Puppet, Gitlab, and R10k

At the Space Science and Engineering Center

Scott Nolin, SSEC Technical Computing

10/31/2016

Introduction

I am here to talk about how we do linux configuration management at SSEC. I am doing this to encourage others doing similar work so we can ideally share ideas and learn from each other. This is a topic that is important and useful for UW IT in general, and sharing information we can help each other.

About SSEC and Technical Computing (TC)

The Space Science and Engineering Center at the University of Wisconsin Madison

A research and development center focusing on geophysical research and technology to enhance our understanding of the atmosphere of Earth, the other planets in our Solar System, and the cosmos.

Major SSEC Initiatives

- atmospheric studies of Earth and other planets
- interactive computing, data access, and image processing
- spaceflight hardware development and fabrication

The SSEC Mission and Structure Shape Technical Computing

Mission

- Satellite meteorology requires high performance computing and large data. The ratio of computing to data is weighted towards data.

Structure

- SSEC is a matrix organization – teams of people from various units work together on multiple projects.
- Multiple simultaneous projects means often varying requirements for computing. SSEC computing must be responsive to this, we must be able to change, adapt, and provide project needs rapidly.

The Technical Computing Group (TC)

Support all computing at SSEC

- Uniquely responsible for helping center as a whole
- Routine operations
- Security
- Research computing

Six Full Time staff

- 2 Windows/Mac admins – more traditional IT infrastructure
- 3 Linux - Science integration, instrumentation, and HPC focused
- 1 – Head of TC

Up to Six Students

- Assist with helpdesk

SSEC Computing Footprint

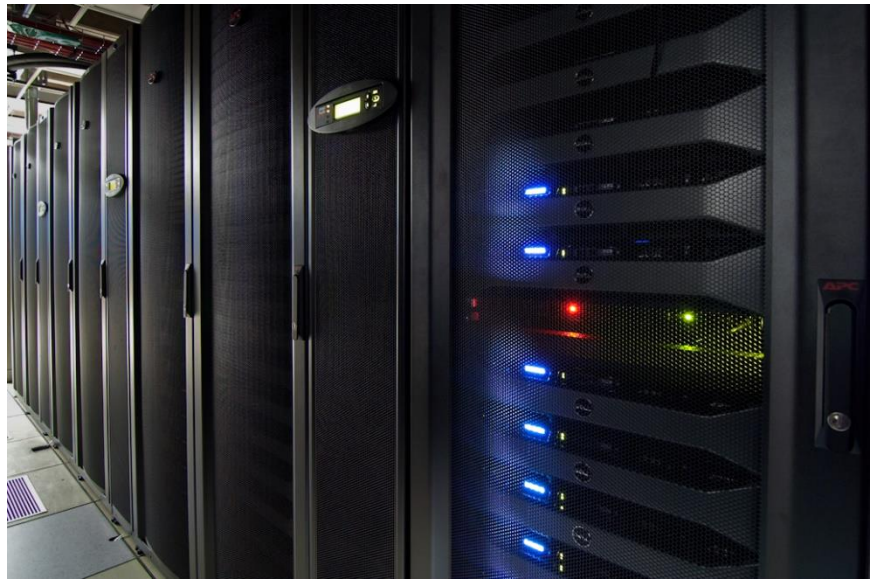
100 macs – mostly laptops

135 Windows (8 virtual) – mostly desktops and laptops

755 Linux (50 virtual, 705 physical) – supporting research mission

- 259 cluster nodes
- 170 HPC Storage servers, about 12PB of storage

250 staff users, 50 student staff, 150 collaborators

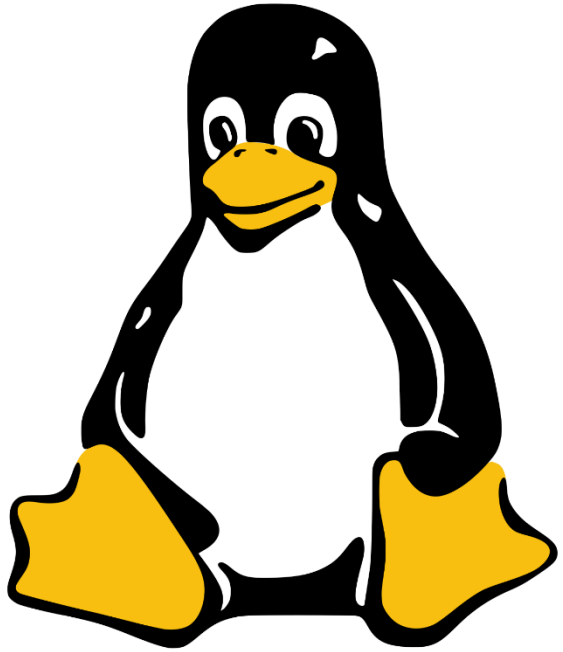


Configuration Management, Version Control

Why We need Linux configuration management

The center's mission and structure means there are many systems, and many groups with different needs running systems. Linux systems are the core of our research systems.

- Ensure consistency and repeatability
- Faster time from install to production
- Adapt to change quickly
- Lower labor costs
- Computer Security
  - Applying computer policy consistently
  - Auditing



Why Version Control

- Version control of configurations reduces fear and increases accountability.
  - See Dave Plonka's 'Sysadmin File Revision Control with RCS' from 1998
- We had done this for many years with CVS, SVN, and now git and some of our configurations.
  - DNS zone files
  - Monitoring server configuration (check_mk/omd)
  - Version control was the 'end' of for example a script updating zone files or the check_mk configuration. We saved configurations we applied to config managment.
- With Puppet, you manage 'everything', so configuration management is even more important.
  - We move version control to the forefront. We **start** with version control and end with changes in puppet.
  - Version control is how we trigger a configuration change, not how we store a change we made.

The SSEC System

Puppet History at SSEC

2011, Initial: Puppet server for testing and light use

- Used to distribute science software rpms/lmod modules, yum repositories, and monitoring client checks
- Edit modules directly on server
- Assign modules to hosts in server configuration files
- No integrated version control (ad-hoc for some modules)

February 2015, Configuration as Code: Puppet, Gitlab, gitlab hooks, and puppet-sync

- **Version Controlled**
- pdxcat/puppet-sync
  - Pre-receive hooks for tests, Post-receive hooks for puppet-sync
- branches are environments
- Git workflow accelerated development of modules
- Used for ‘everything’, new stuff and converting existing.
- Problem with git client and gitlab gui and hooks - tests were in pre-receive, never get triggered in GUI
- Monolithic gitlab repository bad for sharing with other groups

July 2016, Gitlab-CI method: Puppet, Gitlab, Gitlab-CI, and R10k

- The gitlab GUI is a first class citizen
- CI/CD methods are generally well understood and accepted
- Gitlab-CI provides a gui, making it easier to visualize and maintain your production pipeline.

SSEC Workflow: Routine Administration

- **Node Classification** – we use Foreman as External Node Classifier, there are other options
  - Assign nodes to environments, such as a test environment
  - Assign modules (puppet classes) to nodes
- Simple **data changes** for hosts or groups of hosts are done in master branch of control repository. Gitlab GUI or git cli tools.
  - User operations: add local users, override attributes (example: modify UID if needed), Allow access to a machine for a domain user.
  - Manage public keys
  - Sudoers

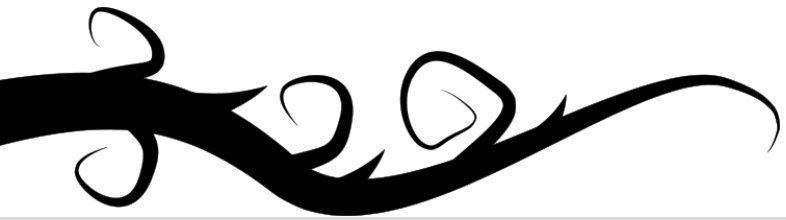
SSEC Workflow: Bigger Changes

1. Create short lived branches, for example “scott_test”
2. Assign test nodes to scott_test environment
3. Test changes
4. Merge test branches to master
5. Move test nodes back to master

Long Lived Branch Workflow

Our short lived branch workflow is only one way to do this. Another idea is to have a long lived test branch.

- Force all changes to go to the test branch and merge to master (or ‘production’) after passing
- Have some hosts permanently assigned to the test environment
- This can be a highly structured workflow with checks and balances. Only some admins could be allowed rights to merge for example.
- These methods can be combined. Permanent testing VM’s for a long lived branch, but still doing the short-lived branch test methods is a direction we may go at SSEC. I especially like the idea of some nightly complete system rebuild testing.



Where did I get the idea for puppet/r10k/gitlab?

This is based on first looking at Phil Zimmerman's
'An Automated R10K Workflow That Works' and attempting to replicate it.
http://philzim.com/2014/05/02/an-automated-r10k-workflow-that-works/

This uses

- R10k
- Github Enterprise
- Sinatra
- Capistrano
- Jenkins
- Ruby
- https://github.com/pzim/reaktor

As I was wrestling with various components gitlab-ci developed rapidly and showed great promise. It worked.

SSEC System Components

- Puppet – configuration management server
- Gitlab– git server, with integrated Continuous Integration features (CI).
  - Includes gitlab-ci
  - CI Scripts you write do the work: bash, perl, python, whatever you want
- Foreman – Puppet GUI and node classifier
  - optional, but nice
- R10K
  - Code management tool that maps Git branches to Puppet environments.
  - Make a branch called ‘testing’ in git, and an puppet environment called ‘testing’ is created. Assign test machines to testing branch and test.

Aside: Puppet Enterprise Code Manager

Puppet enterprise now includes Puppet Code Manager, a tool that manages r10k for you. Our method uses gitlab-ci scripts.

Code Manager is Enterprise only.

One technical difference that may be important to some is it pauses the puppet server during updates to avoid conflicts.

R10K: Control Repository and Puppetfile

The control repository controls the puppet environments, it is a critical component. The control repository contains the Puppetfile and more.

The puppetfile is a file listing all other repositories with modules you use. These can be any combination of your own git or puppet forge, your own private, or someone else's. For security we use a dedicated gitlab server, and download, curate and re-host any repositories we use from outside.

Consider that you likely would not share your control repository with others, but may share other modules, which are building blocks.

```
#Puppetfile example
#Here are some Puppet Modules. These reference our git server, but
#could be any git or puppet forge
mod 'ipmi',
  :git => 'git@sgit.ssec.wisc.edu:tc-puppet/ipmi.git',
  :ref => 'master'
mod 'winbind',
  :git => 'git@sgit.ssec.wisc.edu:tc-puppet/winbind.git',
  :ref => 'master'
mod 'augeasproviders_core',
  :git => 'git@sgit.ssec.wisc.edu:tc-puppet/augeasproviders_core.git',
  :ref => 'master'
mod 'zpool_scrub',
  :git => 'git@sgit.ssec.wisc.edu:tc-puppet/zpool_scrub.git',
  :ref => 'master'
```

R10K: Control Repository Contents

Anything that is routinely changed will be more convenient to be in control than an external repository. You could have a mostly empty puppetfile and put most puppet modules in control, mimicking the monolithic puppet-sync method.

- **hieradata** – this can be it's own module, but for SSEC, is changed frequently. For example, it's how we allow users login to linux systems. So we put it in control.
- **site/<modulename>** – we put the 'site' directory is in the modulepath, so this adds modules in the control repo directly
  - You can put any modules you want in site/
  - We use - site/profiles, site/role, and site/hostgroups. Notice these are generally 'site specific' things for SSEC, likely would not be shared. They're how we compose things, using other basic building block modules from outside control repo, as defined in the Puppetfile.

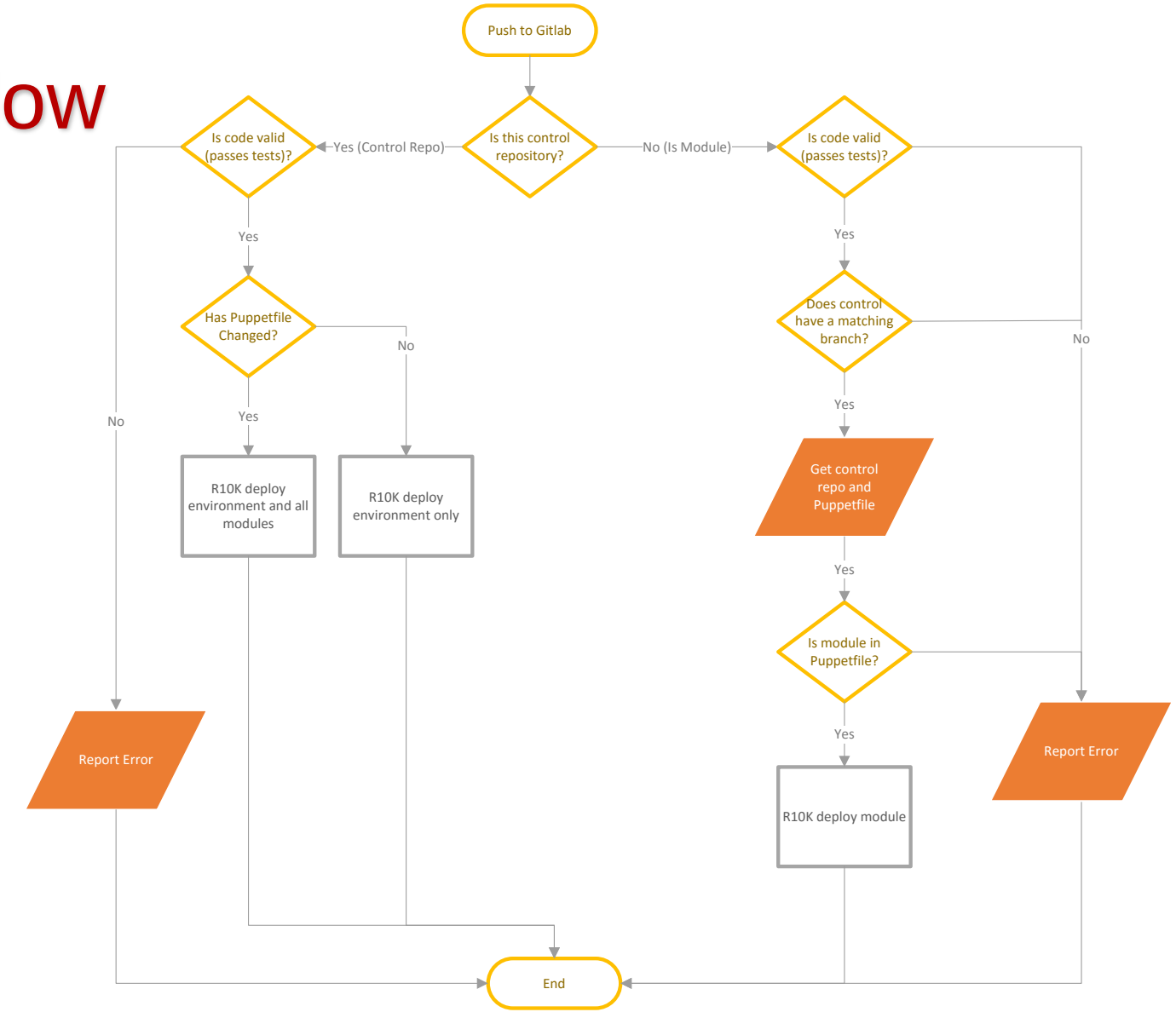
Gitlab CI

Gitlab CI is an integrated CI/CD system that comes with gitlab for testing, building, and deploying code

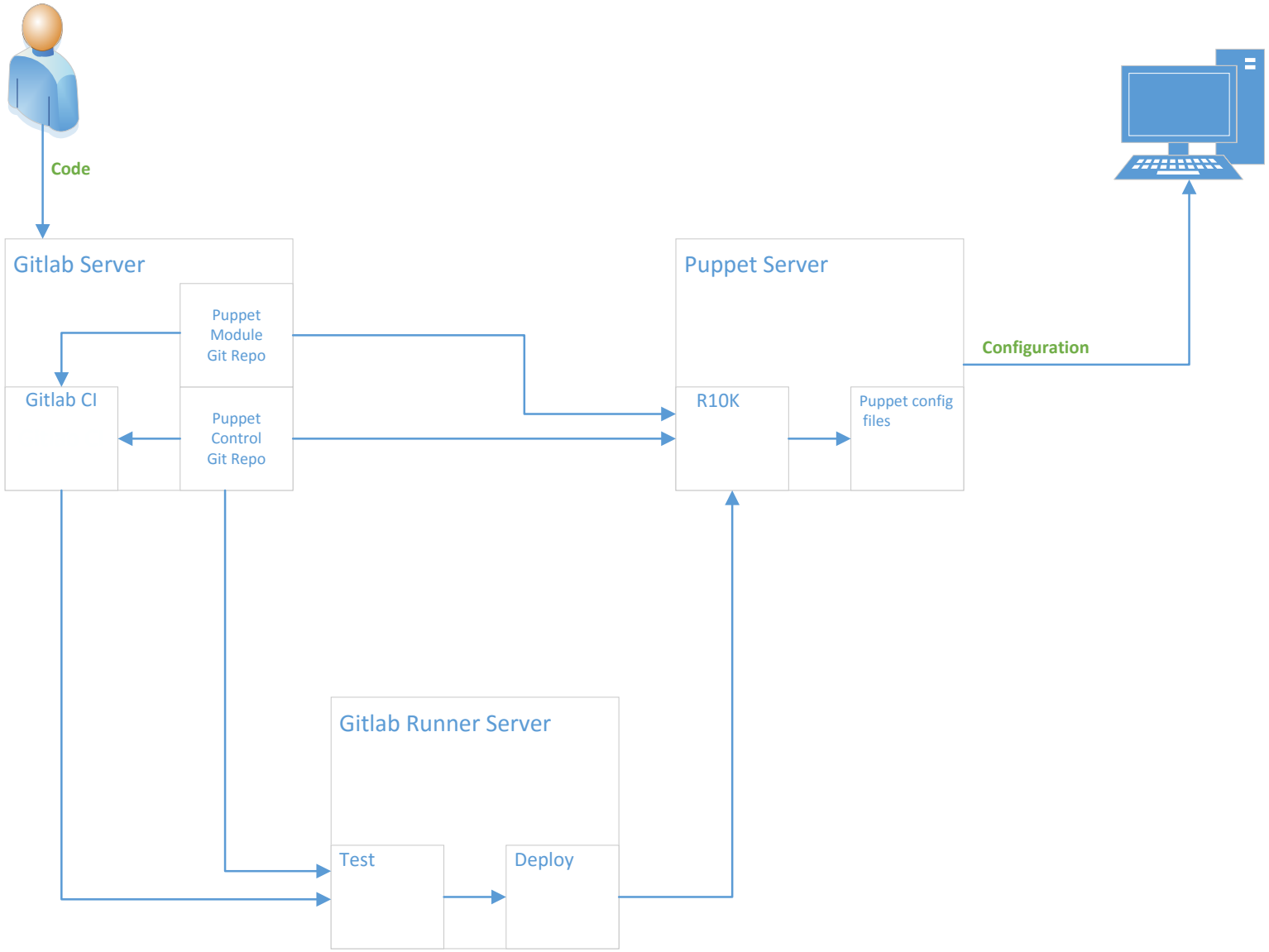
- Continuous Integration (CI) is a development practice where you integrate code for a project frequently. Check-ins are tested via an automatic build process.
- Continuous Deployment (CD) is similar, focused on delivery of software to the end user.

In our use case, we test our puppet modules, and deploy them to the puppet server. We use Gitlab-CI and R10K instead of our previous git server-side hooks and pdxcat/puppet-sync.

Process Flow



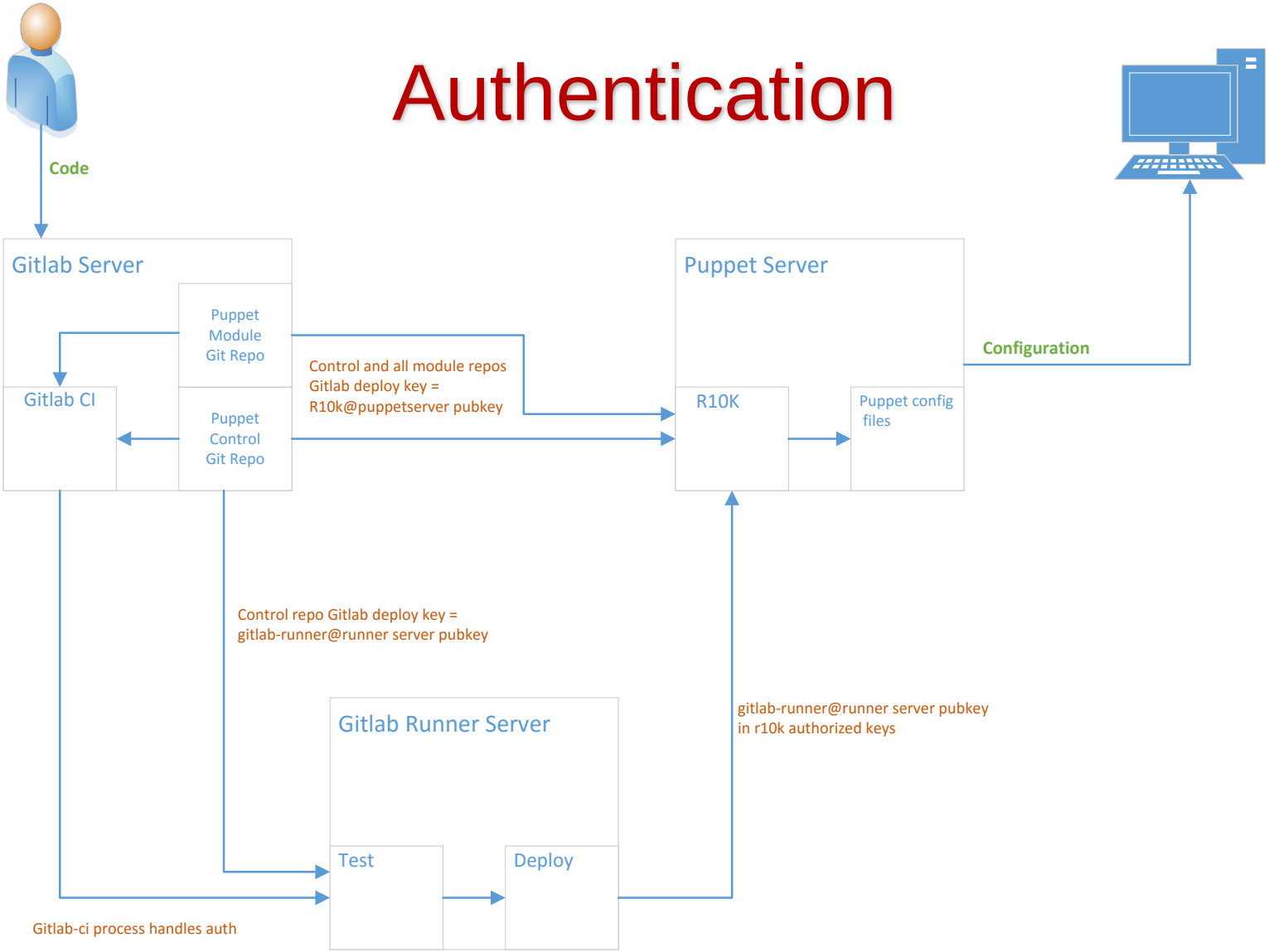
Data Flow


Code
Gitlab Server
Puppet Module Git Repo
Gitlab CI
Puppet Control Git Repo
Puppet Server
R10K
Puppet config files
Configuration
Gitlab Runner Server
Test
Deploy

Authentication Between Components

- **Gitlab deploy keys** – allow read-only access to a gitlab repository
  - R10K user @ puppet server's ssh public key is used as gitlab deploy key on control and all module repos. This is how r10k@puppet_server grabs the repositories from gitlab.
  - Gitlab-runner user @ runner server's ssh public key is used as gitlab deploy key on the control repository only. The runner uses this to get the puppetfile for tests.
- **Normal system SSH keys**
  - Gitlab-runner@runner_server ssh public key is added to r10k@puppet_server authorized keys. This is how the gitlab runner can ssh to the puppet server to run r10k commands.
  - We manage public keys with puppet, adding them to authorized_keys. Of course there's the bootstrap issue for these in particular.
- Maybe a diagram will help?

Authentication



Gitlab CI: Control Repository

Adding .gitlab-ci.yml file to the repo defines the process

```
stages:
  - test
  - r10k

puppet_test:
  stage: test
  script: puppet_test
  allow_failure: false

hiera_test:
  stage: test
  script: hiera_test
  allow_failure: false

r10kdeploy:
  stage: r10k
  script: r10k_control_deploy
  allow_failure: false
```

Scripts are run on the gitlab-runner linux host. There are docker and shell runner methods, we use shell, but either will work.

The gitlab-runner user simply executes whatever script you give it in it's shell.

This naturally has security concerns, the gitlab-runner needs to be treated as securely as all other system components. If compromised, all your hosts are compromised easily.

Pipeline #1615 with 4 builds for master in 6 seconds

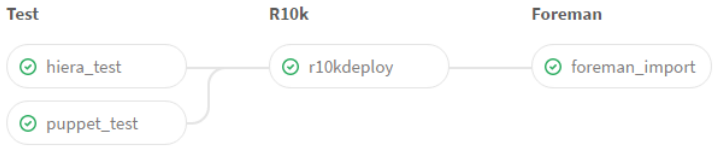
passed

Authored by **Scott Nolin** 9 minutes ago

Commit 783fd984280ffde51091f3fc382478e29b2bb387

cleanup_die doesn't exist

Hide pipeline graph



Status	Build ID	Name		
Test				
passed	#6208	hiera_test	00:01 less than a minute ago	
passed	#6207	puppet_test	00:01 less than a minute ago	
R10k				
passed	#6209	r10kdeploy	00:02 less than a minute ago	
Foreman				
passed	#6210	foreman_import	00:01 less than a minute ago	

Gitlab CI: All Other Modules

Adding .gitlab-ci.yml file to the repo defines the process for modules

```
stages:
  - test
  - r10kdeploy
  - foreman

puppet_test:
  stage: test
  script: puppet_test
  allow_failure: false

r10K_modtest:
  stage: test
  script: r10k_modtest
  allow_failure: false

r10k_moddeploy:
  stage: r10kdeploy
  script: r10k_moddeploy
  allow_failure: false
```

Our general puppet tests are the same, and notice we don't test hieradata – that lives in control only.

We do module specific tests and deployment.

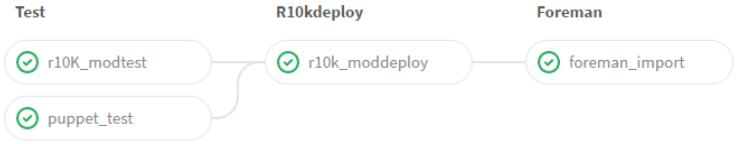
Pipeline #1729 with 4 builds for master in 3 seconds (queued for 1 second) — passed

Authored by **Jesse Stroik** about an hour ago

Commit 62a5768c95eff6b54ac3a41705a4e0c98e4010ba

replacing the silent.cfg files

Hide pipeline graph



Status	Build ID	Name		
Test				
passed	#6618	puppet_test	00:00 about an hour ago	
passed	#6619	r10K_modtest	00:01 about an hour ago	
R10kdeploy				
passed	#6620	r10k_moddeploy	00:01 about an hour ago	
Foreman				
passed	#6621	foreman_import	00:00 about an hour ago	

Gitlab CI Script Details

- The following slides examine parts of the gitlab-ci scripts that may help explain the processes I think are critical. I will share complete scripts if you would like, just let me know.

- We can also go back and discuss the overall process more at this point, as the following slides are details.

CI Scripts: Control Modules

r10k_control_deploy

```
#!/usr/bin/perl

use strict;
use warnings;
my $PREFIX="tc";
my $BRANCH="$ENV{CI_BUILD_REF_NAME}"; #branch name                    #deploy the branch
my $r10k_env="
my $NEW_MODULE                                                                    S:\n";
my $NEW_MODBRA                                                                    L0k deploy
my $NEW_ENV=0;                                                                    ');
my $PUPPETSERV
my $exit_val;
#debugging and                                                                    SERVER} r10k
                                                                                  verbose\n");
#get changed f
my $puppetfile
my @changedfil
die("FAILED: g                                                                    L0k deploy
foreach my $line (@changedfiles) {                environment $r10k_env --verbose");
        chomp $line;                                      $exit_val = $? >> 8;
        print "$line\n";                                  if ($exit_val != 0) {
        my @linearray = split(' ',$line);                   die("failed to: ssh r10k\@${PUPPETSERVER} r10k
        if ($linearray[1] eq 'Puppetfile') {      deploy environment $r10k_env --verbose\n");
                $puppetfile_changed = 1;                  }
        }                                         }
}
```

If you find my scripts depressing, make better versions and share please!

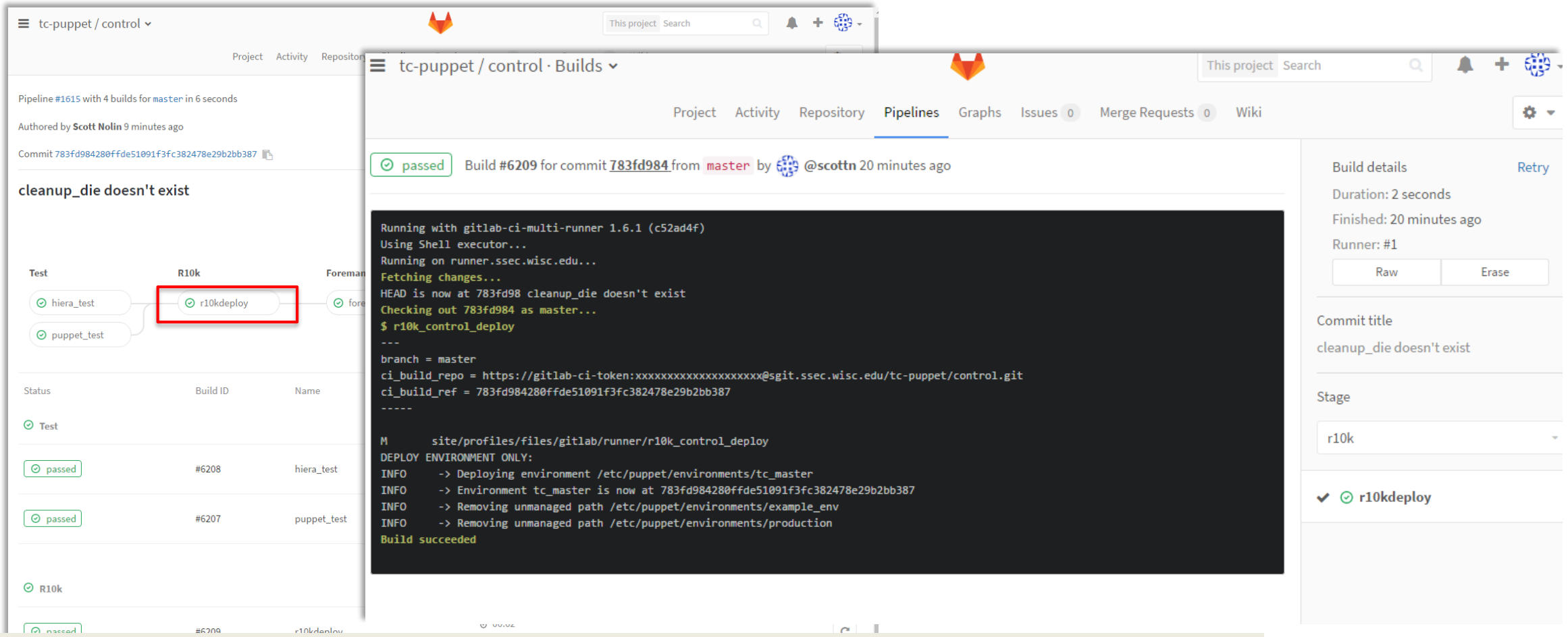
r10k_control_deploy

```
#!/usr/bin/perl

use strict;
use warnings;
my $PREFIX="tc";
my $BRANCH="$ENV{CI_BUILD_REF_NAME}"; #branch name
my $r10k_env="${PREFIX}_${BRANCH}";
my $NEW_MODULE=0;
my $NEW_MODBRANCH=0;
my $NEW_ENV=0;
my $PUPPETSERVER="myserver.com";
my $exit_val;
#debugging and info subroutine removed for brevity

#get changed files
my $puppetfile_changed = 0;
my @changedfiles = `git diff --name-status HEAD^` or
die("FAILED: git diff --name-status HEAD^\n");
foreach my $line (@changedfiles) {
        chomp $line;
        print "$line\n";
        my @linearray = split(' ',$line);
        if ($linearray[1] eq 'Puppetfile') {
                $puppetfile_changed = 1;
        }
}

#deploy the branch
if ($puppetfile_changed) {
        print "DEPLOY ENVIRONMENT AND MODULES:\n";
        system ("ssh r10k\@${PUPPETSERVER} r10k deploy
environment $r10k_env --puppetfile --verbose");
        $exit_val = $? >> 8;
        if ($exit_val != 0) {
          die("failed to: ssh r10k\@${PUPPETSERVER} r10k
deploy environment $r10k_env --puppetfile --verbose\n");
        }
} else {
        print "DEPLOY ENVIRONMENT ONLY:\n";
        system ("ssh r10k\@${PUPPETSERVER} r10k deploy
environment $r10k_env --verbose");
        $exit_val = $? >> 8;
        if ($exit_val != 0) {
          die("failed to: ssh r10k\@${PUPPETSERVER} r10k
deploy environment $r10k_env --verbose\n");
        }
}
```



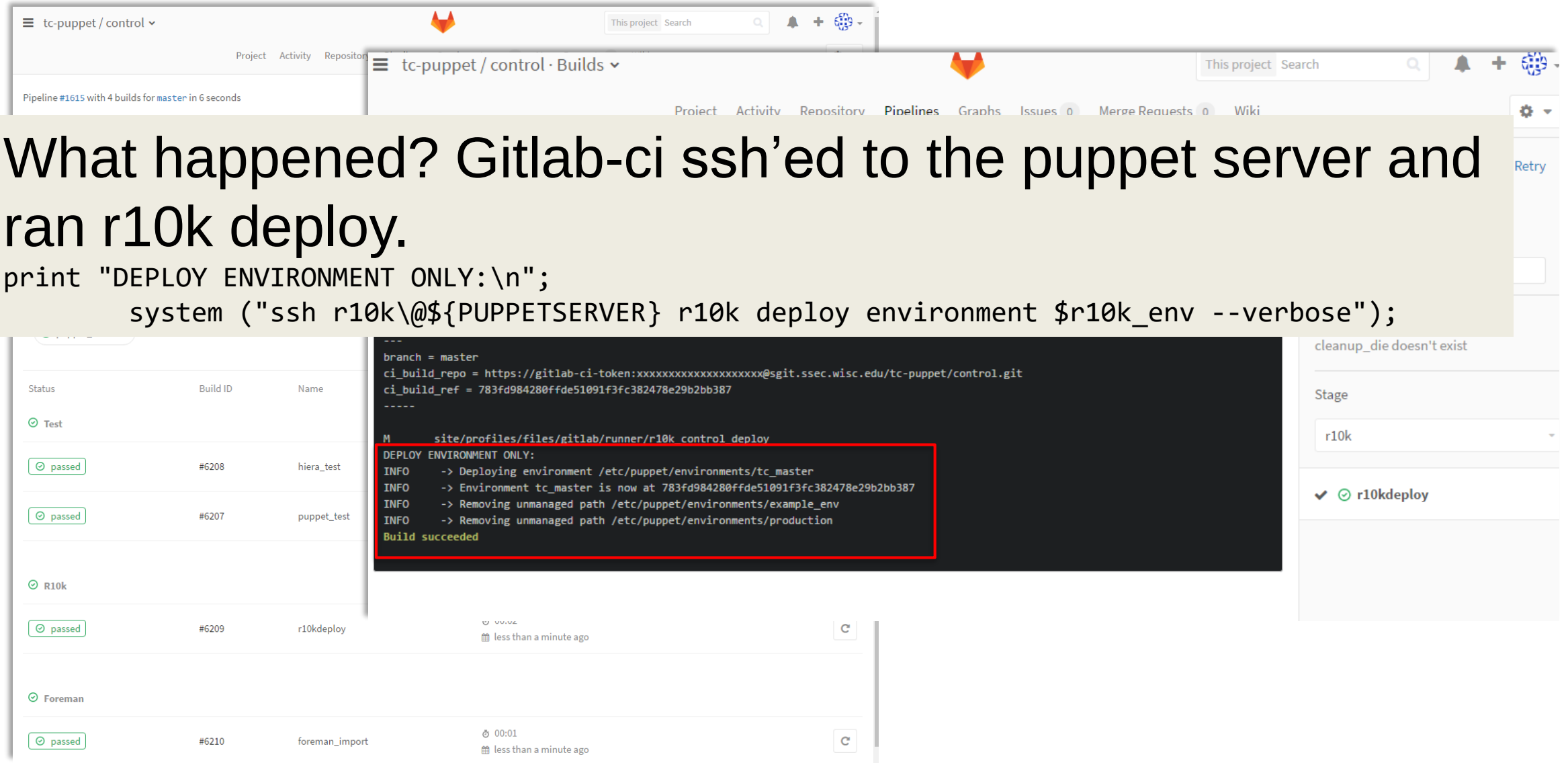
Clicking the CI Step shows the output.

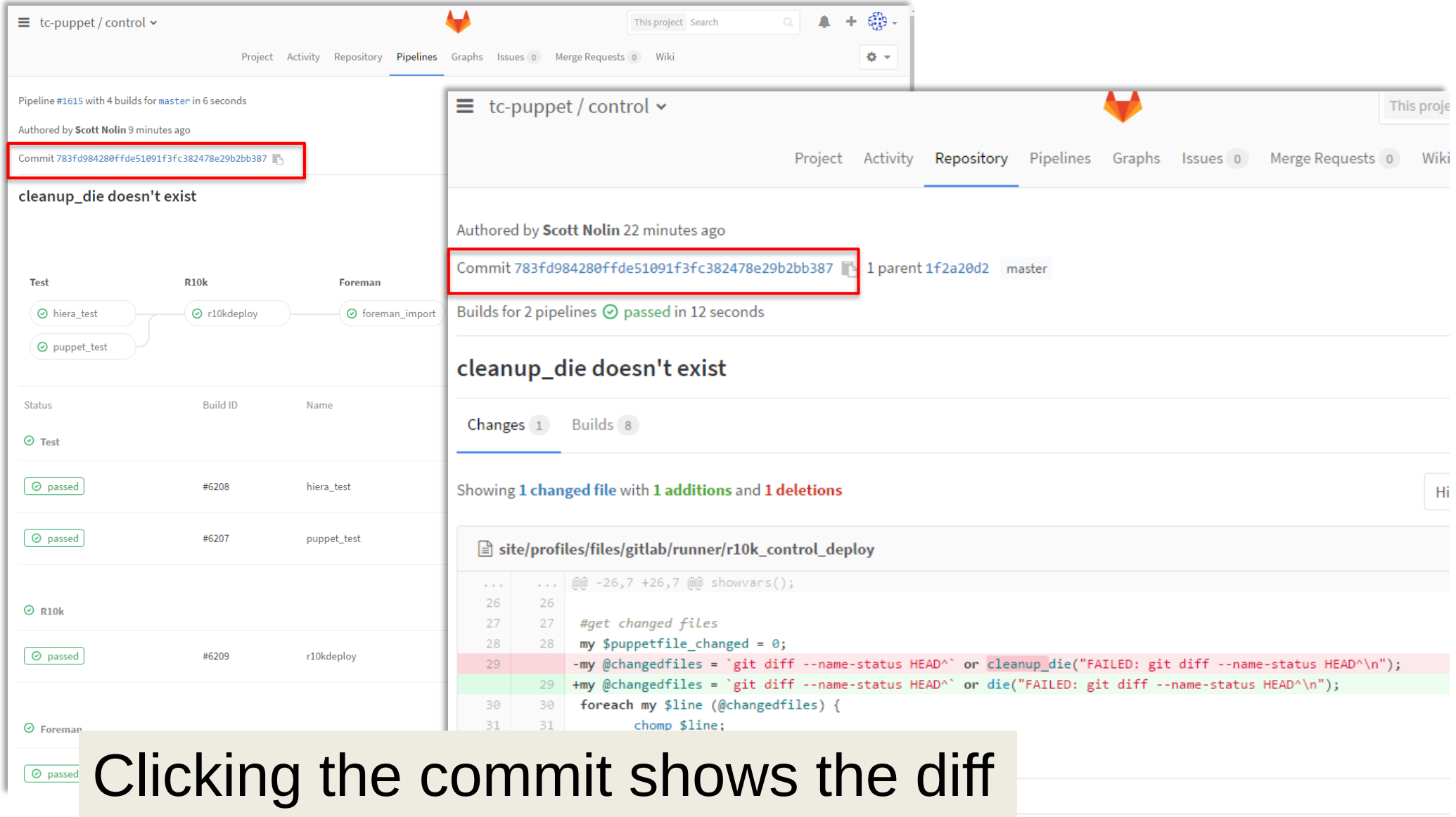
In this case I modified r10k_control_deploy itself.

I found a bug while making these slides.

What happened? Gitlab-ci ssh'ed to the puppet server and ran r10k deploy.

```
print "DEPLOY ENVIRONMENT ONLY:\n";
        system ("ssh r10k\@${PUPPETSERVER} r10k deploy environment $r10k_env --verbose");
```





Clicking the commit shows the diff

CI Scripts: Other Modules (not control)

r10k_modtest: check for matching control branch

```
my $BUILD_TMP="/home/gitlab-runner/tmp/$ENV{CI_BUILD_ID}";
my $PUPPETFILE="${BUILD_TMP}/Puppetfile";
my $GITBASE='git@sgit.ssec.wisc.edu:tc-puppet';
my $CONTROLREPO="${GITBASE}/control.git";
my $BRANCH="$ENV{CI_BUILD_REF_NAME}"; #branch name

my $heads = `git ls-remote --heads $CONTROLREPO $BRANCH`;
my $exit_val = $? >> 8;
  #system returns 0 on success, shift 8 bits
if ($exit_val != 0) {
  cleanup_die("Failed to\n git ls-remote --heads $CONTROLREPO $ENV{CI_BUILD_REF_NAME}\n\n");
}
print "$heads\n";
my @head_array = split ('/', $heads);
chomp @head_array;
if (exists $head_array[2]) {
  if ( $head_array[2] eq $BRANCH ) {
    $CONTROL_BRANCH_EXISTS = "true";
  }
}
```

R10k_modtest is too large for a slide. Complete scripts are available, see addenda.

If you make a branch called “test” on the “foo” repository, control should have a matching “test” branch.

So gitlab-runner user needs permission to read the control repo to see if the branch exits.

Our process simply fails if it does not exist. Branching control is manual. Our daily work is all in ‘control’ typically, so this doesn’t happen often. Junior admins work entirely in control repository.

r10k_modtest: checking against puppetfile

```
#PUPPETFILE - Puppet Modules
mod 'ipmi',
  :git => 'git@sgit.ssec.wisc.edu:tc-puppet/ipmi.git',
  :ref => 'master'
mod 'winbind',
  :git => 'git@sgit.ssec.wisc.edu:tc-puppet/winbind.git',
  :ref => 'master'
mod 'augeasproviders_core',
  :git => 'git@sgit.ssec.wisc.edu:tc-puppet/augeasproviders_core.git',
  :ref => 'master'
mod 'zpool_scrub',
  :git => 'git@sgit.ssec.wisc.edu:tc-puppet/zpool_scrub.git',
  :ref => 'master'
```

Gitlab-runner is working with the module “other” remember? The puppetfile is in control repo, and should contain references to this other module. So the gitlab runner user must be trusted to do a git clone of control.

For example “impi” is shown in our example puppetfile.

We then parse puppetfile for the expected entries, and fail if they are missing as we did with our branch check.

r10k_modtest: checking against puppetfile

```
system ("git clone  -b ${BRANCH} --single-branch ${CONTROLREPO} ${BUILD_TMP}");
$exit_val = $? >> 8;
if ($exit_val != 0) {
  cleanup_die("failed to:\n git clone  -b ${BRANCH} --single-branch ${CONTROLREPO}
${BUILD_TMP}\n\n");
}

#parse puppetfile
# - we read the
open my $fh, $PUPPETFILE or cleanup_die("Failed to open $PUPPETFILE\n");
chomp(my @lines = <$fh>);
close $fh;

....(etc)
```

Here's just the bit where we clone the control repository.

r10k_moddeploy

```
#!/usr/bin/perl

use strict;
use warnings;

my $PREFIX="tc";
my $BRANCH="$ENV{CI_BUILD_REF_NAME}"; #branch name
my $r10k_env="${PREFIX}_${BRANCH}";
my $PUPPETSERVER="foreman.ssec.wisc.edu";
my $exit_val;

#NOTE MOD_SHORT - we are assuming the module name is always
embedded
# in the repo name as the last path entry
# example https://gitlab-ci-token:xxxx@sgit.ssec.wisc.edu/tc-
puppet/szip.git
#  = 'szip'
#
my @THISREPO = split ('/', $ENV{CI_BUILD_REPO});
my $MOD_SHORT = pop @THISREPO;
$MOD_SHORT =~ s/\.git//g;

print "DEPLOY\n";
system ("ssh r10k\@${PUPPETSERVER} r10k deploy module -e
$r10k_env $MOD_SHORT --verbose");
$exit_val = $? >> 8;
if ($exit_val != 0) {
  die("failed to: ssh r10k\@${PUPPETSERVER} r10k deploy module -
e $r10k_env $MOD_SHORT --verbose\n");
}
```

Discussion