CI/CD with Jenkins and Beaker

Thomas Løkkeborg, 473157

November 5, 2018

Abstract

CI/CD is high-level term that is hard to grasp without practical examples. In this report I will put forward my own experiences from trying to set up a Jenkins CI/CD pipeline running in OpenStack using Beaker and Vagrant.

Contents

1 Introduction

The original goal of this project was to experiment with CI/CD using Jenkins and Beaker. As a vessel for achieving this goal I decided to try and create an infrastructure with a Puppet Master, a Jenkins server and an application server, where changes to the infrastructure were run through a CI/CD pipeline on the Jenkins server. The whole infrastructure should be completely defined in code, such that all that is needed to bring up a copy is some environment-dependant ”yaml” files and a `openstack create` command.

See 4 for a list of what I worked on during the project.

2 Background terms and technology

2.1 CI/CD

As the whole project is focused around CI/CD I feel it's appropriate to present the definition of the term I've been using during the project. The term refers to the combined practise of ”Continuous Integration” and ”Continuous Delivery”. Our course book[58] refers to Martin Fowler for definitions of these terms, so I will as well.

2.1.1 Continuous Integration

“Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.”[57]

2.1.2 Continuous Delivery

“Continuous Delivery is a software development discipline where you build software in such a way that the software can be released to production at any time.”[56]

2.2 Beaker

Beaker[2] is a test harness focused on acceptance testing machines. The project is maintained by Puppetlabs themselves, and is widely used. Beaker abstracts away the underlying machines it is testing on by using the concept of SUT's, ”Systems Under Test”, so it can test for any kind of machine that is supported though plugins. The project consists of many parts, and the relevant ones for this project are ”beaker-openstack”[5] and ”beaker-vagrant”[7].

2.2.1 beaker-openstack

Beaker-openstack is meant to allow Beaker to use OpenStack as a provider of SUT's. This plugin is experimental, and the library code reflects that. I have not found any

examples of people actually using this plugin.

2.2.2 beaker-vagrant

Beaker-vagrant utilizes Vagrant to create and provision nodes. When using this plugin Beaker will create Vagrant configuration files for you, so it seems like there is no room for customization at the Vagrant layer.

2.3 Vagrant

Vagrant[52] is an Open-source automation tool for creating and provisioning virtual machines. It was created to make it straight-forward to create disposable and reproducible virtual machines defined in code. It is from my understanding primarily used to create development environments, but it can be used for anything involving virtual machines. Vagrant is not tied to any particular virtualisation technology, and uses the concept of ”providers” as a high-level abstraction for underlying hypervisors. As Vagrant is widely used you can find providers for pretty much anything. Worth mentioning is the VirtualBox and OpenStack providers, which are the ones I used during this project. The VirtualBox provider is the most popular choice, due to how simple it is to set up. There are a couple OpenStack providers, but I chose to use the ”vagrant-openstack-provider”[51], as it bases it's machines on images from OpenStack, whereas the other one uses OpenStack compliant boxes.

2.4 Jenkins Configuration as Code

”Jenkins Configuration as Code”[26], or ”JCasC”, is a Jenkins plugin that seeks to replace the Jenkins web UI with yaml configuration files. Using this plugin it is possible to completely configure a Jenkins server through code. The project has been accepted as a standard component of the Jenkins project, and will be incorporated into Jenkins itself eventually. JCasC is very new, so documentation is sparse.

3 Survey of similar projects

Following are the projects I looked at when trying to set up control-repo testing. Testing application code is a more general concept, so I did not look at any specific projects for that.

3.1 PSICK

”Puppet Systems Infrastructure Construction Kit”[38], or ”PSICK”, is a feature-rich tool for generating Puppet control-repos. Control-repos generated by this tool come with many features included, and one of those are testing. The project uses Beaker, and allows for testing on Docker containers or through Vagrant.

I did not choose to use this project as my control-repo because it seems rather complex, and because it seems to rely on either Vagrant with VirtualBox or Docker to run it's Beaker tests.

3.2 Onceover

Onceover[30] is a tool for generating and running tests for control-repos. It seems very promising, as it automatically generates tests by looking through your existing codebase. It used to support acceptance testing through Beaker, but the maintainers have abandoned Beaker to look for a more general solution to acceptance testing [31]. I did try and set this up for my project, but could not find a way to make it run on OpenStack.

3.3 Acceptance testing at OpenStack

OpenStack is using a combination of Vagrant and Beaker to run their Puppet module tests on OpenStack instances [39][33][32]. Because the support for OpenStack nodes is so poor they have decided to create the SUT's via Vagrant using OpenStack as the provider, install Beaker as part of the provisioning process and run Beaker directly on the host (Vagrant controlled) machine by setting Beaker's hypervisor to "none" [17].

4 List of work

Following is a description of what I spent my time working on during the project.

4.1 Research

During the project research was done so I had a general idea of how people where solving the issues I was working on. As a result of this I'm left with a better perspective on automated testing, the Jenkins ecosystem, Beaker and on Vagrant.

4.2 The imt3005-project-cicd repository

`https://github.com/tholok97/imt3005-project-cicd`

This repository serves as the control-repo for the infrastructure I built during the project, and also includes OpenStack Heat code to bring up a stack running the control-repo. During the project I manually tested features directly on the OpenStack instances of the stack, and tried to incorporate changes into this repository once they were stable enough.

4.3 The imt3005-vagrant-vm repository

`https://github.com/tholok97/imt3005-vagrant-vm`

This repository contains the Vagrant environment I used to develop for this project. It includes all the tooling needed to work with the course, and is written to be general enough that other people might find it useful. It is not the focus of this report, but I feel it is worth mentioning, as the bulk of my Vagrant experimentation ended up being setting up this environment.

4.4 The gossinbackup repository

https://github.com/tholok97/gossinbackup

Most of the Beaker documentation is about Puppet module testing, so as part of learning Beaker I wrote some basic acceptance tests for my gossinbackup module.

5 Results and discussion

5.1 OpenStack Heat stack

A core part of this project was having the infrastructure completely defined in code, so creating the stack was key. I based my infrastructure on IaC-Heat-cr[22], so creating the initial setup was went quite quickly. What I've needed to change from that stack was to reduce it down to three machines, make it use my own control-repo instead of contro-repo-cr[10] and to inject some ssh keys through the boot scripts ran at stack creation.

I decided to have the infrastructure definition in the same repository as the control-repo for simplicity. This proved to be beneficial as the infrastructure definition and control-repo always were in-sync. A side-effect is that r10k deploys the infrastructure definition together with the roles and profiles, which seems a little dirty.

Code is included as an appendix: D

5.2 Beaker

Beaker proved to be more difficult to work with than expected. The initial goal was to have my whole control-repo tested through Beaker, but this proved to be a difficult task inside of Skyhigh.

5.2.1 Issues with the documentation

While learning with Beaker I discovered several issues in the documentation. This seems strange, as the project appears to be well maintained. The biggest bug I found was in what might be the most important piece of documentation of all: the "getting started" tutorial [1]. Following the tutorial step by step rewards you with a big error message following the final command, which is not very inviting. The bug was not easy to fix either. It turns out the tutorial uses functionality from Beaker version 3 which has been completely revised for Beaker 4. Installing Puppet on SUT's used to be a part of the core Beaker project, but has since been moved out into a separate plugin and revised. This meant that before I could even run a Beaker test I had to learn one of it's plugins.

I intend to create an issue on the Beaker issue tracker [3] as soon as I have the time. This will be my first Open Source contribution, which is exciting.

5.2.2 The Beaker-openstack plugin

The most common way to run Beaker seems to be using Vagrant with VirtualBox. This was what I initially planned on doing, but it turns out Skyhigh instances do not support hardware virtualization. VirtualBox running on a system without hardware virtualization is not a pleasant experience, as virtual machines must be 32-bit, and creation and deletion will be very slow. I abandoned VirtualBox for what in principle is a much better solution anyways: using Skyhigh itself as the hypervisor for Beaker.

In concept Beaker using Skyhigh for it's SUT's is a good idea. You could run your functional tests on the exact same images as the production systems are using. You could even provision them the exact same way as the production systems as you have access to the boot scripts for the instances in the stack. The issue is that to do this you are dependant on a plugin, and although there is a Beaker-openstack plugin, it is experimental and does not seem stable. First of all it does not immediately appear to support OpenStack Keystone version 3, which is what Skyhigh seems to be using. Authenticating with Keystone version 2 gives 404 responses, and the top-level API call indicates it is deprecated. Looking into the code for the library it does mention Keystone version 3 though, but it is using it in a very strange way. The following snippet is from the part of the library that is causing issues [4]:

```
48  # Keystone version 3 requires users and projects to be scoped
49  if @credentials[:openstack_auth_url].include?('/v3/')
50    @credentials[:openstack_user_domain]    = @options[:openstack_user_domain] || 'Default'
51    @credentials[:openstack_project_domain] = @options[:openstack_project_domain] || 'Default'
52  end
53
54  @compute_client ||= Fog::Compute.new(@credentials)
```

Line 48 to 52 seems like a temporary solution, as it forces you to use "Default" domain when authenticating. The domain Skyhigh is expecting is "NTNU", so this might be part of the issue. Line 54 is where the library completely fails. There is a bug from inside the function call, which is a poorly documented call to another library, so I did not spend time debugging it.

5.2.3 Using Vagrant for creation of SUT's for Beaker

While researching alternate approaches to having Beaker work with OpenStack this was the one that seemed the most stable. Instead of having Beaker manage it's SUT's, you instead use Vagrant and it's excellent "vagrant-openstack-plugin" to manage OpenStack instances to test on, install Beaker on the instances, and trick Beaker to run it's tests on the machine it is installed on, which is the Vagrant-managed OpenStack instance [39][33]. This lets you create exact copies of the production instances, with the only difference being the Beaker installation. Beaker tests can run as normal, but you are giving away the ability to test the system from the outside, as Beaker is inside the SUT calling itself with localhost. Beaker also has the very interesting ability to create multiple SUT's and do tests between them, which you are missing out on.

I did not end up implementing this in the project control-repo, but I have done enough manual testing to know that automating the process is possible. An example of how this looks like is provided as an appendix: A. With the Vagrantfile set up

you would run `vagrant up --debug` followed by `vagrant destroy -f --debug` in a pipeline script. Beaker would run it's tests during `vagrant up --debug`, and fail if anything goes wrong, which is what we want. Jenkins would catch this error and display all the output back to us so we can figure out what went wrong.

5.2.4 Beaker-rspec tests for gossinbackup module

To learn Beaker I created simple tests for my Puppet gossinbackup module. I created them to run as Vagrant with VirtualBox as a hypervisor, so they unfortunately can not be used in the project infrastructure. The code is included here: B.

5.3 Jenkins

Jenkins is what I spent the majority of my time working with.

5.3.1 Jenkins Configuration as Code

I wanted my control-repo to define my whole Jenkins setup, so I went with using the Jenkins Configuration as Code plugin for configuration. It is a very new project, so documentation is sparse. The idea behind it is to mirror the web UI setup with yaml code, but the mapping is not one-to-one, so trying to set it up without help is like shooting in the dark. After struggling with setting it up from scratch myself, I found a good demo[12] and based my setup around it.

The plugin does have functionality to export yaml code based on a running Jenkins instance. This does not seem very well fleshed out yet, but when this is in place creating the configuration will be a much less painful process. Once JCasC matures I believe it will be massively useful, but currently using it is pretty cumbersome.

My "jenkins.yaml" file is included as an appendix: C. It sets up an admin user, configures the credentials plugin and adds a bunch of seed jobs.

5.3.2 Job DSL

I used the Job DSL plugin [29] to define "seed jobs", which are jobs used to initialize larger jobs found elsewhere. All my testing code was written in Jenkinsfiles, which I'll discuss next. Here is an example of such a "seed job", taken from my JCasC configuration file:

```
jobs:
(...)
  # sets up control-repo testing
  - script: |
      pipelineJob("control repo") {
                  definition {
                      cpsScm {
                          scm {
                              git {
                                  remote {
                                      github('tholok97/imt3005-project-cicd', 'ssh')
                                      credentials('tholok97_imt3005-project-cicd_deploy-key')
                                  }
                              }
                          }
                      }
                  }
      }
```

Line 1 defines a list of jobs, and line 5 to 18 is a ”seed job” that sets up a pipeline for the control-repo. The ”credentials” call on line 12 refers to a private ssh key stored using the credentials plugin. See the security discussion.

An issue worth mentioning is how to invoke these jobs. Continuous Integration is based around integrating each change you make into the main trunk, so triggering a job run on each push is desired. This is not possible in Skyhigh, as the floating IP addresses you are given are internal to NTNU's network. Therefore I've been manually triggering job runs during this project. I could also have had the jobs run on an interval, but as the course book[58] states this is not good enough. For proper CI you would set up webhooks between Github (or whatever version control host you use) and the Jenkins server.

5.4 Test pipelines

Each pipeline has a ”seed job” that simply adds the job and makes the ”Jenkinsfile” found in each repository do the rest.

5.4.1 Control-repo

The control-repo's pipeline currently just does linting tests and deploys to production. With more time I would have written proper Beaker tests for it using the method described here: 5.2.3. The following code snippet describes it's Jenkinsfile, with the parts I did not have time to write added:

```
1   pipeline {
2     agent none
3     stages {
4       stage('puppet parser validate') {
5         agent {
6           // using dockerfile defined in jenkins_agents
7           dockerfile {
8             dir 'jenkins_agents/syntax_agent/'
9           }
10        }
11        steps {
12          sh '/opt/puppetlabs/bin/puppet parser validate --debug --verbose .'
13        }
14      }
15      stage('puppet-lint') {
16        agent {
17          // using dockerfile defined in jenkins_agents
18          dockerfile {
19            dir 'jenkins_agents/syntax_agent/'
20          }
21        }
22        steps {
23          sh '/usr/bin/puppet-lint --error-level all --fail-on-warnings .'
24        }
25      }
26
27      // BEGIN NOT IMPLEMENTED. Added for demonstration
28      stage() {
29        agent {
30          // using dockerfile defined in jenkins_agents
31          dockerfile {
32            dir 'jenkins_agents/vagrant_agent/'
33          }
34        }
35        steps {
36          sh './runVagrantBeakerTest.sh'
37        }
38      }
39      // END NOT IMPLEMENTED. Rest is present in control-repo
40
41      stage('r10k deploy') {
42        agent { label 'master' }
```

```
        steps {
          sh "ssh -o StrictHostKeyChecking=no root@manager.borg.trek 'r10k deploy environment -p --verbose'"
        }
      }
    }
  }
```

5.4.2 gossinbackup

While learning Beaker I set up basic acceptance tests for my gossinbackup module: B. These run fine, but use the Vagrant and VirtualBox solution, so I can not run them with my current setup on Skyhigh. Here is what the pipeline currently looks like, with the parts I did not get to implement added. The syntax checks are the same as the ones we did in lab task 7.

```
pipeline {
  agent {
    dockerfile {
      filename 'pdk_agent_dockerfile'
      dir 'jenkins_agents'
    }
  }
  stages {
    stage('Validate and lint') {
      steps {
        sh 'pdk validate metadata,puppet'
      }
    }
    stage('Unit tests') {
      steps {
        sh 'pdk test unit --debug'
      }
    }

    // BEGIN NOT IMPLEMENTED
    stage('Acceptance test') {
      steps {
        sh 'bundle install && bundle exec rspec spec/acceptance/'
        // OR
        sh './runVagrantBeakerTest.sh'
      }
    }
    // END NOT IMPLEMENTED
  }
}
```

As indicated in the snippet, the acceptance tests could either run as standard beaker test runs, or by using the Vagrant solution I explain here: A. The first solution works, but not in Skyhigh, and the latter is not implemented. Both the unit and acceptance tests run fine in DigitalOcean.

5.4.3 Application

Because of time-constraints I did not get to implement any kind of CI/CD application pipeline. I did look into how I would go forward with doing it though, and set up a pipeline through the web UI that ultimately deployed a Golang[20] binary to the application server following a tutorial[21]. Here are the steps I would take to implement this in the control-repo:

1. Write a basic Golang application with a few tests. This is **trivial** to do in Golang.
2. Set up a seed job for the repository, following the same convention used for the control-repo and the gossinbackup module.
3. Set up a Jenkinsfile in the repository, matching the job created in the tutorial I followed [21].

4. Deploy to the application server using the same basic ssh method used in the control-repo pipeline.

5.5 imt3005-vagrant-vm

Although this was not the focus of this project, I will briefly discuss the Vagrant environment I created to work with this course [46]. Currently it includes all the tools needed, and I believe it is general enough that anyone can clone it and start using it themselves. Environment-specific configuration is done through a yaml file, and all the "data" of the environment is shared with the underlying host, so the environment is completely disposable. It gave me a chance to experiment freely, as I could simply rebuild it any time I messed up.

Relevant to this project is the fact that since I've done all of my work inside this environment, it should be possible for anyone to clone it down and get the exact same setup that I've used. This is one solution to the "it works on my machine" issue.

I have also tried having Vagrant provision it's VM in Skyhigh, which allows it to use the exact same image as production does. You could take this even further by having Vagrant apply the same boot script as the stack instances use as well to it as well. The code for this is not included, as it's just a variation of A combined with the Vagrantfile for the environment.

5.6 Jenkins Docker agents

Jenkins can run it's tests within agents, and with the Docker plugin installed these agents can be Docker containers. I found this very useful as each repository can define it's own agent along with it's Jenkinsfile. This way only the seed job is defined by Jenkins itself, with the rest being defined in the repository the tests concern. For the control-repo I created one such agent [13], which has puppet agent and puppet-lint installed to be able to run `puppet parser validate` and `puppet-lint`. If I later decide to add other tests, say a yaml linting tool, I don't have to touch Jenkins itself, only code within the control-repo.

I have Jenkins running in a Docker container, and to avoid "Docker-in-Docker" problems I have it connected to the underlying Docker host for creation of containers. [49]

I decided on a directory structure where additional agents can be added with ease. For example the control-repo would need an agent with Vagrant installed for the acceptance testing method I discussed here: 5.2.3, and the application (discussed here: 5.4.3) would need an agent with Golang tools installed. See my control-repo [13] for an example of this directory structure.

6 Security aspects

Security was not a primary focus of this project, so there are a few concerns that have not been fixed. I will discuss them here:

6.1 Secrets

Handling secrets was not a focus during this project, so I have chosen some dodgy solutions here and there. The only way of safely bringing in secrets with my setup is during provisioning of the Heat stack, by injecting them into the machines from Heat variables. The ssh key pair between the Jenkins server and the Puppet master is injected in this way, as well as the Github ssh key the Puppet master uses for r10k deploys. With access to my development machine an attacker would have access to every secret in my infrastructure, which I would consider a huge security issue for larger deployments. A solution here would be to generate the key pair between the Puppet master and Jenkins server either during stack creation or through Puppet.

JCasC lets you define user passwords in plaintext in the yaml configuration file, so I chose to do that for simplicity. This is an obvious security flaw that could be fixed by the below list of solutions.

The above problems are way smaller than the greatest sin I did during this project though, which was to include a private key in plaintext in the control-repo. The Jenkins server needs to authenticate with Github to clone down repositories for testing, and I did not find any straight-forward way to give Jenkins a private key for this, so I ended up just including the key as part of it's JCasC configuration. The key is a "Github deploy key"[19], however, so an attacker with access to this key would only have **read** access to **the repository the key was linked to**. I can think of a couple potential solutions to this issue:

- **Using JCasC secrets**: JCasC includes some basic functionality for handling secrets. One is injecting secrets into the configuration from environment variables. I could have set up environment variables during stack creation and included the deploy key this way
- **Using hiera-eyaml**: This would allow the key to be stored in the repository in an encrypted form. A decryption key could be given to the Puppet master during stack creation.
- **Using an external secrets technology**: There lots of technologies for keeping secrets out of version control altogether. I did not look into any of them.

6.2 SSH keys

Currently the Puppet master has ssh access to both of the other machines. This is for more easily debugging the infrastructure, but does mean that access to the Puppet master means access to everything. The Jenkins server also has ssh access back to the Puppet master. This is because it runs `r10k deploy` through ssh to deploy after testing. This was done for simplicity. A more secure solution would be to set

up an API endpoint on the Puppet master that runs `r10k deploy` itself when it receives request. That way access to the Jenkins machine only means an attacker could spam deploys, which is much better than an attacker having full access to the Puppet master through ssh. A simple socat one-liner [43] could do this.

6.3 Concerns around defining Jenkins as Code

Keeping all test definitions under version control means that an attacker with access to my repositories could see everything I test for, and more importantly, everything I **don't** test for. This is a consequence of having everything defined as code, and makes securing the repositories even more important. This issue is somewhat mitigated by the fact that the control-repo only contains "seed jobs". The actual test definitions are in each individual repository in the form of a "Jenkinsfile".

6.4 Infrastructure defined in control-repo

I chose to have the infrastructure definition in the control-repo, which means r10k "deploys" this code as well to the Puppet master. This means that with access to the Puppet master an attacker could find security holes in my setup.

7 Conclusions

My original goal was to experiment with CI/CD, and I feel I have accomplished that. I am left with a technical understanding of how CI/CD could be implemented using the technologies I have explored. Continuous Integration in my project exists in the form of the pipeline jobs running tests on changes made (although running them automatically was not possible, as I explained). With the features I explored, but didn't implement, I would feel pretty safe inviting others to work on the codebase as well.

I feel I skipped over Continuous Delivery and went straight to Continuous Deployment in this project, so the CD part of CI/CD I have still not gotten my head completely around.

Beaker turned out to be difficult to work with in Skyhigh. Most examples are of using Beaker with Vagrant and VirtualBox, which is not viable in Skyhigh, so most of the time spent on Beaker was getting it to even run. The method I arrived at does work though, although it has it's drawbacks, and I did not have time to fully implement it.

As for the infrastructure I wanted to set up, I am not finished. I am convinced I would be able to do it with more time however, and I have tried to lay out how I would achieve this in this report. What is implemented does meet my `openstack create` requirement though. Following this requirement has slowed me down quite a bit, but in return I have a greater understanding and respect for how difficult it can be to define everything in code, especially with regards to secrets.

References

[1] Beaker: Beaker Quick Start Tasks. [Online; accessed 4-November-2018]. 4

[2] Beaker Github page. [Online; accessed 4-November-2018]. 1

[3] Beaker issue tracker. [Online; accessed 4-November-2018]. 4

[4] Beaker-openstack code with problems. [Online; accessed 4-November-2018]. 5

[5] Beaker-openstack Github page. [Online; accessed 4-November-2018]. 1

[6] Beaker-puppet docs. [Online; accessed 4-November-2018].

[7] Beaker-vagrant Github page. [Online; accessed 4-November-2018]. 1

[8] BKR-1530: Beaker 4.0.0 does not work with Beaker-Puppet Latest version(1.3.0). [Online; accessed 4-November-2018].

[9] BKR-821: Beaker tries to install puppetlabs-release-xenial.deb which does not exist. [Online; accessed 4-November-2018].

[10] control-repo-cr Github page. [Online; accessed 4-November-2018]. 4

[11] Create a Continuous Deployment Pipeline with Golang and Jenkins. [Online; accessed 4-November-2018].

[12] Demo of JCasC in use. [Online; accessed 4-November-2018]. 6

[13] Directory for Jenkins Docker agents. [Online; accessed 4-November-2018]. 9

[14] Diving Deeper in to Beaker - Acceptance testing for Puppet. [Online; accessed 4-November-2018].

[15] Example of control-repo Jenkinsfile. [Online; accessed 4-November-2018].

[16] Example of Jenkins Docker container with it's own Docker installation connected to underlying Docker host. [Online; accessed 4-November-2018].

[17] Example of node yaml file used in P̈uppet Module Functional Testing with Vagrant, OpenStack and Beakerärticle. [Online; accessed 4-November-2018]. 3

[18] Getting started with Jenkins Configuration as Code. [Online; accessed 4-November-2018].

[19] Github blog post: Read-only deploy keys. [Online; accessed 4-November-2018]. 10

[20] Golang website. [Online; accessed 4-November-2018]. 8

[21] How to build on Jenkins and publish artifacts via ssh with Pipelines. [Online; accessed 4-November-2018]. 8

[22] IaC-heat-cr Github page. [Online; accessed 4-November-2018]. 4

[23] Jenkins-as-code: comparing job-dsl and Pipelines. [Online; accessed 4-November-2018].

[24] Jenkins Configuration as Code - first encounter! [Online; accessed 4-November-2018].

[25] Jenkins Configuration as Code: Automating an Automation Server. [Online; accessed 4-November-2018].

[26] Jenkins Configuration as Code website. [Online; accessed 4-November-2018]. 2

[27] Jenkins Job DSL Plugin documentation - pipelineJob. [Online; accessed 4-November-2018].

[28] jenkins4casc Github page. [Online; accessed 4-November-2018].

[29] job-dsl-plugin Github page. 6

[30] Onceover Github page. [Online; accessed 4-November-2018]. 3

[31] Onceover PR: Deprecated Beaker and removed dependency. [Online; accessed 4-November-2018]. 3

[32] OpenStack documentation: Continuous Integration. [Online; accessed 4-November-2018]. 3

[33] OpenStack documentation: Puppet Module Functional Testing. [Online; accessed 4-November-2018]. 3, 5

[34] OpenStack documentation: Testing. [Online; accessed 4-November-2018].

[35] OSDC 2016 - Introduction to Testing Puppet Modules by David Schmitt. [Online; accessed 4-November-2018].

[36] Pipeline as Code with Jenkins. [Online; accessed 4-November-2018].

[37] praqma/docker4jcasc Docker image. [Online; accessed 4-November-2018].

[38] PSICK Github page. [Online; accessed 4-November-2018]. 2

[39] Puppet Module Functional Testing with Vagrant, OpenStack and Beaker. [Online; accessed 4-November-2018]. 3, 5

[40] puppet-openstack-integration Github Page. [Online; accessed 4-November-2018].

[41] puppetlabs/beaker-puppet˙install˙helper issue: Getting an error with beaker 4.0. [Online; accessed 4-November-2018].

[42] R. Tyler Croy of Jenkins on infrastructure as code & testing Thank you. [Online; accessed 4-November-2018].

[43] StackOverflow: How to listen to webhooks from bash script? 11

[44] StackOverflow: Is it possible to Git merge / push using Jenkins pipeline. [Online; accessed 4-November-2018].

[45] The Careful Puppet Master: Reducing risk and fortifying acceptance testing with Jenkins CI. [Online; accessed 4-November-2018].

[46] tholok97/imt3005-vagrant-vm Github page. [Online; accessed 4-November-2018]. 9

[47] Top 10 Best Practices for Jenkins Pipeline Plugin. [Online; accessed 4-November-2018].

[48] Use Onceover to start testing with rspec-puppet. [Online; accessed 4-November-2018].

[49] Using Docker-in-Docker for your CI or testing environment? Think twice. [Online; accessed 4-November-2018]. 9

[50] Using Docker with Pipeline. [Online; accessed 4-November-2018].

[51] vagrant-openstack-provider Github page. [Online; accessed 4-November-2018]. 2

[52] Vagrant website. [Online; accessed 4-November-2018]. 2

[53] Acceptance testing on a control-repository using beaker with vagrant and docker, 2014. [Online; accessed 4-November-2018].

[54] Beaker 101: Cloud-enabled, acceptance testing, 2014. [Online; accessed 4-November-2018].

[55] Beaker 101: Cloud-enabled, acceptance testing, 2016. [Online; accessed 4-November-2018].

[56] FOWLER, M. Continuous delivery. [Online; accessed 4-November-2018]. 1

[57] FOWLER, M. Continuous integration. [Online; accessed 4-November-2018]. 1

[58] MORRIS, K. *Infrastructure as Code: Managing Servers in the Cloud*. O'Reilly Media, 2016. 1, 7

Following is what I consider the most important pieces of code. The rest can be found in the repositories described in 4.

A Beaker inside Vagrant example

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

require 'vagrant-openstack-provider'

#
# This is quite the minimal configuration necessary
# to start an OpenStack instance using Vagrant on
# an OpenStack with Keystone v3 API
#
# NOTE: this example is heavily
# inspired by http://my1.fr/blog/puppet-module-functional-testing-with-vagrant-openstack-and-beaker/
#
Vagrant.configure('2') do |config|

  config.ssh.username = 'ubuntu'

  config.vm.provider :openstack do |os, ov|
    os.server_name                     = 'vagrant_machine_in_openstack'
    os.security_groups                 = [ 'default', 'linux' ]
    os.identity_api_version            = '3'
    os.openstack_auth_url              = 'https://api.skyhigh.iik.ntnu.no:5000/v3'
    os.project_name                    = '<PROJECTNAME>'
    os.user_domain_name                = 'NTNU'
    os.project_domain_name             = 'NTNU'
    os.username                        = '<USERNAME>'
    os.password                        = '<PASSWORD>'
    os.region                          = 'SkyHiGh'
    os.floating_ip_pool                = 'ntnu-internal'
    os.floating_ip_pool_always_allocate = true
    os.flavor                          = 'm1.small'
    os.image                           = 'Ubuntu Server 16.04 LTS (Xenial Xerus) amd64'
    os.networks                        = [ '<INTERNALNETID>' ]

    ov.nfs.functional = false
  end

  # you could provision this machine using the same provisioning scripts used by
  # Heat, to create an exact duplicate
  config.vm.provision "shell", path: "bootscriptFromHeat.sh"


  # shell to install beaker, setup ssh, and run beaker tests.
  # written inline for sake of example
  config.vm.provision "shell", inline: <<-SHELL
    #!/bin/bash

    # install deps
    sudo apt-get update
    sudo apt-get install -y libxml2-dev libxslt-dev zlib1g-dev git ruby ruby-dev build-essential

    # prepare ssh
    echo "" | sudo tee -a /etc/ssh/sshd_config
    echo "Match address 127.0.0.1" | sudo tee -a /etc/ssh/sshd_config
    echo "    PermitRootLogin without-password" | sudo tee -a /etc/ssh/sshd_config
    echo "" | sudo tee -a /etc/ssh/sshd_config
    echo "Match address ::1" | sudo tee -a /etc/ssh/sshd_config
    echo "    PermitRootLogin without-password" | sudo tee -a /etc/ssh/sshd_config
    mkdir -p .ssh
    ssh-keygen -f ~/.ssh/id_rsa -b 2048 -C "beaker key" -P ""
    sudo mkdir -p /root/.ssh
    sudo rm /root/.ssh/authorized_keys
    cat ~/.ssh/id_rsa.pub | sudo tee -a /root/.ssh/authorized_keys
    sudo service ssh restart

    # prepare gems
    # this uses my gossinbacup module as an example, but it would be
    # possible to have the module as a parameter to this process
    git clone https://github.com/tholok97/gossinbackup
    cd gossinbackup
    sudo gem install bundler --no-rdoc --no-ri --verbose
    bundle install

    # run tests
```

```
    # this relies on SUT yaml definitions with hyporvisor set to "none",
    # like here: https://github.com/openstack/puppet-keystone/blob/master/spec/acceptance/nodesets/nodepool-xenial.yml
    export BEAKER_debug=yes
    bundle exec rspec spec/acceptance
SHELL

end
```

B Beaker tests for my gossinbackup module

B.1 spec_helper_acceptance.rb

```
require 'beaker-pe'
require 'beaker-puppet'
require 'beaker-rspec'
require 'beaker/puppet_install_helper'
require 'beaker/module_install_helper'

# use helpers to prepare the hosts with puppet, this
# module and it's dependencies
run_puppet_install_helper
configure_type_defaults_on(hosts)
install_module_on(hosts)
install_module_dependencies_on(hosts)

RSpec.configure do |c|
  c.formatter = :documentation

  c.before :suite do
    # Install module to all hosts
    hosts.each do |host|
      # this module uses git as a provider. Installing git on the hosts
      install_module_from_forge('puppetlabs-git', '0.5.0')
      apply_manifest_on(host, 'include git', catch_failures: true)
    end
  end
end
```

B.2 class_spec.rb

```
require 'spec_helper_acceptance'

describe 'gossinbackup module' do
  let(:pp) { 'class { "gossinbackup": source_directory => "/home", destination_directory => "/backups", }' }

  context 'when manifest is applied' do
    it 'gives no errors' do
      apply_manifest(pp, catch_failures: true) do |r|
        expect(r.stderr).not_to match(%r{error}i)
        expect(r.exit_code).to eq(2)
      end
    end
  end

  context 'when manifest is applied a second time' do
    it 'does not change anything (is idempotent)' do
      apply_manifest(pp, catch_failures: true) do |r|
        expect(r.stderr).not_to match(%r{error}i)
        expect(r.exit_code).to be_zero
      end
    end
  end
end
```

C Jenkins.yaml

```
# JCasC configuration file. Heavily inspired by https://github.com/Praqma/praqma-jenkins-casc
jenkins:
  systemMessage: "Greetings friend! I am a system message configured through JCasC"
  agentProtocols:
    - "JNLP4-connect"
  securityRealm:
```

```
      local:
        allowsSignup: false
        users:
         - id: insecureAdmin
           password: insecurePassword
    authorizationStrategy:
      globalMatrix:
        grantedPermissions:
          - "Overall/Read:anonymous"
          - "Job/Read:anonymous"
          - "View/Read:anonymous"
          - "Overall/Administer:authenticated"
    crumbIssuer: "standard"
  credentials:
    system:
      domainCredentials:
        - credentials:
            - basicSSHUserPrivateKey:
                scope: GLOBAL  # this has to be global for scm to work in seed jobs
                id: tholok97_imt3005-project-cicd_deploy-key
                username: tholok97
                passphrase:  ""
                description: "ssh private key used to connect ssh slaves"
                privateKeySource:
                  directEntry:
                    # YES. Including a private key in plaintext in VCS is horrible, but did it
                    # for simplicity. See my report for a discussion for possible solutions
                    #
                    # Note that is a Github deploy key, so it only has READ access to ONE
                    # repository, which I'll make public eventually anyways.
                    privateKey: |
                      -----BEGIN RSA PRIVATE KEY-----
                      <github deploy key was stored here. left out for brevity. see
                      security discussion>
                      -----END RSA PRIVATE KEY-----

  jobs:
  # taken from https://github.com/jenkinsci/configuration-as-code-plugin/blob/master/docs/seed-jobs.md
  # and https://marcesher.com/2016/08/04/jenkins-as-code-comparing-job-dsl-and-pipelines/
  #
  # The following three jobs are just here to show some examples of job dsl configured through JCasC
    - script: |
        def myJob = freeStyleJob('SimpleJob')
        myJob.with {
            description 'A Simple Job'
        }
    - script: |
        job('example-job-from-job-dsl') {
            scm {
                github('jenkinsci/job-dsl-plugin', 'master')
            }
            triggers {
                cron("@hourly")
            }
            steps {
                shell("echo 'Hello World'")
            }
        }
    - script: |
        pipelineJob("pipeline-calls-other-pipeline") {
                    logRotator{
                        numToKeep 30
                    }
                    definition {
                        cps {
                            sandbox()
                            script("""
                                node {
                                    echo 'Hello World 1'
                                }
                            """.stripIndent())
                            }
                        }
                    }
  # The below job sets up a pipeline job for my gossinbackup module. The Jenkinsfile of that module is used.
  # Added as a demonstration that jobs work
  # Reference used: https://jenkinsci.github.io/job-dsl-plugin/#path/pipelineJob
    - script: |
        pipelineJob("pipeline-job-to-run-gossinbackup") {
                    definition {
                        cpsScm {
                            scm {
                                git('https://github.com/tholok97/gossinbackup', 'master')
```

```
                        }
                    }
                }
        }
    # sets up control-repo testing
      - script: |
          pipelineJob("control repo") {
                    definition {
                        cpsScm {
                            scm {
                                git {
                                    remote {
                                        github('tholok97/imt3005-project-cicd', 'ssh')
                                        credentials('tholok97_imt3005-project-cicd_deploy-key')
                                    }
                                }
                            }
                        }
                    }
        }
    # These jobs are from the demo I copied this config from. Left in for reference
    #  - url: https://raw.githubusercontent.com/Praqma/job-dsl-collection/master/configuration-as-code-dsl/pipeline.dsl #casc
    #  - url: https://raw.githubusercontent.com/Praqma/memory-map-plugin/master/jenkins-pipeline/pipeline.groovy #memory map
    #  - url: https://raw.githubusercontent.com/Praqma/codesonar-plugin/master/jenkins-pipeline/pipeline.groovy #codesonar
    #  - url: https://raw.githubusercontent.com/Praqma/pretested-integration-plugin/master/jenkins-pipeline/pipeline.groovy
     ↪    #pretested integration
    #  - url: https://raw.githubusercontent.com/Praqma/ClearCaseUCMPlugin/master/jenkins-pipeline/pipeline.groovy #ccucm
    #  - url: https://raw.githubusercontent.com/Praqma/Praqmatic-Automated-Changelog/master/jenkins-pipeline/pipeline.groovy
     ↪    #pac
    #  - url: https://raw.githubusercontent.com/Praqma/PlusBump/rebirth/jenkins-pipeline/pipeline.groovy #plusbump
    #  - url: https://raw.githubusercontent.com/Praqma/job-dsl-collection/master/web-pipeline-dsl/web_pipeline_dsl.groovy
     ↪    #websites
    #  - file: ./jobdsl/rut.groovy #rut (private repo...better keep it here)

    tool:
      git:
        installations:
          - name: Default
            home: "git"

    security:
      remotingCLI:
        enabled: false
```

D OpenStack Heat infrastructure definition

D.1 topology.yaml

```
heat_template_version: 2013-05-23

description: >
  HOT template to create a new neutron network plus a router to the public
  network, and for deploying three servers into the new network. The template also
  assigns floating IP addresses to each server so they are routable from the
  public network. This creates the basic borg.trek infrastructure with a
  manager (Ubuntu), a monitor (Ubuntu) and a DNS server (Windows Server)

parameters:
  key_name:
    type: string
    description: Name of keypair to assign to servers
  image_linux:
    type: string
    description: Name of image to use for servers
    default: Ubuntu Server 18.04 LTS (Bionic beaver) amd64
  flavor_manager:
    type: string
    description: Flavor to use for servers
    default: m1.large
  public_net:
    type: string
    description: >
      ID or name of public network for which floating IP addresses will be allocated
    default: ntnu-internal
  internal_net_name:
    type: string
    description: Name of internal network to be created
    default: internal_net
  internal_net_cidr:
```

```
      type: string
      description: Internal network address (CIDR notation)
      default: 192.168.190.0/24
    internal_net_gateway:
      type: string
      description: Internal network gateway address
      default: 192.168.190.1
    internal_net_pool_start:
      type: string
      description: Start of admin network IP address allocation pool
      default: 192.168.190.100
    internal_net_pool_end:
      type: string
      description: End of internal network IP address allocation pool
      default: 192.168.190.199
    sec_group_linux:
      type: comma_delimited_list
      description: Security groups
      default: default,linux
    github_ssh_private_key:
      type: string
      description: >
        Private key used to authenticate with Github for access to private repositories
      default: |
        no key here!!
    jenkins_ssh_private_key:
      type: string
      description: >
        Private key used to authenticate with master from Jenkins
      default: |
        no key here!!
    jenkins_ssh_public_key:
      type: string
      description: >
        Public key placed in master to allow ssh from Jenkins
      default: |
        no key here!!

  resources:

    internal_net:
      type: OS::Neutron::Net
      properties:
        name: { get_param: internal_net_name }

    internal_subnet:
      type: OS::Neutron::Subnet
      properties:
        network_id: { get_resource: internal_net }
        cidr: { get_param: internal_net_cidr }
        gateway_ip: { get_param: internal_net_gateway }
        allocation_pools:
          - start: { get_param: internal_net_pool_start }
            end: { get_param: internal_net_pool_end }

    router:
      type: OS::Neutron::Router
      properties:
        external_gateway_info:
          network: { get_param: public_net }

    router_interface_internal:
      type: OS::Neutron::RouterInterface
      properties:
        router_id: { get_resource: router }
        subnet_id: { get_resource: internal_subnet }

    manager:
      type: OS::Nova::Server
      properties:
        name: manager
        image: { get_param: image_linux }
        flavor: { get_param: flavor_manager }
        key_name: { get_param: key_name }
        networks:
          - port: { get_resource: manager_port }
        user_data_format: RAW
        user_data:
          str_replace:
            template: { get_file: lib/manager_boot.bash }
            params:
              openstack_heat_strreplace_github_ssh_private_key: { get_param: github_ssh_private_key }
              imt3005_tholok_project_cicd_jenkins_public_key: { get_param: jenkins_ssh_public_key }
```

```
115
116     manager_port:
117       type: OS::Neutron::Port
118       properties:
119         network_id: { get_resource: internal_net }
120         security_groups: { get_param: sec_group_linux }
121         fixed_ips:
122           - subnet_id: { get_resource: internal_subnet }
123
124     manager_floating_ip:
125       type: OS::Neutron::FloatingIP
126       properties:
127         floating_network: { get_param: public_net }
128         port_id: { get_resource: manager_port }
129
130
131     app:
132       type: lib/managed_linux_server_one_nic.yaml
133       properties:
134         key_name:            { get_param: key_name }
135         server_name:         app
136         image:               { get_param: image_linux }
137         flavor:              m1.medium
138         sec_group_linux:     { get_param: sec_group_linux }
139         public_net:          { get_param: public_net }
140         admin_net:           { get_resource: internal_net }
141         admin_subnet:        { get_resource: internal_subnet }
142         dns_ip:              { get_attr: [manager, networks, get_param: internal_net_name, 0] }
143         manager_ip:          { get_attr: [manager, networks, get_param: internal_net_name, 0] }
144         jenkins_ssh_private_key: { get_param: jenkins_ssh_private_key }
145
146     jenkins:
147       type: lib/managed_linux_server_one_nic.yaml
148       properties:
149         key_name:            { get_param: key_name }
150         server_name:         jenkins
151         image:               { get_param: image_linux }
152         flavor:              m1.large
153         sec_group_linux:     { get_param: sec_group_linux }
154         public_net:          { get_param: public_net }
155         admin_net:           { get_resource: internal_net }
156         admin_subnet:        { get_resource: internal_subnet }
157         dns_ip:              { get_attr: [manager, networks, get_param: internal_net_name, 0] }
158         manager_ip:          { get_attr: [manager, networks, get_param: internal_net_name, 0] }
159         jenkins_ssh_private_key: { get_param: jenkins_ssh_private_key }
```

D.2 managed_linux_server_one_nic.yaml

```
1   heat_template_version: 2013-05-23
2
3   description: >
4     HOT template for a GNU/Linux server with a nic in the admin network
5     and a nic in another network. A floating IP will be associated with
6     the IP in the other network (not the admin network).
7
8   parameters:
9     key_name:
10      type: string
11      description: Name of keypair to assign to servers
12    server_name:
13      type: string
14      description: Name of linux server
15    image:
16      type: string
17      description: Name of image to use for servers
18      default: Ubuntu Server 18.04 LTS (Bionic Beaver) amd64
19    flavor:
20      type: string
21      description: Flavor to use for servers
22    sec_group_linux:
23      type: comma_delimited_list
24      description: Security groups
25    public_net:
26      type: string
27      description: >
28        ID or name of public network for which floating IP addresses will be allocated
29    admin_net:
30      type: string
31      description: UUID of admin net created in base template (iac_admin_net)
32    admin_subnet:
33      type: string
```

```
      description: UUID of admin subnet created in base template (192.168.180.0/24)
    dns_ip:
      type: string
      description: IP address of dns server created in base template
    manager_ip:
      type: string
      description: IP address of server manager created in base template
    jenkins_ssh_private_key:
      type: string
      description: >
        Private key used to authenticate with master from Jenkins
      default: |
        no key here!!

resources:

  server:
    type: OS::Nova::Server
    properties:
      name: { get_param: server_name }
      image: { get_param: image }
      flavor: { get_param: flavor }
      key_name: { get_param: key_name }
      networks:
        - port: { get_resource: server_port_admin }
      user_data_format: RAW
      user_data:
        str_replace:
          template: { get_file: managed_linux_server_one_nic.bash }
          params:
            manager_ip_address: { get_param: manager_ip }
            dns_ip_address: { get_param: dns_ip }
            imt3005_tholok_project_cicd_jenkins_private_key: { get_param: jenkins_ssh_private_key }

  server_port_admin:
    type: OS::Neutron::Port
    properties:
      network_id: { get_param: admin_net }
      security_groups: { get_param: sec_group_linux }
      fixed_ips:
        - subnet_id: { get_param: admin_subnet }

  server_floating_ip:
    type: OS::Neutron::FloatingIP
    properties:
      floating_network: { get_param: public_net }
      port_id: { get_resource: server_port_admin }
```

D.3 managed_linux_server_one_nic.bash

```
        #!/bin/bash -v
tempdeb=$(mktemp /tmp/debpackage.XXXXXXXXXXXXXXXXX) || exit 1
wget -O "$tempdeb" https://apt.puppetlabs.com/puppet5-release-bionic.deb
dpkg -i "$tempdeb"
apt-get update
apt-get -y install puppet-agent
echo "$(ip a | grep -Eo 'inet ([0-9]*\.){3}[0-9]*' | tr -d 'inet ' | grep 192.168.180) $(hostname).borg.trek $(hostname)"
  ↪    >> /etc/hosts
echo "manager_ip_address manager.borg.trek manager" >> /etc/hosts
/opt/puppetlabs/bin/puppet resource service puppet ensure=stopped enable=true
/opt/puppetlabs/bin/puppet config set server manager.borg.trek --section main
/opt/puppetlabs/bin/puppet config set runinterval 300 --section main
/opt/puppetlabs/bin/puppet agent -t # request certificate
/opt/puppetlabs/bin/puppet agent -t # configure
/opt/puppetlabs/bin/puppet resource service puppet ensure=running enable=true
cat <<EOF >> /etc/netplan/50-cloud-init.yaml
            nameservers:
                search: [borg.trek]
                addresses: [dns_ip_address]
EOF
netplan apply
# set up public key for jenkins as an authorized host
# (this is set up for app as well, but don't have time to fix)
cat <<EOF >> /home/ubuntu/.ssh/id_rsa
imt3005_tholok_project_cicd_jenkins_private_key
EOF
chmod 600 /home/ubuntu/.ssh/id_rsa
chown ubuntu /home/ubuntu/.ssh/id_rsa
```

D.4 manager_boot.sh

```
#!/bin/bash -v
tempdeb=$(mktemp /tmp/debpackage.XXXXXXXXXXXXXXXXX) || exit 1
wget -O "$tempdeb" https://apt.puppetlabs.com/puppet5-release-bionic.deb
dpkg -i "$tempdeb"
apt-get update
apt-get -y install puppetserver pwgen
/opt/puppetlabs/bin/puppet resource service puppet ensure=stopped enable=true
/opt/puppetlabs/bin/puppet resource service puppetserver ensure=stopped enable=true
# configure puppet agent, and puppetserver autosign
/opt/puppetlabs/bin/puppet config set server manager.borg.trek --section main
/opt/puppetlabs/bin/puppet config set runinterval 300 --section main
/opt/puppetlabs/bin/puppet config set autosign true --section master
# keys for hiera-eyaml TBA...
# set up public key for jenkins as an authorized host
# ALLOWING LOG IN AS ROOT FOR SIMPLICITY
cat <<EOF > /root/.ssh/authorized_keys
imt3005_tholok_project_cicd_jenkins_public_key
EOF
# setup git ssh key
## add host entry to .ssh/config
cat <<EOF > /root/.ssh/config
Host github.com
    StrictHostKeyChecking no
    IdentityFile /root/.ssh/imt3005_tholok_project_cicd_key
EOF
## add github key taken from Heat parameter
echo "openstack_heat_strreplace_github_ssh_private_key" > /root/.ssh/imt3005_tholok_project_cicd_key
## chmod the key to make ssh happy
chmod 600 /root/.ssh/imt3005_tholok_project_cicd_key
# r10k and control-repo:
/opt/puppetlabs/bin/puppet module install puppet-r10k
cat <<EOF > /var/tmp/r10k.pp
class { 'r10k':
  version => '2.6.4',
  sources => {
    'puppet' => {
      'remote'  => 'git@github.com:tholok97/imt3005-project-cicd.git',
      'basedir' => '/etc/puppetlabs/code/environments',
      'prefix'  => false,
    },
  },
}
EOF
/opt/puppetlabs/bin/puppet apply /var/tmp/r10k.pp
r10k deploy environment -p
cd /etc/puppetlabs/code/environments/production/
bash ./new_keys_and_passwds.bash
## temp fix to have correct fqdn for puppet, before permanent fix in dhclient
## (which requires reboot)
echo "$(ip a | grep -Eo 'inet ([0-9]*\.){3}[0-9]*' | tr -d 'inet ' | grep -v '^127') $(hostname).borg.trek $(hostname)" >>
 ↪   /etc/hosts
/opt/puppetlabs/bin/puppet resource service puppetserver ensure=running enable=true
/opt/puppetlabs/bin/puppet agent -t # request certificate
/opt/puppetlabs/bin/puppet agent -t # configure manager
/opt/puppetlabs/bin/puppet agent -t # once more to update exported resources
/opt/puppetlabs/bin/puppet resource service puppet ensure=running enable=true
# permanent fix for domainname
cat <<EOF >> /etc/netplan/50-cloud-init.yaml
            nameservers:
                search: [borg.trek]
                addresses: [127.0.0.1]
EOF
netplan apply

#wc_notify --data-binary '{"status": "SUCCESS"}'
```