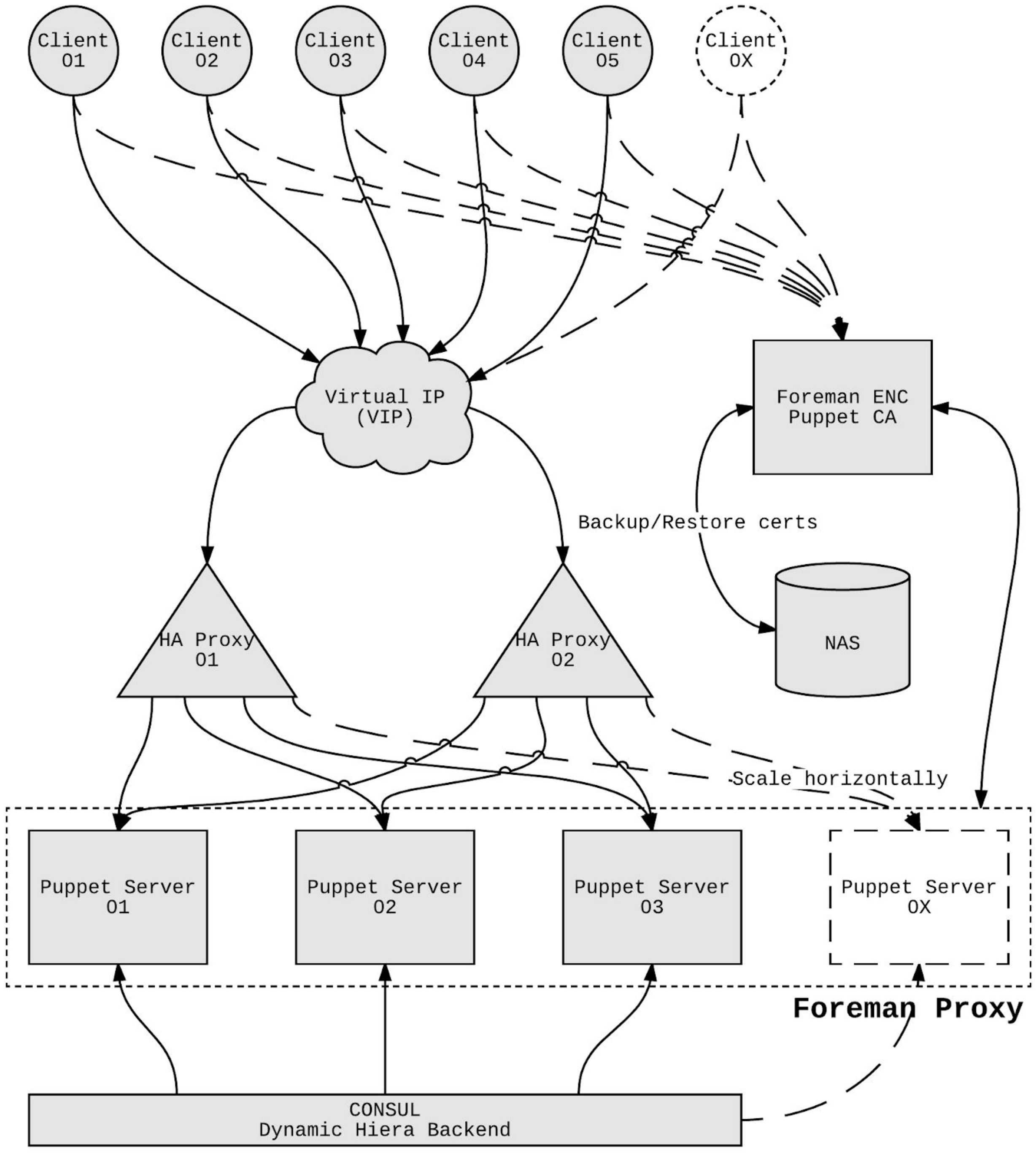
Puppet Architecture

This documentation provides an architecture overview of Puppet 4. It will detail highly available puppet infrastructure, deploying modules, handling secrets, developer testing, unit and acceptance testing, style guide, insight into puppet logs and other cool stuff. Our goal is not just to upgrade Puppet but make it scalable, reliable and self-serviceable.



Puppet 4 infrastructure consists of 2-3 servers. These servers will be fronted by a pair of HAProxy servers. The HAProxy servers will be configured with VIP so if one of the HAProxy nodes goes offline the other node can pick up. If we need to scale more puppet servers can be added to the pool.

There will be 1 certificate authority (CA) on the Foreman node. The certs on the CA will be backed up nightly to a NAS. The disaster recovery plan for the CA is to spin up a new CA and download the backed up certs. Please note no new nodes can be added or rebuilt while the CA server is down.

Please note the above architecture is *per* datacenter.

Server Roles

A server role should include all classes at the top level. By looking at the role we should be able to determine everything needed to build the node without too much conditional logic. If such logic is needed it should be moved to a wrapper module and then include the wrapper module in the class; by wrapping we are two levels deep. Going any deeper is not recommended since you have to open multiple files and sleuth through the codebase to understand what a role accomplishes.

Keep the number of classes per role to a handful, and to make it easy to read, configure all params via Hiera without using `class`.

Let's take redis for example, it would look something like this.

```
class roles::site_redis {
  include ccm_os::centos
  include ccm_profile_redis::server
}
```

The `ccm_os::centos` class is included in every centos role as it sets up basic things like networking, security patches, base packages, OS specific configuration etc. on every node in the infrastructure. Any parameters (for example # of redis instances) would be configured via Hiera.

By reading the role we learn it's a centos node installing Redis via wrapper module.

Modules and R10K

In Puppet 4 we can use R10K to manage upstream modules alongside our in-house local modules. Currently everything is in a monolithic repo, by using R10K we only need modules we've written or patched locally, the rest can be pulled from Puppet Forge (https://forge.puppet.com/).

Naming: While most modules can be pulled from Forge, business specific modules are be written in-house. The module names cannot be the same as the upstream module so we needed to come up with a specific format for naming modules.

- Modules with CCM business logic will be prefixed with **ccm_**.
- Modules which build on top of Forge modules will be prefixed with **ccm_profile_**. These are called wrapper modules.
- Modules written in-house with the intent of open sourcing should not have any prefix and should be deployed to Forge when ready to use.

R10K Puppetfile for our example above:

```
# cat Puppetfile

forge "http://forge.puppetlabs.com"

mod 'arioch-redis', '1.2.4'
mod 'ccm-nagflux', '0.1.0'

mod 'ccm_os',
:git => 'https://git.ccmteam.com/scm/sm/ccm_os.git',
:tag => '0.5.8'

mod 'ccm_profile_redis',
:git => 'https://git.ccmteam.com/scm/sm/ccm_profile_redis.git',
:tag => '0.3.7'

mod 'nagflux',
:git => 'https://git.ccmteam.com/scm/sm/nagflux.git',
:tag => '0.1.0'
```

https://github.com/puppetlabs/r10k/blob/master/doc/puppetfile.mkd

Tagging/Versioning: As you can see in our example above we have tags for all of our modules. By tagging commits we know which version is deployed to a particular environment, we can send this metric to influx during R10K deploy. This could be immensely helpful when debugging production if we can correlate a particular version to a change in graph and if need be rollback to the older version.

The format of the versioning is loosely based on SemVer (http://semver.org/). We should have our own documented procedure as to when it's reasonable to bump major or minor versions, patch needs to be updated regardless on every change.

Environments

We should think of environment as infrastructure environment and not app environment (i.e. not DEV, PRD, QA etc). There will be 2 main infrastructure environments; CANARY and LIVE. Apart from these there would be ad-hoc environments created by developers to do ad-hoc testing. If other environments are needed (for module upgrades etc.) these can be deployed via R10K as well on a case by case basis.

Each environment will have its own Puppetfile.

https://en.wiktionary.org/wiki/canary_in_a_coal_mine

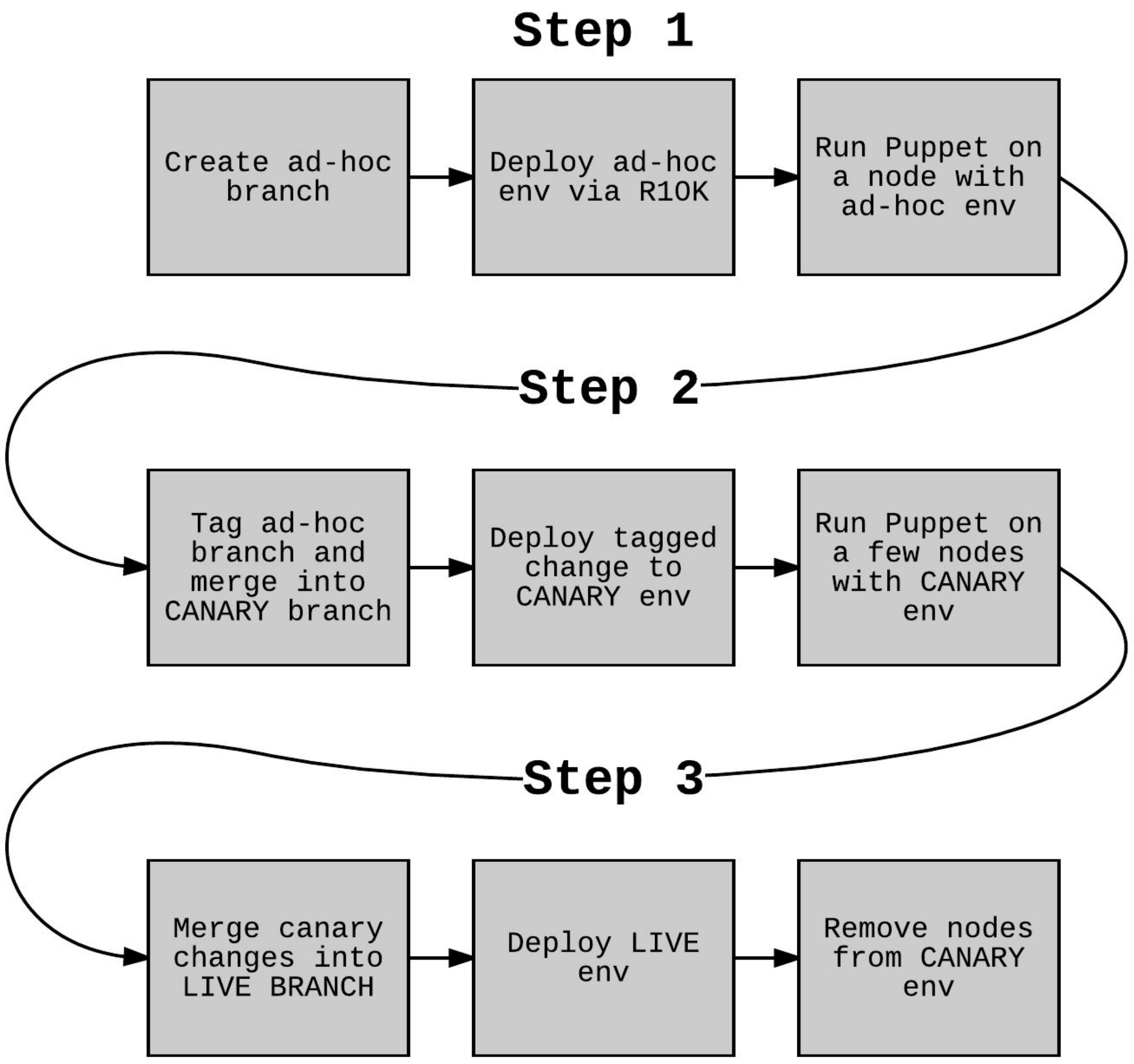
How to make a change
Lets walk through how deploying a change would look like.

Step 1
Create an ad-hoc branch prefixed with a Jira ticket: SM_5746_cool_school. This will be the puppet environment R10K will deploy to puppet servers. Once deployed pick 1 or 2 nodes and run puppet with SM_5746_cool_school for the ad-hoc changes to apply. Nodes should be removed from ad-hoc environments as soon as the ad-hoc testing is done to prevent them from going out of date. There will be a check to ensure this doesn't happen. Once you are happy with the change make a PR and let all tests pass (more on testing later). When the PR is approved merge and tag it.

Step 2
Using the tag created in the previous step open a PR for CANARY environment and update the tag in Puppetfile, merge the PR once

approved. At this point this merge will have no effect until a few nodes are in the canary env. Set the environment for 50% of the nodes to canary and monitor the rollout.

Step 3
Make one final PR with the same content from canary Puppetfile to LIVE Puppetfile. Once the PR is approved merge it and remove the other 50% nodes from canary because now live has the same changes.


Step 1
Create ad-hoc branch
Deploy ad-hoc env via R10K
Run Puppet on a node with ad-hoc env
Step 2
Tag ad-hoc branch and merge into CANARY branch
Deploy tagged change to CANARY env
Run Puppet on a few nodes with CANARY env
Step 3
Merge canary changes into LIVE BRANCH
Deploy LIVE env
Remove nodes from CANARY env

Dynamic Hiera

We could manually set the environment for each node when applying changes to ad-hoc and canary nodes, that's cumbersome, Consul and Etcd can help in this regard.

These are Key-Value data stores that Puppet can query for hiera data. Why do we want to do this? Well for one we only need one permanent environment; LIVE. Ad-hoc and Canary environments are temporary environments for a node. R10K can push code to these environments and we can dynamically set the key to override the environment via hierarchy.

This suddenly becomes really powerful because we don't need to have a static but a dynamic set of nodes. In other words we can do things like deploying a change to 30% of the nodes, then 60%, and if are happy deploy it 100% (or simple delete the override key).

Puppet killswitch: The KV data store can also be leveraged to stop puppet runs en masse. For example say we are rolling out a change from 50% -> 100% but we noticed something and want to stop puppet runs. Sure you can roll back but that takes time, you want to stop puppet runs on those nodes immediately. With a call to consul or mco you can disable puppet on those nodes.

Below is an example of the hiera.yaml would look like.

```
:backends:
  - consul
  - eyaml
  - yaml
```

https://forge.puppet.com/venmo/consulr
https://forge.puppet.com/lynxman/hiera_consul

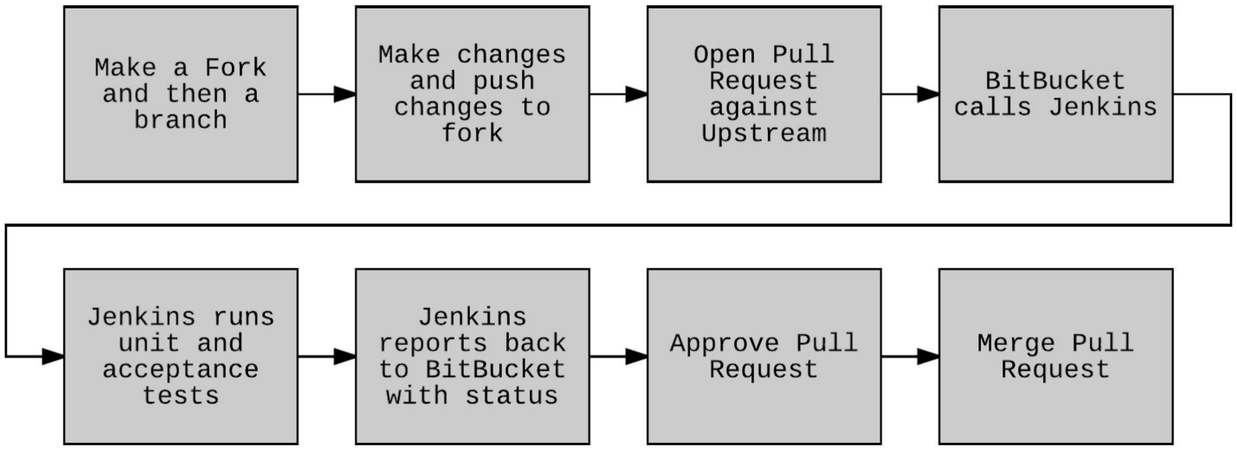
Developer Testing

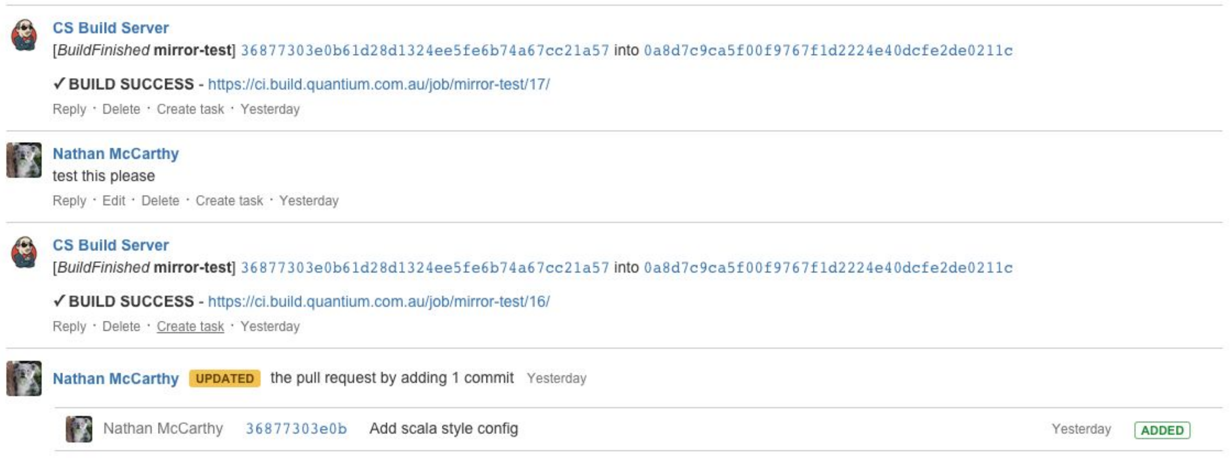
The resources in our DataCenter are finite, we only want to develop in the DC if we have specific hardware requirements. Most development can easily happen on our local laptops by using something like Vagrant/Docker. We can build our own images with the base puppet resources. The puppet agent on the developer VM will be talking to a puppet server built with the same specifications as live, launched in a separate container, so we can keep the client-server model of communication during development as well.

The images will be build using Packer and will be setup exactly like live environment. The user can also mimic various roles by modifying a YAML file and certain facts can be propagated up the the VM to modify a behavior.

Rspec, Kitchen, Jenkins

Rspec-puppet help write unit tests and Kitchen-puppet help write acceptance tests for puppet. Unit tests ensure the logic in our puppet manifest theoretically produces a desired outcome while acceptance step goes one step further and launches a VM/Container to test actual puppet code by applying the resources. This process can be automated via Jenkins-BitBucket plugin. This is where PRs are crucial. When the PR is made BitBucket will make a call to Jenkins and run all these tests. Once all tests run successfully Jenkins will report back with the status to BitBucket.


Make a Fork and then a branch
Make changes and push changes to fork
Open Pull Request against Upstream
BitBucket calls Jenkins
Jenkins runs unit and acceptance tests
Jenkins reports back to BitBucket with status
Approve Pull Request
Merge Pull Request



CS Build Server
[*BuildFinished* **mirror-test**] 36877303e0b61d28d1324ee5fe6b74a67cc21a57 into 0a8d7c9ca5f00f9767f1d2224e40dcfe2de0211c

✓ **BUILD SUCCESS** - https://ci.build.quantium.com.au/job/mirror-test/17/

Reply · Delete · Create task · Yesterday

Nathan McCarthy
test this please

Reply · Edit · Delete · Create task · Yesterday

CS Build Server
[*BuildFinished* **mirror-test**] 36877303e0b61d28d1324ee5fe6b74a67cc21a57 into 0a8d7c9ca5f00f9767f1d2224e40dcfe2de0211c

✓ **BUILD SUCCESS** - https://ci.build.quantium.com.au/job/mirror-test/16/

Reply · Delete · Create task · Yesterday

Nathan McCarthy UPDATED the pull request by adding 1 commit Yesterday

Nathan McCarthy 36877303e0b Add scala style config Yesterday ADDED

Puppet Reports in ELK

While puppet reports are sent to Foreman they are not really useful. Sure we can see the reports but we want to do more analysis on these reports. For example, we want to know when packages are being upgraded across nodes, when services are being restarted, which subset of nodes take longest to run, which nodes puppet runs are failing, etc. Best part is we can expose all these logs via Kibana or Grafana Dashboards.

Here is a good talk from Puppetconf 2015
https://www.slideshare.net/pkill/puppetconf-2015-puppet-reporting-with-elasticsearch-logstash-and-kibana

Handling Secrets

Currently all secrets are in plain sight. We would want to encrypt them before putting them in the repo. For this we can use Eyaml (Encrypted YAML) as a data source for Hiera. Eyaml works like regular YAML but it encrypts the data using one of two ways; GPG and PKCS.

- GPG requires everyone to have a unique private key. If one key gets compromised it's not a big deal, we just have to remove the user from the key ring. The overhead of educating and working with GPG is to be considered.
- PKCS has a single RSA key which everyone needs to have in order to encrypt the data. If the key gets compromised everyone needs to get the new key before they can decrypt again. Also keys need to be rotated quite often. On the upside it's relatively painless to setup and manage.

Style Guide

A style guide allow us to follow the same pattern across multiple modules. For example, if we want to be able to integrate with upstream modules sometimes it helps to design our own modules in a specific way. The style guide helps us with this and also the usual commas, semicolons, etc. Puppet has an official style guide but what works best is a combination of the Puppet style guide and our own in-house style guide that complements the upstream one.

https://docs.puppet.com/puppet/latest/style_guide.html

Follow up discussions

The following are items that are yet to be discussed.

- Revise Hiera hierarchies.
- Can we leverage Puppet Facts/Cert for node classification? How to fetch the role during Kickstart process?
- How would this architecture change in a Hybrid cloud environment?
- Monitor Puppet health, notices and alerts?