Scaling Agile Infrastructure to People

2015 J. Phys.: Conf. Ser. 664 022026

(http://iopscience.iop.org/1742-6596/664/2/022026)

Journal of Physics: Conference Series **664** (2015) 022026 doi:10.1088/1742-6596/664/2/022026

Scaling Agile Infrastructure to People

B Jones, G McCance, S Traylen and N Barrientos Arias

European Organization for Nuclear Research (CERN), 1211 Geneva 23, Switzerland

Ben.Dylan.Jones@cern.ch

Abstract. When CERN migrated its infrastructure away from homegrown fabric management tools to emerging industry-standard open-source solutions, the immediate technical challenges and motivation were clear. The move to a multi-site Cloud Computing model meant that the tool chains that were growing around this ecosystem would be a good choice, the challenge was to leverage them. The use of open-source tools brings challenges other than merely how to deploy them. Homegrown software, for all the deficiencies identified at the outset of the project, has the benefit of growing with the organization. This paper will examine what challenges there were in adapting open-source tools to the needs of the organization, particularly in the areas of multi-group development and security. Additionally, the increase in scale of the plant required changes to how Change Management was organized and managed. Continuous Integration techniques are used in order to manage the rate of change across multiple groups, and the tools and workflow for this will be examined.

1. Introduction

In response to wider changes in the IT industry, CERN adopted [1] industry standard open source tools to manage CERN's contribution to the computing infrastructure of the Worldwide LHC Computing Grid [2]. Products and projects such as OpenStack [3] and Puppet [4] were expected or known to be able to scale to the numbers of machines required by the Tier0. However, existing workflows and practices grew alongside homegrown toolsets. In fact some have suggested that by definition tools match organizational structure, the (in)famous Conway's Law [5] says "organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations". As we adopted Puppet in particular to manage the resources in the CERN Computer Centre we made changes both technical and procedural in order to get the most from the toolset for our wide range of Service and Infrastructure Managers. Clearly software from the wider industry would not be designed specifically with our use case and structure in mind, and some adaption, both technically and in terms of process, would be required. This paper reviews the challenges we faced and the solutions that were adopted.

2. Configuration Management as an infrastructure service

2.1. CERN IT Services

Services provided from the CERN Computer Centre can be both horizontal and vertical in scope. Certain shared infrastructure is clearly horizontal, such as datacenter infrastructure (power, cooling, etc), the network or hardware provisioning. There is also what we could term service infrastructure, such as tools to configure and monitor services, which is also provided as a horizontal service.

Services to which the primary customer or consumer is external to the IT department tend to be vertical services. Managers of the vertical services expect to be able to manage their entire service. Unlike what may be the practice in some other large organizations, perhaps that have more regulatory oversight, service managers at CERN expect to be able to manage their entire service. Changes, excluding those that have an impact beyond the scope of the managed service, should be effected by the manger concerned.

2.2. Why centralize?

The Configuration Management service therefore needs to be able to provide distinct groups of service managers the ability to make changes, but is there value in centralizing? Could each group effectively do their own configuration with no need to worry about how another group configuring their service, or in fact what tools they use to accomplish this. There are though a number of reasons why a centralized configuration management service is desirable.

2.2.1. Efficiency. It's not sensible for the managers of each service to have to decide which way is best to configure basic system services such as NTP.

2.2.2. Sharing. Services typically rely on a number of other infrastructure elements. An easy way to share knowledge between groups is to have different groups provide configuration – for instance monitoring configuration, or storage configuration for services like AFS, CASTOR or CVMFS.

2.2.3. Overall view of system. With a shared system, products such as PuppetDB [6] can give a view into the entire infrastructure, which can be leveraged in a number of different ways.

2.2.4. Security. All changes to the system have a single audit point, and a well defined shared security model.

2.3. Alternative models.

Other models are of course possible, and may in fact be more suited to the use of something like puppet "out of the box". There are clearly more small organizations, businesses and other use cases than something of the size of CERN. Many issues of adoption would be resolved if a small team was responsible for all configuration, as may be the case in these organizations. In larger organizations regulatory constraints or similar may lead to small teams effecting change on behalf of internal customers. There are solutions such as the integration of Service Now [7] with puppet in order to have mechanisms to automate these kinds of organizational setups.

3. Puppet definitions

It's important to define some of the terms we'll be using in this paper, and which in and of themselves steps we took at a very early stage to adapt puppet to our environment. However in this instance at least these decisions were more or less defined by the technology so should hopefully be familiar to readers familiar with puppet and foreman [8]. Though puppet code has no such distinctions per se, we make a distinction between reusable code and service definitions. We do so by defining, and using within the puppet namespace "modules" and "hostgroups".

3.1. Modules

Modules are reusable packages of code, that should be created to configure a particular "thing". That "thing" could be a basic OS daemon or service (i.e. sudo), a standard daemon (i.e. apache), CERN specific bundle of configuration (i.e. hardware tools), or community or service specific bundles (i.e. Grid software). Modules should be in the form of what kind be found on the puppet forge [9]. They can indeed be installed from and via the forge, or created locally, but they should be reusable.

3.2. Hostgroups

A hostgroup is a group of machines that run a particular, or similar service. In terms of puppet code, it should be both the code and the data that describes a service. This is achieved by invoking the appropriate modules in the way specific to that service definition.

4. Development model

4.1. Problems with single git repository

Ultimately, puppet code has to be provided to the puppet master as a unitary tree. In the early days of our puppet migration, we treated the tree as a single project, with all the code hosted in a single git [10] repository. The initial workflow we settled on was that changes would be tested in git feature branches mapped to puppet environments [11]. Once tested to the satisfaction of the author, they would be cherry-picked into a development branch, and then after a week cherry-picked to a production branch/environment. Unfortunately, cherry-picking changes is not always simple. Not all changes could be cherry-picked cleanly, leading to an onerous and often long task that could only be performed by a central configuration team. At times some cherry-pick operations had so many dependencies and conflicts that additional risk was introduced into the change. We agreed with the statement that "Asking experts to do boring and repetitive, and yet technically demanding tasks is the most certain way of ensuring human error that we can think of, short of sleep deprivation, or inebriation" [12].

4.2. Distributed development

Given the problems with a unitary tree, the development process was distributed with each hostgroup or module being provided as git repository. In order to provide a way of providing a QA workflow, each git repository would provide both a "master" and "qa" branch at minimum, which would be used to create "production" and "qa" environments respectively. In the case of hostgroups, each team managing the service would have rights to push to both master and qa branches. In the case of modules, the module maintainers would have the rights to push, but in the case of "shared" modules there would be formal QA process for any changes. We will discuss the identification of shared modules and the QA process later in the paper.

4.3. Scale of development

Some numbers gleaned from git statistics may help show the size of the problem, and why a single git repository was impractical – even before we consider the positive effects on the workflow with multiple repositories. The numbers are assembled from hostgroup statistics rather than modules. Since many modules are upstream rather than created in-house, the statistics often reflect external effort and complexity.

Table 1. Hostgroup authors by year

2013	**2014**	**2015 (to May)**
83	242	191

Table 2. Git repository statistics

Hostgroups	138
Modules	283
Hostgroup lines of code	1015449
Hostgroup avg. commits / month	2989.33

4.4. Shared Modules
A shared module is defined as a module that is used by more than one hostgroup. This is determined by using PuppetDB to see which nodes have included or declared classes within a puppet module. Using the resources endpoint of the PuppetDB API [13] makes this a fairly simple operation. If those nodes belong to more than one hostgroup, then the module is deemed to be shared, and subject to the QA process.

4.5. Module QA process
The QA process has been adopted as mandatory for shared modules, and is also recommended for other changes. The process is in summary:

- Change tested by the author in a feature branch and test environment
- Merged into the QA environment, at which point a ticket is created to inform service managers about the change.
- The change must stay in QA for a minimum of one week, during which time any service manager can stop the change if it causes problems.
- Finally, assuming the change has caused no issues it can be merged into the production environment.

4.6. Continuous Integration and testing
The QA process lends itself well to using Continuous Integration tools and techniques to reduce the risk involved in making changes. As all modules are distinct and separate, they can be tested separately, and test coverage across the entire system can be built up incrementally. The aim has been to make testing changes "frictionless" by which to say for the module maintainer making the change it should be if possible easier to test than to not do so. We do this by using Jenkins [14] alongside a Jira [15] workflow for change management tickets.

4.6.1. Automatic test workflow. When creating a Jira ticket for a module change, the Service Manager can elect to use the automated test framework. A script monitors ticket creation using the Jira API, and where selected will take over the repository management for that change. Before merging change to the QA environment, Jenkins jobs will run unit tests for the modules, as well as creating VMs to run functional and system tests against the change. This can be a basic test, such as testing a "base build" of a puppet managed machine, or it can be more sophisticated, testing functionality that the module may impact. For example, by creating a new virtual machine running a webservice running apache using Single-Sign-On (SSO) authentication. The feature branch will be merged into QA only if these changes are successful. After the defined QA period, Jenkins will again run tests, including an additional test to check if any Service Manager has defined the change as "broken in QA" which must all pass before merging to production. All changes in state of the workflow, or indeed failures, are communicated to the module maintainer and the wider community by updating the Jira ticket.

4.6.2. Continuous Tests such as the creation of SSO websites, the creation of base VMs, or of VMs of particular OS versions are performed constantly. These tests are displayed in a dashing [16] radiator, and provide immediate feedback on issues.

4.7. *Jens – a puppet librarian*
In order to support the multiple repository development model that we chose, we needed something to compile a module tree from all the repository sources. When we began we'd hoped that projects such as Puppet Librarian [17] would be useful for our use case, but unfortunately they did not fulfill our requirements. Other projects that may be useful to consider such as r10k [18] were not available at the time. We were looking for something that could generate a puppet module tree for a number of targets

– either different filesystems or different puppet masters. We wanted to generate a tree from a number of different arbitrary git sources. We wanted to be able to generate puppet environments on demand. Additionally, whereas hostgroup is a concept that is familiar to Foreman, and is analogous to the "clusters" that were previously defined in Quattor [19], it is by no means used in all puppet installations.

4.7.1. Jens design. Jens is a python toolkit to generate puppet environments dynamically based on some input metadata. The metadata is a YAML file containing a list of modules, a list of hostgroups and a list of "common" code such as puppet site information and common hiera [20] data. A "production" environment is generated by using the master branch of each repository defined in the metadata. Likewise a qa environment is generated from each qa branch of each repository. It's also possible for service managers to define environments themselves with environment definition YAML files. These may state that, for instance, the "qa" branch is taken as default for each repository but that for a particular subset of hostgroups and modules that a feature branch is instead chosen. This enables testing of only the change, and to keep getting changes for the other modules. It means that as little as possible is forked to test a particular change. This gives the maximum flexibility possible to manage change across the range of module managers and service managers.

4.7.2. Snapshots. With a strong QA process, it's usual that most services accept changes into production using this channel. However it's not always the case that all services wish to take changes at the same rate. As has been remarked by too many people to reference, continuous delivery does not need to mean continuous deployment. What's important is to ensure that everything is always tested and deployable, not that it's instantly deployed. It's our opinion that risk is introduced the longer changes are delayed into to the change process, where the subject matter expert responsible for the change is further away from the point of deployment. Regardless though a delayed deployment model is necessary. Therefore support for snapshots was introduced to Jens. Whereas normally environment definitions are based on git branches for individual repositories, it's also possible to refer to git references. There is some fragility in this system as git references can be removed by developers making non fast-forward changes, but this is normally not the case in qa & master. This means that service managers can take a snapshot of the whole tree, or the parts of the tree not under their control, and effectively freeze change.

5. Security

5.1. A question of trust.
As can be seen in Table 1, CERN has a large number of people working with the configuration management system at any one time. It's also the case that CERN's staffing model leads to a reasonably high throughput of staff. Clearly there are a lot of privileged users, and they change often, and this introduces risk. The risk doesn't have to express itself in terms of malicious staff, it's much more likely that an account can be compromised. We should therefore trust an individual account as little as possible.

5.2. Minimise scope of problems.
Whether it be a compromised account or just a mistake, problems should bleed into other services as little as possible. A problem introduced by the account of a batch service manager shouldn't have adverse effects on the CASTOR service.

5.3. Puppet issues with this model
Puppet has a number of issues with this model. All puppet developers can write puppet code that is executed in the context of the puppet master. This code includes puppet DSL, but also templates and

functions. Both of these latter examples basically give the developer a shell on the puppet master with the ability to run anything in the context of the puppet master.

5.4. Pluginsync

Pluginsync [21] allows for "libdir" code to be downloaded from the puppet master as shipped in puppet modules. This includes things like "facts" which is ruby code that runs with a privileged account on client machines. By default all libdir in the module path is downloaded to all puppet clients. This is obviously problematic and so we provide functionality to "whitelist" modules to prevent potentially malicious or just unwise code running on service managers' machines.

5.5. *Secrets*

The mechanisms to distribute secrets to machines all rely on the puppet master being able to decode the secret. This means that any secrets are effectively secret only to the people that have access to the puppet master. Our solution was to develop a system to use credentials on the client machine (either Kerberos keytab or x509 certificate) rather than the context of the puppet master. It's notable that there are now products emerging in this space such as Vault [22] or KeyWhiz [23]

6. Conclusions

There are lots of organizations at the same size or larger than CERN and with larger compute problems. Tools such as Puppet do scale to the size of the compute environment that we have to deal with. This is not to say that we can expect tools in this space to work out of the box to the size of teams that we are typically working with. The sweet spot for such tools may well be SMEs or similar sized departments. Work is going to be required with these tools to adapt them to large organizations.

References

[1] Tim Bell et al, Review of CERN Data Centre Infrastructure, 2012 J. Phys.: Conf Ser. 396 042002
[2] LCG – Worldwide LHC Computing Grid, http://ldg.web.cern.ch/LCG/
[3] https://www.openstack.org
[4] https://puppetlabs.com/puppet/what-is-puppet
[5] http://en.wikipedia.org/wiki/Conway's_law
[6] https://docs.puppetlabs.com/puppetdb/latest/
[7] http://wiki.servicenow.com/index.php?title=Puppet_Configuration_Automation#gsc.tab=0
[8] http://theforeman.org
[9] https://forge.puppetlabs.com
[10] http://git-scm.com
[11] https://puppetlabs.com/blog/git-workflow-and-puppet-environments
[12] Jez Humble & David Farley 2010 *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*
[13] https://docs.puppetlabs.com/puppetdb/2.3/api/query/v3/resources.html
[14] https://jenkins-ci.org
[15] https://www.atlassian.com/software/jira
[16] http://dashing.io
[17] https://github.com/rodjek/librarian-puppet
[18] https://github.com/puppetlabs/r10k
[19] http://www.quattor.org
[20] https://projects.puppetlabs.com/projects/hiera
[21] https://docs.puppetlabs.com/guides/plugins_in_modules.html
[22] https://www.vaultproject.io
[23] https://square.github.io/keywhiz/