The Careful Puppet Master: Reducing risk and fortifying acceptance testing with Jenkins CI

Jason A. Smith, Gabriel Richman, John DeStefano, James Pryor, Tejas Rao,
William Strecker-Kellogg, Tony Wong

Brookhaven National Laboratory, PO Box 5000, Upton, NY 11973, USA
E-mail: smithj4@bnl.gov, grichman@oberlin.edu, jd@bnl.gov, pryja02@gmail.com,
raot@bnl.gov, willsk@bnl.gov, tony@bnl.gov

Abstract. Centralized configuration management, including the use of automation tools such as Puppet, can greatly increase provisioning speed and efficiency when configuring new systems or making changes to existing systems, reduce duplication of work, and improve automated processes. However, centralized management also brings with it a level of inherent risk: a single change in just one file can quickly be pushed out to thousands of computers and, if that change is not properly and thoroughly tested and contains an error, could result in catastrophic damage to many services, potentially bringing an entire computer facility offline.

Change management procedures can—and should—be formalized in order to prevent such accidents. However, like the configuration management process itself, if such procedures are not automated, they can be difficult to enforce strictly. Therefore, to reduce the risk of merging potentially harmful changes into our production Puppet environment, we have created an automated testing system, which includes the Jenkins CI tool, to manage our Puppet testing process. This system includes the proposed changes and runs Puppet on a pool of dozens of RedHat Enterprise Virtualization (RHEV) virtual machines (VMs) that replicate most of our important production services for the purpose of testing. This paper describes our automated test system and how it hooks into our production approval process for automatic acceptance testing. All pending changes that have been pushed to production must pass this validation process before they can be approved and merged into production.

1. Motivation

When we started using Puppet over four years ago, tools that are common to configuration management today, like r10k, didn't exist yet, so we created our own Git hooks and scripts to manage our Puppet master's modules and environments, and to link them to our custom Git repository, which contained our entire Puppet code catalog. We started with three main Puppet environments, development, testing & production, which later evolved to include dynamic & directory environments as Puppet added those features. We relied on manual testing of changes before they would get merged into the main production environment, and created a manual review & approval interface to do the actual merge into production. Even with the manual review & approval system, we occasionally had a few close calls where people would quickly and carelessly push & approve changes to production
without careful review or testing beforehand. The main problem here was that not all changes would get properly and thoroughly tested before they got merged into production. Small changes might not get tested at all, and large changes might only get tested by one or a few people on a limited set of systems. Relying on manual testing is prone to human error: mistakes can easily be made and not all possible variations will be tested. Therefore, we needed a way to implement some sort of minimal automated testing system with feedback before allowing changes to be merged into production, so it could catch at least some of the more obvious mistakes and problems.

2. Automated Testing Plan
Since the U.S. DOE (Department of Energy) runs a few summer intern programs for students, we decided to recruit a student from the SULI (Science Undergraduate Laboratory Internship) program to help us investigate and setup a prototype automated testing system. We started by looking at all of the common community standard testing systems available and choosing something that would integrate into our existing workflow and fit all of our other requirements. The existing tools that we considered were: RSpec, Serverspec, Beaker, Jenkins CI and Travis CI. These all fit into 3 basic categories of automated testing: unit, acceptance, and integration. The terminology comes from the software development community and, when applied to Puppet configuration code testing, the individual units are the Puppet modules.

   Unit tests, as the name implies, are tests usually written for the smallest functional unit of a project. For a software project, this would usually be an individual method or function, or perhaps a class. In Puppet, this unit would be treated as an individual module. Unit tests are usually the most useful when updating or re-factoring your code, as you want to make sure the changes don't break any existing functionality or introduce any regressions for which you've written specific tests.

   Acceptance testing is usually performed by running the code on a real-world system and examining the result to see whether it performed as expected. To keep debugging potential problems as simple as possible, just like unit testing, this is usually performed on individual units if possible given potential dependencies. As is the case with unit testing, doing this with Puppet usually means operating on individual modules, which would normally be initiated by a pre-commit hook to automate the testing of modules during the normal process of editing and committing changes to the code.

   Integration testing is done by combining individual modules and testing them as an integrated group. In Puppet, this would normally mean combining all of the necessary modules typically used on servers and testing their combined result. The main advantage is the discovery of conflicts and problems across modules that individual unit testing would not find. This form of testing is what interests us the most, especially if we are going to hook it into our existing Git production approval system, since pushed change sets could touch many modules. Therefore, we need a fully integrated testing system.

3. Testing Tool Options
We evaluated several testing tools that are common in the Puppet community and frequently used by many others.

3.1. RSpec
RSpec is a unit test framework that was originally developed for the Ruby programming language. Puppet has an RSpec add-on to facilitate writing unit spec tests for individual modules. It is very common with PuppetForge modules and is used by GitHub to automate the testing of pull requests. We didn't choose this tool for our purposes, because we wanted a more comprehensive test that could be used on the host level across multiple modules.
3.2. **Serverspec**[^2].  
Serverspec uses RSpec to perform acceptance tests of modules by actually running the module code on a computer. This test host will typically either be your local workstation or another host or VM to which it will connect remotely, or access through some remote API. Since it is also usually done on individual modules, it is another form of unit testing. It can perform basic acceptance tests like checking if the correct package was installed, the correct service is running, the correct service port is listening. Since we were looking for a more comprehensive integration testing system we didn't choose this tool either.

3.3. **Beaker**[^3].  
Beaker is another acceptance testing system, like Serverspec, that was created by PuppetLabs, which also incorporates provisioning of VMs to help automate the testing of Puppet modules. It supports several of the most common VM back-ends: VMWare, EC2, vSphere, Vagrant, GCE, Docker, Openstack, and Zones. The typical workflow used by Beaker is: provision the OS → validate the OS → configure with Puppet → test the resulting configuration → revert the OS image to pre-test state → cleanup & report results. Like the other test systems, it is typically used to run acceptance tests on individual modules. Also, since Beaker was still in the early development stage when we were looking at testing systems, we decided to keep looking for other alternatives.

3.4. **Jenkins CI**[^4].  
Jenkins CI is a continuous integration system that is used to automate the testing of software projects. It is usually used in a source code management system, like Git, with a commit or push hook that initiates a build/compile of the entire software project and then runs various tests. It is basically an elaborate job scheduling & management system with status reports and full historical data, and since it also provides many plugins to customize to specific needs, it was well suited to fit our requirements.

3.5. **Travis CI**[^5].  
Travis CI is another continuous integration system, but is provided as a hosted service instead of software that you can download and setup on your own internal infrastructure. It is normally a paid service, but it also offered for free to open source projects and commonly used by many of GitHub projects to automatically test pull requests.

4. **Automated Testing with Jenkins**
We chose to use the Jenkins CI tool because of its general job scheduling & management features, and plugins that allow it to be easily customized to fit into our existing production approval system. We setup a Jenkins master server and a few slave pool nodes to execute the necessary Puppet tests. This allowed us to test changes to Puppet modules at the whole host catalog level, and catch potential problems between modules that most of the other unit & acceptance testing wouldn't be able to detect. The Jenkins test results are verified by our production approval script: only changes that pass will be allowed to be merged into our production environment. This testing process is fully automated by Jenkins; the only manual step required is to populate the pool of slave test nodes with critical and important Puppet configurations using all of the necessary modules.

In addition to using Jenkins, we decided to use Foreman[^6] as the Puppet External Node Classifier (ENC) for the slave test pool, because we could automate the classification & environment settings using Foreman's REST API. Since we had an existing RHEV[^7] cluster, we also decided to use that to setup the Jenkins slave pool VMs, automating their management using the RHEV REST & shell APIs.

[^2]: Server Spec
[^3]: Beaker
[^4]: Jenkins CI
[^5]: Travis CI
[^6]: Foreman
[^7]: RHEV
4.1. Jenkins Configuration.
We modified our existing Git hooks and CGI approval script to initiate the Puppet tests with Jenkins, and verify the test results before allowing a change to be merged into the production pending environment. The existing hook already worked by diverting pushes to production into a temporary pending branch. To facilitate this Jenkins testing, we added a second copy of this pending branch that is used as the actual testing environment. The Git hook then starts the test process by executing a control script on the Jenkins master that starts four jobs managed by Jenkins:

4.1.1. Sync Git branch to Foreman. We use the Foreman puppet:puppet_classes[batch,$env] rake task to import the newly created testing environment into Foreman; the test pool nodes can then be assigned this environment.

4.1.2. Assign VMs to test environment. Using the Foreman REST API, we assign each of the VMs in the Jenkins slave test pool to this newly created and imported testing environment.

4.1.3. Run Puppet agent on each test pool VM. The Puppet agent is run on each test pool VM host to verify that the pushed changes do not introduce any errors, conflicts, or other problems. An optional, second Puppet agent run can also be performed to test for idempotent Puppet configurations, which means that every Puppet run produces the same results, and no changes to the system are made during a second run.

4.1.4. Reset VMs. The test pool VMs are reset to their pre-test state using the RHEV REST & shell APIs. Each VM host is shutdown, and its image snapshot is rolled back to the state in which it was before the Puppet agent run. This step prepares the test pool VMs for the next set of changes to be tested, since all changes performed by the Puppet agent run will be undone and the OS image is restored to a base configuration.

4.2. Jenkins Test Process
Figure 1 below shows a simplified flowchart of the Jenkins jobs and how administrators will interact with the modified system when they push changes to the production environment, all of which is then automatically tested by Jenkins.

When an administrator pushes a change to the production environment in Puppet, it instantly kicks off an automated test of that change on the Jenkins slave pool VMs. When the test is complete, the slave pool is reset and the CGI approval screen displays the results of the test. If any node fails the test, then the pending change will not be merged; it must be deleted, and the administrator must fix the problem and try pushing an updated change to restart the process again. If the test passes, then the change can be approved and merged, and if there are any other pending changes remaining, then that approved change is also merged into the remaining pending environments, which are then retested to verify that they do not introduce any problems or conflicts with the change that was just approved.
Figure 2 below shows the modified CGI approval screen with the Jenkins test status icon added. In this case, the first change passed the test and the second change failed. In addition to pass (green check mark) and fail (red ball) indicators, there are icons for queued (watch) and in-progress testing (blinking watch), and an unknown status icon (gray ball). The black marks in the screenshots obscure internal data like usernames & hostnames that were redacted before being publicly published.

**Figure 1: Jenkins process flowchart.**

Figure 2 below shows the modified CGI approval screen with the Jenkins test status icon added. In this case, the first change passed the test and the second change failed. In addition to pass (green check mark) and fail (red ball) indicators, there are icons for queued (watch) and in-progress testing (blinking watch), and an unknown status icon (gray ball). The black marks in the screenshots obscure internal data like usernames & hostnames that were redacted before being publicly published.

**Figure 2: New Puppet approval page showing Jenkins test status.**
Clicking on the Jenkins status icon shows a table in which you can see the status of the Puppet agent test run on each VM in the Jenkins slave test pool, as shown in Figure 3 below. The table includes the slave VM hostname, timestamp of the most recent test for that pending branch, links to the Foreman Puppet report page for that host, and the Jenkins console output for that Puppet agent run, all of which help the user to easily view the details of a test and figure out what went wrong if necessary. If the Jenkins test is queued or currently in progress, then, instead of the table, a short message prompts the administrator to wait for the test to finish.

| Jenkins Test Host | Status | Timestamp | Puppet Report | Puppet Output |
|-------------------|--------|-----------|---------------|---------------|
| bnl.gov           |        | 03/25/15, 14:25:20 | foreman reports | jenkins console |
| bnl.gov           |        | 03/25/15, 14:25:20 | foreman reports | jenkins console |

For more details, visit the Jenkins dashboard.

Go back to the puppet approve page.

**Figure 3: Jenkins slave pool status table.**

When clicking on the merge link from the pending changes list displayed in Figure 2 above to approve a change that has passed the Jenkins test, the confirmation page with a detailed code diff is shown as before, but it will first verify the Jenkins test results for that pending change. This is shown in Figure 4 below.

![Jenkins merge passed screenshot](image)

**Figure 4: Jenkins merge passed screenshot.**

If an administrator clicks on the merge link to approve a change that failed the Jenkins test, then instead of the confirmation page with detailed diff, a failure message is displayed, followed by the same detailed slave pool table that was shown above when clicking on the Jenkins status icon (Figure 3). This can be seen in Figure 5 below. If the merge link is clicked for a change that is still being tested or is queued for testing, then a short message prompts the administrator to wait until the Jenkins test is complete.
To help debug the cause of a failed test, the administrator can either click on the Foreman link to view the Puppet agent report for the failed hosts, or click on the Jenkins console link to view the Puppet agent console output for the failed hosts. A screenshot of the Jenkins console is shown below in Figure 6. To produce this particular failure case, one of our Puppet modules was edited to include a second resource definition for the wget package, which conflicts with the original definition that already exists in another Puppet module. When both modules are included on the same host, then Puppet fails with a duplicate resource definition error, as can be seen in the Jenkins console output, in red text, in Figure 6. This error also demonstrates how the continuous integration testing method can find problems and conflicts across modules, since it tests the entire host level catalog, which includes many Puppet modules. A normal unit or acceptance test typically used on individual Puppet modules would never detect such a potential conflict.

Figure 5: Jenkins merge failed screenshot.

Figure 6: Jenkins screenshot showing Puppet agent console output.
4.3. Miscellaneous
In addition to the four Jenkins jobs that are defined and used in Puppet agent testing, we also have a monthly job defined to keep the slave pool VM images up to date. Since Jenkins has the ability to schedule jobs based on date and time, we have scheduled a job to run overnight once per month on each of the slave pool nodes. This job does a yum update to keep the base RHEL OS updated, copies the Jenkins slave.jar file from the Jenkins master, and performs a Puppet agent run to keep the basic OS configuration updated. After everything is updated, the job shuts the VMs down, updates their base snapshot used for testing, and boots them back up. Finally, the monthly job purges old, archived test results to keep from filling up Jenkins with years’ worth of outdated data.

5. Future Plans
The last major remaining task to complete before we incorporate this automated Jenkins testing system into our main production Puppet master is to automate the reconfiguration of the Puppet agent run during the monthly VM update job. Since we want the Puppet agent run during the Jenkins test jobs to include all of the important and critical Puppet modules that need testing, but the monthly update job to only do a minimal base Puppet agent run, we need to be able to automatically update our customized Puppet ENC system to be able to reclassify the nodes to a bare minimum configuration during the monthly update job, and then return to the normal, Foreman-based ENC definitions for all Jenkins testing.

Once this has been completed and implemented on our production Puppet master, we will also need our facility staff to assist us in setting up the test pool slave VM images in order to validate all of the critical Puppet modules that need to be tested, and to try to replicate as best as we can all important production hosts and services, so we can automatically validate all changes that are pushed to our production environment.

5.1. Long Term Plans
After several months of using this system on our production Puppet master, we would like to look into the possibility of adding acceptance testing after the Puppet agent run, to validate that the Puppet agent configuration behaved as expected when setting up hosts and services, and not simply verify that the Puppet agent ran without any problems. Some possibilities for achieving this goal include the use of a monitoring system like Nagios or Icinga, or Puppet Serverspec acceptance testing tool to schedule a fifth monitoring or validation job after the Puppet agent run, and to verify its results before allowing the changes to be merged.

References
[1] RSpec – RSpec tests for Puppet manifests, http://rspec-puppet.com/
[2] Serverspec – RSpec tests for servers, http://serverspec.org/
[3] Beaker – Puppet Labs cloud enabled acceptance testing, https://github.com/puppetlabs/beaker/
[4] Jenkins CI – Continuous job execution & monitoring system, https://jenkins-ci.org/
[5] Travis CI – Open-source hosted continuous integration service, https://travis-ci.org/
[6] Foreman – Complete lifecycle mgmt. tool for physical & virtual servers, http://theforeman.org/
[7] RHEV – Red Hat Enterprise Virtualization, http://www.redhat.com/rhev3/