How to Organise Your Infrastructure as Code

Abhishek Tiwari

Citation: *A. Tiwari*, "How to Organise Your Infrastructure as Code", Abhishek Tiwari, 2015. doi:10.59350/6ey3f-dxa55

Published on: December 22, 2015

DevOps is a cultural shift with immediate focus on maximising the business value by opting better communication, collaboration and feedback within and across IT development and operation teams. Infrastructure as Code (IaC) is a key element of DevOps philosophy with benefits for both development and operation teams. The term infrastructure as code is sometimes also referred to as programmable infrastructure. In an infrastructure as code implementation whole infrastructure lifecycle including orchestration, provisioning, configuration, monitoring, self-healing can be managed in an automated fashion.

Traditionally infrastructure lifecycle management has been a manual process which has often resulted in environmental inconsistencies. For instance, staging and UAT environments not fully compatible with production environment is a very common issue in traditional technology functions. This type of environmental inconsistencies often creates a lot of friction for application and solution delivery hence slowing down or sometimes blocking the pace of innovation in a digital organisation. In general, infrastructure as code offers the advantage for both cloud as well as on-premise infrastructure but it particularly brings major benefits when applied in a cloud environment. When used in a cloud environment infrastructure as code has the potential to supercharge application delivery without compromising the quality.

In this article, I would like to focus on how to organise your infrastructure as code - something I haven't seen covered anywhere else in detail. In recent years, I have implemented infrastructure as code at the scale at least 3 times and this article heavily borrows from my learnings and experience on these projects.

Infrastructure as Code Best Practices

First thing first, best practices are quite important for managing infrastructure as code in a highly efficient way, so please consider following key recommendations for your infrastructure as code project.

Codify everything

Whenever possible use code to describe the infrastructure. As you can see below most parts of traditional and cloud infrastructure can be described as code.

- For infrastructure (physical or virtual servers) management Terraform, CloudFormation, YAML and Python scripts
- For DNS management various scripts typically interacting with DNS system via APIs
- For configuration management Puppet/Chef modules, Powershell/Bash scripts
- For network management Puppet/Chef modules

- For container management Dockerfile
- For remote command execution and deployment Fabric/Capistrano
- For local testing and development using Vagrantfile

Version everything

You should be versioning everything including build and binary artifacts. For versioning and change tracking, you can use any source code management system (SCM), but I **highly recommend Git** to manage infrastructure as code repository. Git is an open-source distributed version control system. Due to distributed nature and various collaborative branching models Git is highly scalable for a project of any size. Although, Git is highly popular in DevOps community I am particularly interested in Git Submodule functionality.

Moreover, I will suggest using an appropriate branching model according to your business needs. At the minimum, you can use two branches `master` and `develop`. You can assume`master` branch as fully tested shippable or deployable infrastructure code, everything else which is not fully tested or under development remains in `develop` branch. Once changes in `develop` are ready you can merge them in `master`. Before release or deploy from `master`, please use Git tagging functionality to mark release points (v1.0, v 2.0 and so on).

One repository

I highly recommend strictly one infrastructure as code repository per organisation or company. In my experience, one infrastructure as code repository is more than sufficient for an organisation of any scale. In my opinion, one infrastructure as code repository should be an integral part of the DevOps philosophy. This repository acts as an entry point or main function for your infrastructure as code implementation. Having said that, you can always modularise your infrastructure as code implementation and take one repository concept to next level as described below.

Modular components

Often one need to break infrastructure down into modular components and tie them together in an automatable way. You can always modularise the external and internal dependencies. For example, if you are using a Puppet module or Chef cookbook for cookbooks then it makes perfect sense to modularize them as the dependency. Modular components can be more manageable and offer several benefits. Dependency management and access control are two key benefits of modular components.

In order to do modular components at scale, I have found it's quite essential to use Git submodules. Git submodule functionality allows you to keep another Git repository in a subdirectory of your infrastructure as code repository. The other Git repository maintains its own history and it has no impact on the history of the infrastructure as code repository. Git submodule can be used to have external as well as internal dependencies. This can be used as a mechanism to manage access control as you will be maintaining separate Git repositories each with it's own read and write permissions. When cloning or pulling a repository infrastructure as code repository, submodules are not checked out automatically. You will need to issues explicit command to update or checkout Git submodule.

Test coverage

Testing infrastructure as code is quite necessary. To a large extent, having an optimal test coverage will ensure there are no post-deployment bugs. Typical change management process can not guarantee that there will be no post-deployment issues. In fact, this is one area where operations can learn and adopt a lot of techniques from development. Currently, there are two popular testing approaches in the infrastructure space: Behavior-driven (inspired from BDD) and Test-driven (inspired from TDD). Depending on your need a range of test suits can be developed: unit tests, regression tests, acceptance tests, end-to-end tests, and property tests. These test suites differ in terms of scope and focus.

Continous Integration

Continuous integration of your infrastructure is highly desirable. Continuous integration can enable you to run automated tests suits every time a new change is committed into your infrastructure as code repository. In addition, continuous integration can create deployable artifacts such backed virtual machines, AMIs and docker containers. Last but not least, it also pipelines the change management and deployment process.

Organising Infrastructure as Code

Now that we have covered key best-practices for managing infrastructure as code, we can now discuss how to organise your infrastructure as code. To make it easy, I have created a seed project for infrastructure as code on Github which can be used as starting point or skeleton for your infrastructure as code implementation. You can clone this repository and use it to quickly bootstrap your own infrastructure as code project.

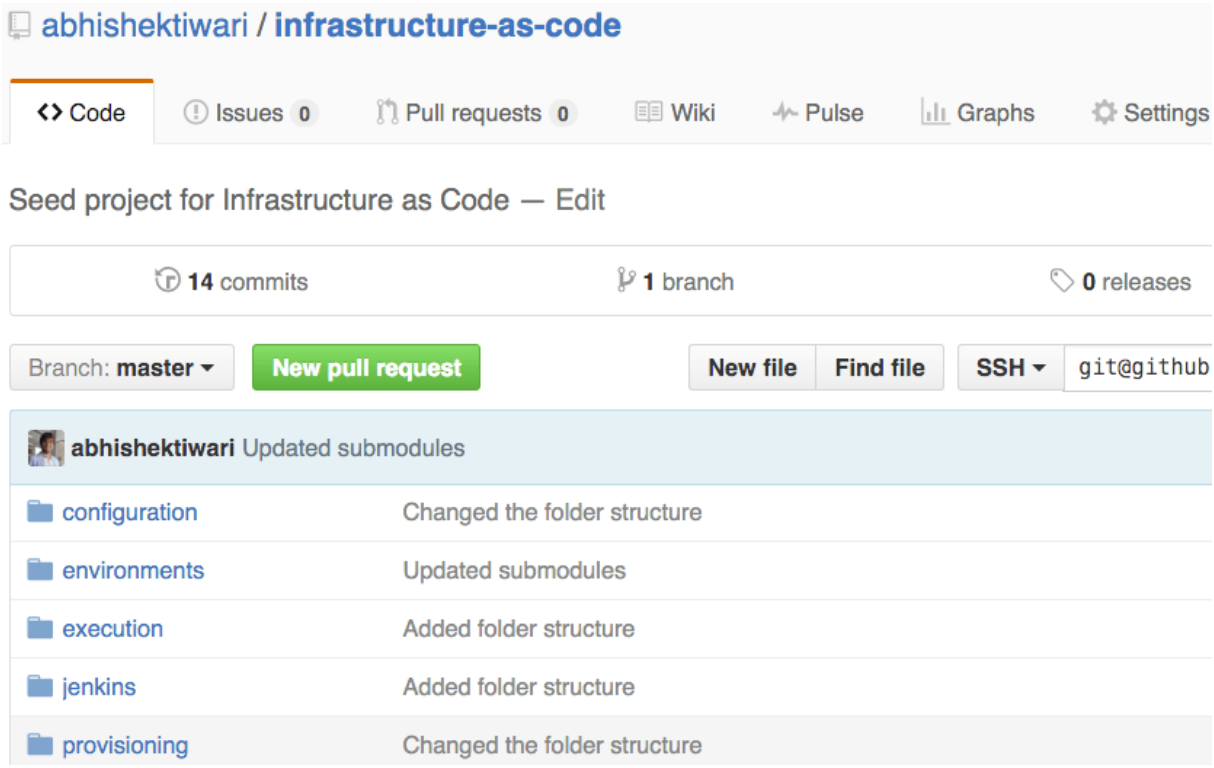

Figure 1: Seed project for Infrastructure as Code

To clone seed repository please run following commands from your terminal,

```
git clone https://github.com/abhishektiwari/infrastructure-as-code.git
```

Configuration

All configuration management code is placed under `configuration` folder which has subfolders for providers (`puppet` and `chef`). Below these folders we have folders containing third-party packages as Git submodules (Pupppet `modules` or Chef `cookbooks`). These modules are responsible for configuring individual servers but they can also be used for application deployment.

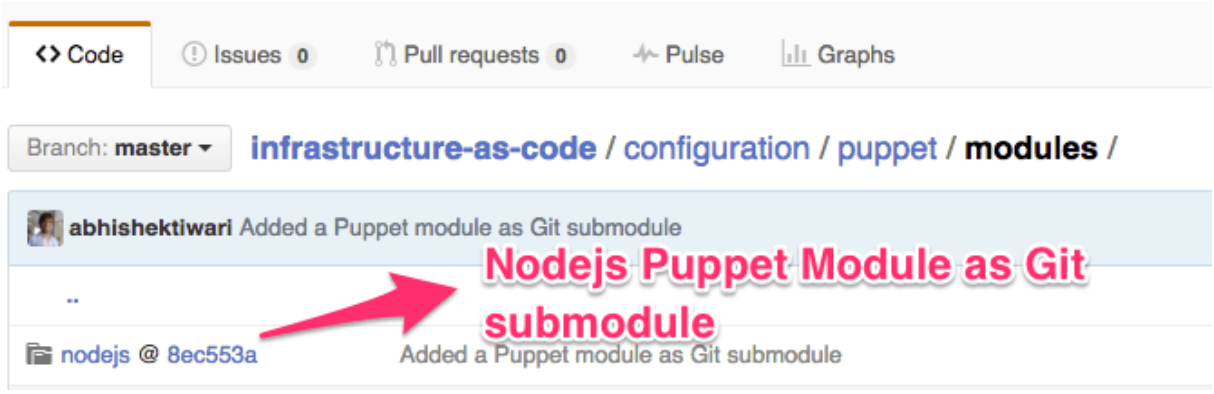

Figure 2: Infrastructure as code repository: Puppet module as Git submodule

Provisioning

All infrastructure orchestration and provisioning code is placed under `provisioning` folder which subfolders for providers. At this stage, it accommodates Terraform, Cloudformation, boto and lib-cloud scripts. CloudFormation and boto are more focused on Amazon Web Services cloud, but Terraform along with libCloud are platform agonist and support majority of providers including Open Stack and VMware.

Execution

For remote server automation and command execution Fabric and Capistrano scripts are used. When using Fabric and Capistrano on a large number of servers, execution can be in parallel or sequentially. It is possible to perform the remote execution using the Puppet/Chef but their approach is mostly pull based (nodes pulling configuration from master) and Fabric/Capistrano execution model is push based. Both Fabric and Capistrano execute predefined tasks. Tasks are nothing but Python/Ruby functions with the wrapper around the Bash/Powershell commands.

```
fab task1 task2 task3
cap task1 task2 task3
```

Normally, the first task sets the remote host where the commands or tasks will be executed followed by a list of tasks to execute on remote hosts.

```
fab production update_os
cap production update_os
```

Let's look into `update_os` task defintion for Fabric,

```
@parallel
def update_os():
    sudo('apt-get update -y')
    sudo('apt-get upgrade -y')
```

As you can see from above examples that the tasks are executable units and they can be easily described as code.

Environments

Now that we have building blocks to create, configure and update the infrastructure environment, we can now define and run applications on these environments. In this particular case, `environments` folder is more like testbed to perform development and testing using Vagrant and Docker. Under `environments` you can create Git submodule for application environments such as `website`, `cms`, etc. and include appropriate `Vagrantfile` and `Dockerfile`. Using submodules means your application environment dependencies can resolve. For instance, your `Vagrantfile` refers to Puppet manifest and modules both sits inside the infrastructure as code repository. Ideally, your application developers should be able to use these Vagrantfile and Dockerfile included in these modules as application development environment.

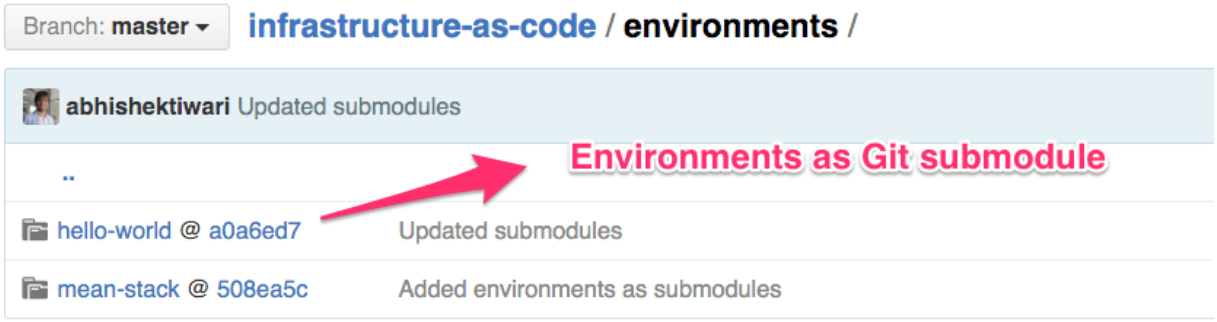

Figure 3: Environments as Git submodules

Closing comments

This seed project for infrastructure as code still a work in progress. So feel free to send a pull request. Currently, repository structure is highly opinionated and convention driven. I can think of few additional ways to organise infrastructure as code, but despite any possible shortcomings the repository structure described in this article can scale very well. Lastly, if you think one repository per organisation is not scalable then think once more.