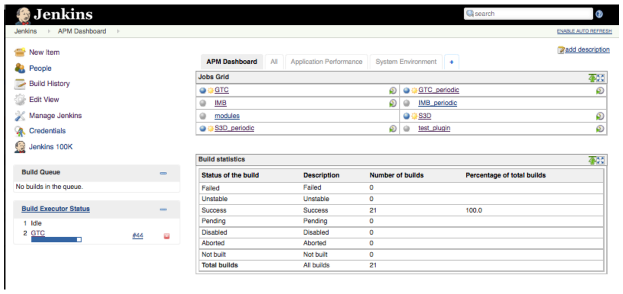
III. Tools Evaluated

Over thirty continuous integration tools are available today [9]. This work has focused primarily on open source continuous integration tools for two reasons. First, open source CI tools are often available at no cost, which allows us to test the tool without requiring a financial investment. Additionally, their open source nature gives us access to the entire code base, which in turn gives us the opportunity to customize the tool if this becomes necessary.

For this work we studied several open source CI tools including: Buildbot, Continuum, CruiseControl, Go, Jenkins, and Travis CI. In addition, we briefly reviewed commercial CI tools like Bamboo and TeamCity, and explored a few of tools that, although not considered CI tools, have interesting monitoring capabilities including CDash, Codespeed, Inca, XDMoD, and Splunk.

A. *Jenkins*

Jenkins [5] is one of the most popular and flexible open source continuous integration servers available today. It has a community of over 100,000 users and has over one thousand plugins available [6]. Jenkins is a Java-based tool capable of automating software building and testing processes as well as monitoring external jobs. One advantage of Jenkins is that



No builds in the queue.

1 Idle
2 GTC #44

Jobs Grid

GTC	GTC_periodic
IMB	IMB_periodic
modules	S3D
S3D_periodic	test_plugin

Build statistics

Status of the build	Description	Number of builds	Percentage of total builds
Failed	Failed	0	
Unstable	Unstable	0	
Success	Success	21	100.0
Pending	Pending	0	
Disabled	Disabled	0	
Aborted	Aborted	0	
Not built	Not built	0	
Total builds	All builds	21	

Figure 1. Jenkins Dashboard View

it has a dashboard (see Fig. 1 that can be used to fully control project and test creation, modification, scheduling and reporting. Although Jenkins does provide several plugins to plot test results, namely the Plot and the Measurements Plot plugins, the plots are not highly customizable. Jenkins also provides customizable user notifications of hazard events via email. Additionally, Jenkins is easy to install and start using without a great deal of prior configuration for user jobs.

B. *Buildbot*

Buildbot [10] is a flexible continuous integration framework used to automate the process of building, testing and releasing software. Implemented in Twisted Python, the Buildbot framework provides a set of components that can be used to easily fit complex workflows. Buildbot uses a master-slave paradigm to schedule and execute jobs, and, as most CI tools, it supports integration with version control systems. Jobs in Buildbot are configured via Python scripts and can be tracked via the Buildbot dashboard. Although Buildbot has more powerful building blocks and has customizable source code, there are fewer plugins ready out-of-the-box to support our target workflow. In addition, the dashboard in Buildbot is primarily used as a reporting tool rather than a job configuration control system. Though the source code of the tool is readily customizable, significant manual scripting work is required to implement tests.

C. *Continuum*

Continuum [12] is a fully equipped CI server developed by the Apache foundation. As most CI tools, Continuum is capable of automating builds and managing the code deployment process. Continuum also provides a dashboard that can be used to launch new jobs and view their status. However, Continuum's dashboard does not provide full job management features and reporting is limited to a table format.

D. *CruiseControl*

CruiseControl [13] is a freely available Java-based framework for continuous build processes. It provides a visualization dashboard with limited job control, and standard email notifications. Unfortunately, the latest available release is from 2010, and it only mentions dozens of plugins. In addition, reporting is fairly limited and plotting capabilities are not available.

E. Go

Go [14] is a formerly commercial continuous integration and release management tool that was recently open-sourced. Go's interface is very polished and is capable of modeling a variety of workflows. Unlike other CI tools, Go uses the concepts of pipelines to build different test and delivery systems. Even though Go's capabilities appear advanced in comparison to other open source CI tools, there are only a few plugins available.

F. Travis CI

Travis CI [16] is a powerful hosted CI service that is available for free to open source projects and is fully integrated with GitHub. It supports a wide range of languages and can be set up in minutes. Though Travis CI primarily targets web applications, its notification features and dashboard capabilities could potentially fit our targeted workflow well. In addition, Travis CI has a paid enterprise version that reportedly allows for local deployment of the tool. Because the freely available version is exclusively a hosted service, it would not fit our requirements.

G. Commercial Tools

Commercial tools such as Bamboo [11], and TeamCity [15] meet many or most of the requirements needed for the application monitoring system workflow. However, TeamCity and Bamboo allow only a limited number of builds and agents with their free and low cost plans. In addition, because our application monitoring system workflow is not equivalent to the CI workflow, there are concerns about the potential inability to extend closed source tools to adapt to the targeted workflow.

H. Non-CI Tools

We also considered several other types of tools such as system administration monitoring systems like Splunk [7], software testing servers like CDash, workflow tools like FireWorks, grid monitoring tools like Inca, operational metrics monitoring tools like XDMoD, and application performance management tools such as New Relic APM and Codespeed. Though similar in spirit to the targeted usage, these tools were on initial assessment not ranked to be of high interest since they did not closely match the targeted requirements.

1) Splunk: Splunk [7] is a log monitoring tool with robust reporting and notification capabilities which derive from events parsed from system log files. Splunk also provides a highly-customized dashboard that allows multiple arbitrary charts and tables to be combined to fit the specific monitoring requirements. In addition, Splunk has a programming language for customization directly in the dashboard, a plugin API which can be used to further customize reports, and configurable email alerts to notify of specific events. One disadvantage for this tool is that the web interface does not provide job control. Being a commercial tool, the reporting and analysis capabilities are very robust, however, the free-edition has limited log indexing and no email alerts.

2) CDash: CDash [17] is a software testing server that is part of the CMake toolchain and is primarily used for viewing results produced by the CTest testing tool. CDash provides a customizable dashboard to display the status of different builds. In addition, CDash can be configured to send email alerts. The CDash dashboard, however, does not provide the ability to manage jobs via the web interface or create plots for user-defined application performance metrics and does not support a plugin API for extensibility.

3) FireWorks: FireWorks [18] is a Python-based open source software used to manage and automate scientific workflows. One major advantage of FireWorks is its support for several batch schedulers commonly used in HPC, including: PBS, SLURM, and IBM LoadLeveler. In addition, FireWorks provides a web interface that can be used to monitor jobs; however, full job management is not available, and job result reporting is limited to a table format.

4) Inca: Inca [19] is a grid monitoring tool that, unlike many system monitoring tools, runs at user-level. Inca was developed at the San Diego Supercomputer Center and it is widely used within eXtreme Science and Engineering Discovery Environment (XSEDE). Inca, however, relies on certificates for authentication and on agents and consumers to generate, collect, and report data to a central database.

5) XDMoD: XD Metrics on Demand (XDMoD) [20] is developed by the Center for Computational Research at the University at Buffalo. XDMoD is a tool designed to obtain operational metrics from HPC systems. It has been primarily used by XSEDE centers, although a lightweight version called Open XDMoD has recently been made available for campus HPC centers. XDMoD has a highly customizable dashboard and can report statistics from allocations, jobs, resources, among others. More recently features to track the performance of applications across centers were added, but is ability to run appears limited.

6) Codespeed: Codespeed [21] is a web application monitoring tool for code performance. It is used to monitor and analyze code performance, and render results in a web-based dashboard. Codespeed can only be used for monitoring and does not have job management capabilities.

7) New Relic APM: New Relic APM [22], [23] is an application performance monitoring service that allows developers to quickly identify performance bottlenecks in the processing pipeline. New Relic APM is targeted mainly towards web application development, though in combination with other New Relic products such as New Relic Servers, could make a powerful solution to monitor application performance. In addition, the analytics tools included in the New Relic line of products could provide a detailed picture of potential performance issues. However, it is designed for loosely coupled collections of servers rather than tightly coupled HPC systems and clusters, and its suitability for the latter is unclear.

IV. Implementation

The Jenkins CI tool was found to satisfy the main project requirements: job control and management, job reporting including graphs of arbitrary metrics, email notifications, easy inspection of run artifacts, free availability, good support, and extensibility with a plugin API and over a thousand plugins. In this section, we describe specifically how the application performance monitoring tool based on Jenkins was implemented.

The application performance monitoring tool developed here based on Jenkins has two components that are stored in two separate directories. First, the Jenkins code itself is installed and configured with several plugins, including the Plot plugin for rendering performance plots on the dashboard, the Parameterized Trigger plugin for better management of linked builds, the LDAP plugin for access control, the Dashboard View plugin for a full picture view of results, and the AnchorChain plugin for setting up quick links to the build artifacts. Secondly, a directory is set up with shell scripts used to interface Jenkins to the targeted tasks. The scripts directory is put under version control using Git. To track changes in Jenkins dashboard configurations, the configuration files are also put in the scripts directory and connected to Jenkins by symbolic link.

The tool is designed to perform two kinds of tests: execution of applications and benchmark kernels with reporting of failure status and performance metrics, and testing of system configuration characteristics to detect problems. Application and benchmark kernel testing can be done in two ways. First, it is possible to write scripts that are callable by Jenkins that perform the desired tasks directly. The following is the set of steps performed in a Jenkins build target to execute an instance of this case:

- A script is called to perform the application build. This includes a checkout of the code from its repository to the instance workspace directory, configure, and compile, returning a status code for success or failure
- A job submission script is called. This creates the job batch script to perform one or more application runs, submits the batch script, waits for completion, and copies run artifacts to a location visible to the Jenkins dashboard. Part of this process is to collect status codes, and, if desired, performance metrics, from each application run and place these in comma separated values (CSV) files that can be accessed by the Plot plugin.

Secondly, it is possible to exploit build and run procedures already prepared and available in other test harnesses. For this work, we made use of the Oak Ridge Leadership Computing Facility (OLCF) acceptance test harness [24], which contains scripts to build and run dozens of applications, benchmarks and test cases. An interface layer was written to provide access to these tests from Jenkins. Additionally, each application in the acceptance harness was augmented with code to extract the targeted performance metrics to present to Jenkins.

In addition to applications and benchmarks, it is desirable to test basic system characteristics which impact the user. Several characteristics of the user environment can be tested, such as: filesystem mounts, environment variables, available modules and module definitions, default modules and versions, tools, and consistency of the environment across systems and login nodes. These tests have a temporal aspect, insofar as every Jenkins build instance targeting a test of this nature should compare against the results of previous builds to determine whether a change has occurred. Another aspect of particular importance is the need to test filesystems. This includes testing for center-wide filesystem availability, testing for performance, and testing the ability to transfer data between filesystems.

The OLCF provides multiple compute resources. These systems provide users with the ability to perform various tasks including, but not limited to, computation, pre- and post-processing, data transfer, and data analysis. It is important to test not only the center's flagship systems, but also, other user-facing systems. Because each system differs in hardware, configuration, and purpose, tests run across each system may also differ.

Managing and viewing reports from multiple locations is not ideal. Because of this, one goal is to have a single location from which tests can be launched and results can be monitored. For our testing, we created a virtual machine (VM) to host a single Jenkins instance. All tests are launched from the single Jenkins instance regardless of the test's target compute system.

When configuring Jenkins and the tests for the evaluation period, we had to work within center security policies. Two major challenges we encountered were two-factor authentication and shared filesystems. When investigating Jenkins we discovered a third-party plugin created by the BioUno group that allows Jenkins to interact with Portable Batch System (PBS) schedulers. The plugin, however, requires Jenkins to start at least one slave, which in our scenario is not possible without interactive authentication. To work within these limits, we chose to leverage existing center capabilities used to enable workflow between user-facing systems. Shared filesystems and the center-wide batch system with cross-platform submission capabilities were used to aid in access and communication between the Jenkins host and compute systems.

Center security policies require the Jenkins test host to use two-factor authentication when connecting via secure shell (SSH) to each compute system. However, the need for repeated two-factor authentication greatly hinders workflow automation. Because of this, we decided to leverage existing center capabilities that allow users to submit batch jobs from one user-facing compute system to another to work around the two-factor authentication limitation. Shared filesystems mounted on the compute systems and the Jenkins host are used to provide a common data storage location and remove the need to transfer data between the Jenkins host and compute systems.

OLCF compute platforms mount shared filesystems that can be categorized into two groups: an NFS mounted storage area with backup, and a 32PB Lustre scratch area. The NFS filesystem is mounted on both the Jenkins host and the compute systems, while the Lustre scratch area is only mounted on the compute systems. Because of its size and speed, the center-wide Lustre filesystem is used by many performance tests for job output, which in some cases can be much larger than will fit in the smaller NFS filesystem. In order to analyze and report test results from the Jenkins host, a subset of test output must be copied to the NFS filesystem available to both the Jenkins host and the compute systems.

To aid in workflows between the OLCF compute systems, a center-wide batch resource manager was created to allow batch job submissions between systems. A user can, for example, submit a batch job from the center's flagship compute system, Titan, to the center's analysis system, Rhea. As another example, a user can also submit a batch job directly from the Cray XC30, Eos, to the data transfer system. We were able to leverage this center-wide batch cross-submission capability to work within the two-factor authentication security requirement while allowing the Jenkins host to interact with the center's compute resources.

Using the center-wide batch system to access the compute systems means that all tasks that need to run on the compute systems must be submitted through the center-wide batch system. This includes not only parallel work for the compute system's compute nodes, but also other tasks that need to be executed on the systems login nodes such as compiling and environment verification.

Knowing the state of a submitted batch job is a useful key in a test's workflow. Tests that run regularly can, for example, check to see if the previous launch of the test still has a job in the queue. If so, the new test can be aborted to prevent multiple batch jobs for the same test from accumulating in the queue. Limiting the queue wait time for certain tests is also a useful example. If the test does not enter a running state within the time limit, the test is marked as a failure. Knowing the batch job completion time is another example that can be useful to tests that process results on the Jenkins host. Once a batch job completes, the process of analyzing and reporting the test results can begin.

To determine the state of a batch job, we can regularly execute batch system query tools like `qstat` (Torque/PBS) or `showq` (MOAB). This method is very straightforward and is the method used in many of our performance tests. Although we did not experience this during our testing, we do see the possibility of large numbers of queue checks overwhelming the batch system when run simultaneously. Because of this, we also tested another method to monitor batch job progress through the queue.

The second method uses files created at various stages of batch job execution to determine the state of the batch job. Three files are created and removed throughout a batch job's execution to denote batch job submission, execution, and completion. The submission file is created at job submission and removed after the job completes. It can be used to indicate that the test has a batch job in the queue or that errors occurred in the batch process. The file is often used as a lockfile to prevent a test from submitting a new batch job, if the test already has a batch job in the queue. Execution and completion files were used to place a wait and run time limit on certain tests.

In some test cases, it was useful to create files whose purpose was to provide information between various test stages or between related tests. For tests that run as separate but related chained tests through Jenkins, files in a specific known location in the test output tree can be used at later stages or by related tests to retrieve useful information about the run. The last successful output file location and name can, for example, be stored in a file and read at another time by the data processing stage of a test. Test specific files are one method we used to pass information between tests or from one stage within a test to another stage within the same test.

Jenkins performs each test as the user who started the daemon. The user who starts the Jenkins daemon on the Jenkins host owns the created test data and submits batch jobs. For our evaluation, we created a Jenkins specific user and started the Jenkins process as that user. Through the testing user's UNIX group we can limit data access to those in the testing team. By using a separate testing user, we are also able to control the batch priority of the Jenkins tests. This is a similar process to what has been successfully done with system acceptance work.

Without taking steps to limit access and ability, anyone who can connect to the Jenkins web GUI can also initiate tests and view data through the GUI. Because this is a security concern, we use the LDAP plugin to enforce the same two-factor authentication required to connect into an OLCF system. By using LDAP authentication, we can limit access to only OLCF users with specific attributes. For our testing purpose, only users on the testing team would have access. To further limit access to the test Jenkins instance, the test host sits in an enclave with very limited access.

V. Tests

To help us evaluate the CI workflow in a manner that resembles production, we chose test cases from those already being used to monitor OLCF systems. When selecting tests, we focused on those that would help monitor system usability and reliability from a user standpoint. We chose tests that help monitor two general areas: application performance and system environment.

Application performance monitoring makes use of known codes to monitor the system over time. By running codes for which we know historical and expected results, we can monitor a system for degradation or improvement. The codes chosen for the application monitoring workflow were selected from codes that were run as part of Titan acceptance. A subset of the codes is regularly used to test system changes.

We also included a category of tests to monitor user environment changes. Common everyday tasks often require a number of modules, tools, environment variables, and similar to work correctly. Changes to a module, for example, may prevent the ability to build a code. There are a number of pieces, large and small, that can impact the ability to build code, submit batch jobs, and perform other day-to-day tasks. Users should expect these pieces to be consistent over time. Although each OLCF system differs in many ways, each share a common set of OLCF tools, environment variables, and filesystem locations. It is important to verify and provide a consistent standard environment between systems. For systems with multiple login nodes, each node should have the same environment. Tests in the system environment category target finding changes that impact a system's general usability.

A module consistency check is one of the first environment tests added to the evaluation. The test is used to verify the consistency of user software environment modules across each of Titan's login nodes. Software environment modules are used to alter a user's environment to use a particular software package or library. Titan has sixteen login nodes organized into multiple functional groups. Each login node, regardless of group, should contain the same user modules. When user modules available on each login node differ, batch jobs can fail. To add to the complexity and possibility for confusion, users do not, under general circumstances, select or control which login nodes they use when connecting or running batch jobs. Because of this, we want to ensure that each login node contains the same user modules.

To verify that the list of available user modules across each login node is the same, the command `module avail` is executed on each login node and the outputs are compared for differences. The test, which is initiated from the Jenkins host system, submits a batch job to access one of Titan's internal login nodes. Once on a login node, the test is able to connect to each of Titan's login nodes via SSH and run the `module avail` command. The results are written to a filesystem that is visible from both Titan and the Jenkins test host. Upon batch job completion, the results are processed from the Jenkins test host. The number of modules, tested login nodes, and inconsistencies found are written to a CSV file. The Jenkins plot plugin reads the CSV file and creates two plots displaying the data. If the test returns a non-zero code at any point in the test, Jenkins considers the test a failure. The test is executed hourly or manually when faster response is needed, usually following a known system change.

Other possible environment test examples include tests to ensure batch job submissions follow OLCF batch policies, default modules have not changed over time, each system and login node contains common center tools and environment, common filesystems are mounted on each system and login node, and the high performance storage system (HPSS) is accessible from each system and login node.

VI. Results

In this section we present results from the two test classifications described above: application performance and user environment stability. The results shown include those obtained from monitoring the Titan supercomputer for several weeks at different intervals.

A. Application Performance Results

In the Jenkins prototype, we chose four different applications to track application performance over time. The first application is a simple vector operation known as DAXPY. The second one is a microbenchmark called Intel MPI Benchmarks (IMB) suite that is commonly used for testing HPC systems and clusters. The third is a production-quality combustion application called S3D. Finally, we also tracked the performance of a fusion plasma simulation code known as the gyrokinetic toroidal code (GTC).

1) DAXPY: DAXPY is a vector operation that computes $y = a \cdot x + y$, where a is a constant and x and y are vectors. We started with this application due to its simplicity and short runtime. The data collected from running DAXPY several times a day is shown in Fig. 2. The results page shows execution time and effective memory bandwidth from running DAXPY at different core counts for a single node of Titan.

2) IMB: The Intel MPI Benchmarks suite can be used to measure the performance of point-to-point and collective MPI operations over a range of message sizes. The IMB suite contains several benchmarks designed to characterize the efficiency of the particular MPI implementation on the system. Results from running IMB periodically via the Jenkins host are shown in Fig. 3. The graphs show the execution status code of the executed jobs over time as well as measured latency values for the ping-pong test for an arbitrary pair of nodes.

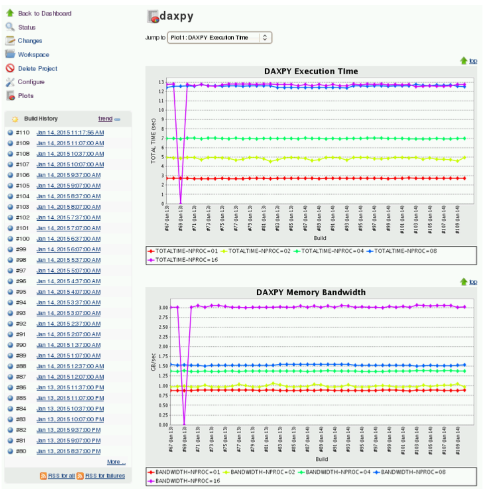


Figure 2. Jenkins test results for DAXPY routine

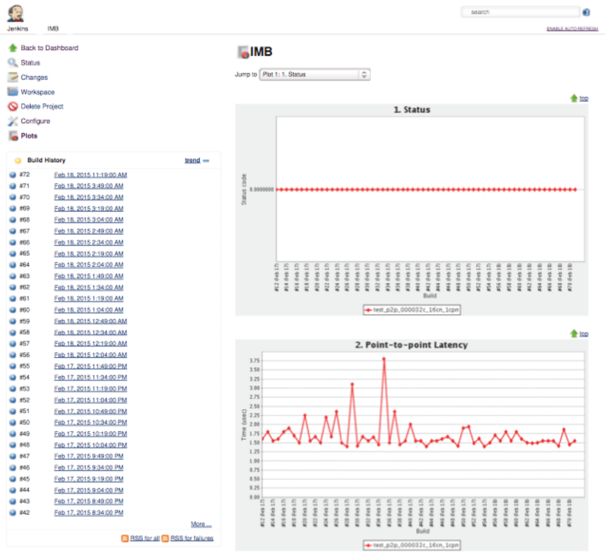


Figure 3. Jenkins test results for IMB benchmark

3) *S3D:* The S3D test is a combustion simulation taking approximately 20 minutes to run and requiring 38 Titan nodes. Because of its size, this test was run 4 times a day to track the variation in runtime as well as in IO bandwidth obtained when writing checkpoint files. As we can see on Fig. 4, even during the same test the IO rate during can vary significantly. This is an interesting finding that we will continue monitoring to see if any performance patterns are identified.

4) *GTC:* The GTC test uses 32 Titan nodes and on average runs for only a few seconds. Because this test is very short, small fluctuations in performance are expected. The plot shown in Fig. 5 clearly shows that at a specific time the write performance significantly decreased. We can use this information to further investigate the source of the

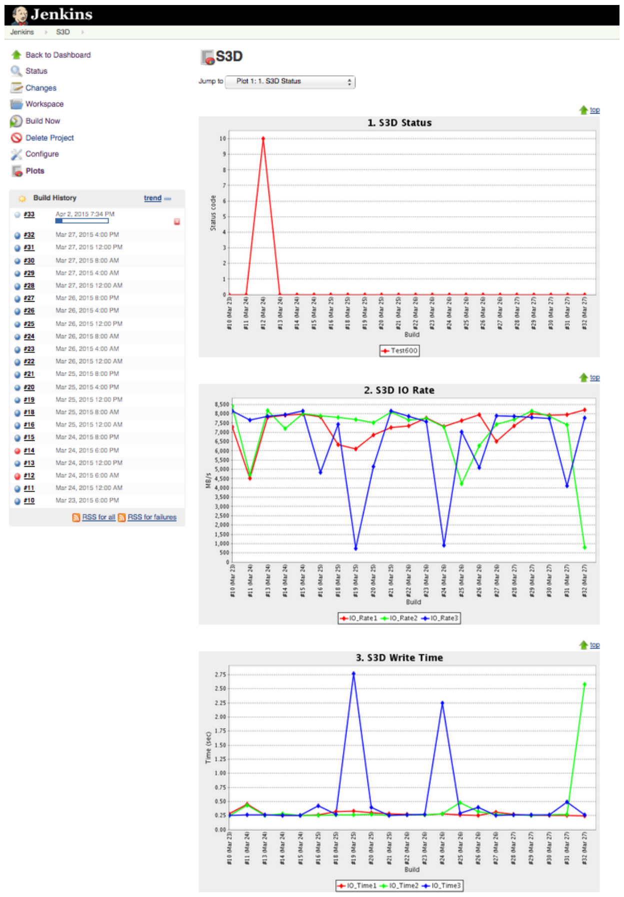


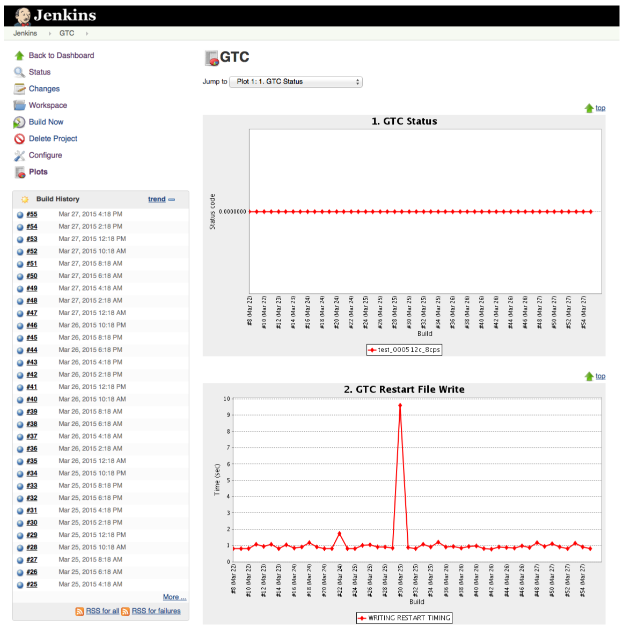
Figure 4. Jenkins test results for S3D

performance degradation.

The primary purpose for deploying these application tests is to monitor the performance of the interconnect and the Lustre filesystem over time. By automatically providing a graphical view of the behavior of the relevant metrics over time, the user is able to assess what is the expected typical behavior of the metrics. A qualitative change in this behavior will indicate to the user the need to investigate the root cause, such as a detrimental change to the system software. Having the graphs provided automatically within a dashboard is much more convenient and faster than manually copying the data into a spreadsheet for inspection.

B. User Environment Stability Results

As described earlier, this test is used to verify that each of Titan's login nodes contains the same set of modules. During our evaluation period the module consistency test was run every thirty minutes. The test successfully found and notified us of module differences before they impacted users in several instances. In one occasion, following an upgrade, the test found a difference in the available modules between some of Titan's login nodes. The finding follows



Build History	trend
#55	Mar 27, 2015 4:18 PM
#54	Mar 27, 2015 2:18 PM
#53	Mar 27, 2015 12:18 PM
#52	Mar 27, 2015 10:18 AM
#51	Mar 27, 2015 8:18 AM
#50	Mar 27, 2015 6:18 AM
#49	Mar 27, 2015 4:18 AM
#48	Mar 27, 2015 2:18 AM
#47	Mar 27, 2015 12:18 AM
#46	Mar 26, 2015 10:18 PM
#45	Mar 26, 2015 8:18 PM
#44	Mar 26, 2015 6:18 PM
#43	Mar 26, 2015 4:18 PM
#42	Mar 26, 2015 2:18 PM
#41	Mar 26, 2015 12:18 PM
#40	Mar 26, 2015 10:18 AM
#39	Mar 26, 2015 8:18 AM
#38	Mar 26, 2015 6:18 AM
#37	Mar 26, 2015 4:18 AM
#36	Mar 26, 2015 2:18 AM
#35	Mar 26, 2015 12:18 AM
#34	Mar 25, 2015 10:18 PM
#33	Mar 25, 2015 8:18 PM
#32	Mar 25, 2015 6:18 PM
#31	Mar 25, 2015 4:18 PM
#30	Mar 25, 2015 2:18 PM
#29	Mar 25, 2015 12:18 PM
#28	Mar 25, 2015 10:18 AM
#27	Mar 25, 2015 8:18 AM
#26	Mar 25, 2015 6:18 AM
#25	Mar 25, 2015 4:18 AM

More ...
RSS for all RSS for failures

Figure 5. Jenkins test results for GTC

the intended purpose of the test and allowed us to correct the difference before the issue impacted a user.

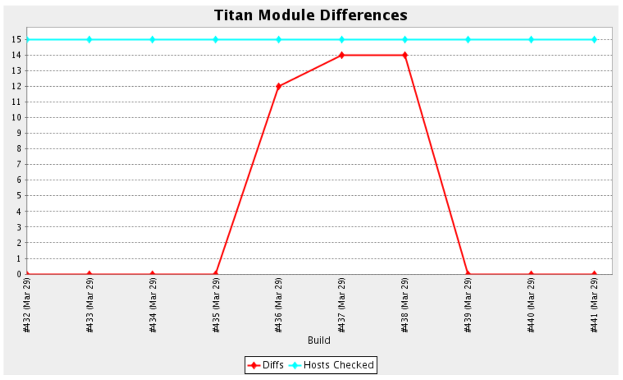
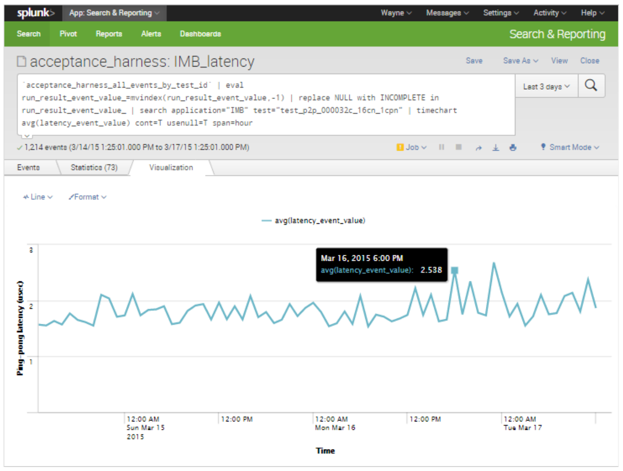


Figure 6. Jenkins test results from Module test

In addition, the test also notified us when a third party software installation was completed with incorrect permissions set. In this instance, the installed modulefiles did not have world read permissions which are necessary for users to access the installation via modules. When the test ran the command to retrieve the modules available, it reported the errors returned by the module command. This was an unintended finding for the test, but it notified us of an issue before users were impacted. If we had not been running the test at thirty minute intervals, the issue may not have been corrected until a user reported the issue. The report also indicates that another test could be created to verify the usability of common tools such as `module avail`. Jenkins can be used to link the two tests, in this case, only running the module consistency test if the `module avail` functionality test completes successfully. We consider this result a validation of the usefulness of the tool as a means to maintain system health as perceived by users.

C. An Alternative: Splunk

From our experience with Jenkins, it was felt that one of the major weaknesses of the tool pertained to its reporting capabilities, e.g., the limited ability to reformat graphs. Because of this, we also examined the Splunk log monitoring tool as an alternative. The Splunk data monitoring and analysis tool does not satisfy several of the previously described requirements: it does reporting and notification only, with no job management, and it is a proprietary tool with licensing fees. However, it is presented here as an alternative for use in conjunction with existing application harness software with job management capabilities, due to the flexibility of its reporting features.



```
`acceptance_harness_all_events_by_test_id` | eval
run_result_event_value_=mvindex(run_result_event_value,-1) | replace NULL with INCOMPLETE in
run_result_event_value_ | search application="IMB" test="test_p2p_000032c_16cn_1cpn" | timechart
avg(latency_event_value) cont=T usenull=T span=hour
```

✓ 1,214 events (3/14/15 1:25:01.000 PM to 3/17/15 1:25:01.000 PM)

Mar 16, 2015 6:00 PM
avg(latency_event_value): 2.538

Figure 7. Splunk test results for IMB benchmark

To use this tool, the OLCF acceptance harness software was modified to write a tagged event string to the system log via the Linux `logger` command for every significant event, such as start and end timestamps for application build, job submit and application run, as well as execution success or failure and, if desired, run performance metrics. Splunk is configured to sweep and index the system log files on a continuous basis. A search query is written by the user within the Splunk browser interface for the purpose of picking up these logs and presenting the results graphically over time. Multiple reports can be assembled on a dashboard page to enable visual monitoring of many tests at the same time. Fig. 7 shows an example report panel used to monitor the result of the IMB suite run periodically over a period of several days and displaying the result of the ping-pong latency test for arbitrary pairs of nodes on Titan. A test like this could be used to detect and notify center staff if a systematic decrease in MPI point-to-point messaging performance were to occur for any reason.

VII. Discussion

For the period of time spent on this project, we were only able to evaluate a small number of tools. Overall, our experience with these tools, namely Jenkins and Splunk, for use with this workflow was positive.

Generally, the Jenkins tool provided a positive usability experience. Jenkins is fairly easy to install out-of-the-box. In our scenario, however, some development was necessary to link the different pieces and plugins together. Alternative tools, on the other hand, require a more significant development effort up-front to prepare the testing system for specific jobs. Once the application monitoring infrastructure was in place, the Jenkins-based prototype allowed us to easily add, remove, start and stop tests via the web interface. It is worth noting that the web interface is extremely convenient and fairly straightforward to use, as it provides center staff good control of tests and reports.

During the deployment of the prototype, several challenges, specific to the security policies of the center, had to be addressed. First of all, security features in Jenkins were configured so that only a set of authenticated users could launch jobs. In addition, Jenkins was launched using a non-standard port on a separate virtual host accessible only from within the internal network. Given that the center requires two-factor authentication via RSA SecurID tokens, instead of allowing Jenkins to start remote slaves, we made use of the cross-submission capabilities available at the OLCF. This allows the Jenkins host to submit jobs to all of the compute resources available at the center, while keeping the monitoring system decoupled from them.

Plotting test results is an area where our attempts to use Jenkins for something it was not designed for shows. Plotting results is likely not very useful in a true continuous integration workflow. But, in our use-case, plots can be very useful. The Jenkins Plot plugin was used to create plots during our evaluation period. The plugin works to display basic charts, but it is limited to a small number of basic plots, is not very configurable, and is not well-documented. For some tests, we created a single CSV file in a manner that can be not only read by the Jenkins Plot plugin but also parsed and read by other reporting tools. We found better reporting and plotting capabilities using Splunk.

Another feature that we found to be limited in Jenkins was build management. While it is very easy to create different types of workflows using plugins in Jenkins, creating reports that combine results from different builds can be cumbersome. The reports and plots are fairly static and even minor changes such as labels are difficult to modify a posteriori.

One concern that arose during the evaluation period was that Jenkins' benefit would merely be that it was a replacement for the Linux `cron` command. Although this is true on some levels, Jenkins does provide additional functionality that has already proved beneficial. The dashboard, for example, allows us to quickly see the last state of each test and tells us when the last successful run and failure occurred for a given test—operations that in fact could be done at the command line but can be completed much more quickly through the dashboard. The ability to visualize test results, in one place, allows us to immediately see the "big picture". In addition to scheduling tests, Jenkins also provides the ability to chain test through dependencies.

For test creation, Jenkins does not remove the need to write tests nor does it aid the test creation process. However, Jenkins did not get in the way of test creation either. Because Jenkins does not require tests to use a specific framework or language, many of the tests we used during the evaluation period can run with little to no modifications outside of Jenkins. This makes importing existing tests considerably easier. The overhead for creating tests can also be reduced through the creation of scripts that handle common functions required by most tests. Most tests, for example, will interact with the center-wide batch scheduler. Providing a suite of in-house created scripts to handle common interactions with the batch system, can speed test creation and migration, as well as, help lower entry barriers for the development of new tests.

A test's standard output and error will be captured by Jenkins. The data is stored on the Jenkins host in the build's log directory. For some tests, sending all of a job's output to `stdout` and `stderr` is not ideal as it introduces the possibility to create very large log files. Depending on the amount of data written, the number of tests, and the frequency of tests, the log directories can become quite large. Data management for the log file directories should be taken into consideration. However, Jenkins build targets can also be written to schedule and run scripts that perform data management tasks, conveniently controlled through the dashboard.

The console link in the Jenkins GUI displays a test's standard output and standard error in real-time as the test runs. If a test fails, the test's standard output and standard error will be emailed to the test owner. When writing tests, the data sent to standard output and standard error and its format can be used to quickly see why the test failed and find out other basic information about the test. This was found to be helpful for quick triaging of failed cases.

Often when looking for third-party tools to perform behind-the-scenes functions like account management and system use reporting, we end up creating in-house solutions. In-house tools can be created to closely meet requirements and offer functionality not available through non-HPC third-party tools. However, in-house tool creation is costly, often requiring large amounts of staff resources and time. While Jenkins does not meet our testing needs as fully as a tool created specifically to test and monitor HPC systems in our center's configuration, it has enabled us to test and monitor our systems in a relatively short period of time and without the need for a dedicated staff development team.

Alternatively, the Splunk tool provided a different kind of functionality. The tool provides reporting and notification only, so an external tool such as a testing harness must be relied upon for job management—alternatively, Jenkins can be used for this function, with Splunk providing supplementary reporting. This lack of integration results in some drawbacks, e.g., there is no capability to click through to view run artifacts from the Splunk dashboard. This being said, the reporting function is much more flexible in Splunk compared to Jenkins, allowing the ability to configure graphs and charts easily and to put multiple panels on a dashboard, though it is not as flexible as, say, a spreadsheet program. Splunk provides its own on-dashboard programming language, making it it extremely configurable and very easy to create on-the-fly what-if queries to investigate issues that arise. However, the SPL Splunk programming language does have a learning curve for a new user, and also in having bash-like pipes and potentially complex regular expressions can lead to code that is difficult to read and maintain. As a software tool Splunk has an industrial-strength feel, due to being a well-supported commercial code. Nonetheless, we did encounter one known bug during this short exercise, so as with any large software system it is not flawless. Overall, we felt the primary strengths of Splunk were: (1) the ability to construct a customized dashboard for understanding the state of many tests quickly at-a-glance, and (2) the ability to construct custom queries to ask questions about the results quickly on-the-fly.

VIII. Conclusion

This project looks at continuous integration tools from a novel perspective. We attempt to leverage features commonly available in CI tools to monitor application performance and environment stability in HPC systems. As described earlier, we found that while many of the tools possess some of the features necessary to successfully monitor changes in an HPC system, Jenkins seemed to provide the best match for the desired functionality and requirements, though the application performance monitoring workflow lies outside the scope of standard CI tools per se. Furthermore, the Splunk tool, though primarily intended for log monitoring, provided useful additional functionality regarding reporting.

The evaluation provided here can be useful to other HPC centers interested in monitoring application performance and user environment stability. In particular, our experiences with Jenkins and Splunk can be beneficial to monitor systems that use Torque and MOAB for resource management and job control. Because OLCF requires two-factor authentication via an RSA SecurID token, unfortunately, we were unable to test BioUno's PBS plugin which requires passwordless secure shell connections to Jenkins' slaves. Centers where two-factor authentication is not required might benefit from investigating that plugin. Nonetheless, we were able to use Jenkins in conjunction with the PBS scheduler in a way that was not limited by authentication concerns.

One major advantage of CI tools like Jenkins is that they work out-of-the-box and have substantial built-in functionality, which can greatly reduce the amount of development and the amount of time needed before a monitoring system such as the one described here is deployed. Additionally, Jenkins provides the largest number of plugins and has a large user community, which are good indicators that support for the tool will continue into the foreseeable future.

Even during the relatively short time that the Jenkins prototype has been running, we have already been able to automatically identify inconsistencies in the environment, which has reduced the impact to our user community. In addition, the automated testing performed with the Jenkins prototype has allowed center staff to proactively monitor and correct issues instead of reacting to user reports. The prototype has served well as a monitoring tool for system environment stability. However, given the limited plotting and reporting capabilities in the Jenkins plugins, additional work would be needed to better extract, analyze, and store application performance over time. For this second purpose, a more robust reporting tool such as Splunk is in our opinion a better candidate.

Acknowledgment

The authors would like to thank Don Maxwell for his comments on system tools and Arnold Tharrington for his work on the acceptance harness which was a building block for the Jenkins prototype . This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] J. Sollom, "Cray's Node Health Checker: An Overview," *Proceedings of CUG 2011*, https://cug.org/5-publications/proceedings_attendee_lists/CUG11CD/pages/1-program/final_program/Monday/04B-Sollom-Paper.pdf.

[2] W. Kramer, J. Shalf, E. Strohmaier, "The NERSC Sustained System Performance (SSP) Metric," Paper LBNL-58868, Lawrence Berkeley National Laboratory.

[3] P. M. Bennett, "Sustained systems performance monitoring at the U.S. Department of Defense High Performance Computing Modernization Program," *SC'11, November 12-18, 2011, Seattle, WA, http://ieeexplore.ieee.org/xpl/abstractAuthors.jsp?arnumber=6114483.*

[4] G. A. Pedinici and J. K. Green, "SPOTlight On Testing: Stability, Performance and Operational Testing of LANL High Performance Compute Clusters," SC'11, State of Practice Reports, Article 25, http://dl.acm.org/citation.cfm?id=2063382.

[5] “Welcome to Jenkins CI,” http://jenkins-ci.org.

[6] “Jenkins Celebration Day is February 26,” http://jenkins-ci.org/content/jenkins-celebration-day-february-26.

[7] “Splunk,” http://splunk.com.

[8] “Usability engineering,” http://en.wikipedia.org/wiki/Usability_engineering.

[9] “Comparison of continuous integration software,” https://en.wikipedia.org/wiki/Comparison_of_continuous_integration_software.

[10] “Buildbot,” http://buildbot.net.

[11] “Bamboo,” https://www.atlassian.com/software/bamboo.

[12] “Continuum: Welcome to Apache Continuum,” https://continuum.apache.org/.

[13] “CruiseControl Home,” http://cruisecontrol.sourceforge.net.

[14] “Continuous Integration: Go CD,” http://www.go.cd.

[15] “TeamCity,” https://www.jetbrains.com/teamcity/.

[16] “Travis CI,” https://travis-ci.org.

[17] “CDash,” http://www.cdash.org.

[18] “Introduction to FireWorks,” https://pythonhosted.org/FireWorks/.

[19] “Inca: User Level Grid Monitoring,” http://inca.sdsc.edu.

[20] “XDMoD Portal,” https://xdmod.ccr.buffalo.edu/.

[21] “Tobami Codespeed,” https://github.com/tobami/codespeed.

[22] “New Relic Production Monitoring Solutions,” http://newrelic.com/solutions/production-monitoring.

[23] “Application Performance Management,” http://en.wikipedia.org/wiki/Application_performance_managementand.

[24] “New Test Harness Keeps OLCF Resources Running Smoothly,” https://www.olcf.ornl.gov/2015/02/10/new-test-harness-keeps-olcf-resources-running-smoothly/.