



How Jenkins Builds the Netflix Global Streaming Service

 @bmoyles @garethbowles #netflixcloud



**Abstract:** Hi, I'm Gareth and this is Brian, we're from Netflix. Welcome to part 2 of the Netflix invasion of the Jenkins User Conference !

Over the last couple of years, Netflix' streaming movie and TV service has moved almost completely to the cloud, using Amazon Web Services. You guys may well have seen some of the great presentations and blog posts by our engineers talking about how our production service works - but what does it take to get that service into production ? This talk will delve into our build and deployment architecture, concentrating on how we use Jenkins as the centrepiece of our production line for the cloud.

# The Engineering Tools Team

- Make build, test and deploy simple and pain-free
- Various skills - developers, build/release engineers, devops guys
- CI infrastructure
- Common Build Framework
- Bakery
- Asgard



Brian and I work on the Engineering Tools team at Netflix. There are 6 of us here today (about half our team, including Justin who gave the talk on getting to your 3rd plugin this morning), so find any of us later if you want to know more about Netflix (especially if you'd like to work with us, we're hiring !)

Our team is responsible for making standard tools for all our engineers to make building, testing and deploying their products simple and pain-free.

We're now a fairly big team with a diverse range of skills, everyone can go pretty far up and down the stack (although don't ask me to do any UI work if you don't want to make your eyes bleed !)

Our team runs the Netflix CI infrastructure, including a Common Build Framework that we wrote to make it trivial for developers to create builds for their products.

Among other tools that we wrote are the Bakery, which creates our custom Amazon Machine Images for deploying to the cloud, and Asgard, the open source console that you can use to manage your AWS deployments. More on these coming up.

# Jenkins History

- Jenkins (and H\*\*\*\*\* before it) in use at Netflix since 2009
- Rapid acceptance and increase in usage, thanks to:
  - Jenkins' flexibility and ease of use
  - Our Common Build Framework
- Quantity, size and variety of jobs means we need automation and extensions



We've been using Jenkins (and its predecessor who Shall Not Be Named) for a few years now and it's become a vital part of our CI infrastructure.

Use of Jenkins caught on really fast, partly because it's easy for people who aren't professional build engineers to use it, and also due to our Common Build Framework that makes it a trivial operation to set up a new build job (more on that later). In fact, Netflix's product teams don't have full time release engineers; our Engineering Tools team aims to make the build / test / deploy process trivial enough not to get in the way of product development. Jenkins has been a big help with that.

We've become one of the bigger Jenkins users (although not the biggest, for those of you who saw the "Planned Parenthood" talk by the Intel guys earlier, they are an order of magnitude bigger than us) quite fast, and that's given our team quite a few challenges in meeting the demand. We've come up with custom plugins, scripts and use patterns for Jenkins that we think are pretty cool and certainly make our lives easier; we'll share many of them in this talk.

# What We Build

- Large number of loosely-coupled Java Web Services
- Common code in libraries that can be shared across apps
- Each service is “baked” - installed onto a base Amazon Machine Image and then created as a new AMI ...
- ... and then deployed into a Service Cluster (a set of Auto Scaling Groups running a particular service)
- So our unit of deployment is the AMI



Let's start with some background on what we build using Jenkins.

To get to the cloud, we rearchitected the Netflix streaming service from a monolithic Java webapp into many individual modules implemented as loosely-coupled web services, usually web applications or shared libraries (JARs). Most services implement a server and a client library that other services can use to talk to it.

We decided that our unit of deployment would be the Amazon Machine Image - a complete description of a server including the operating system, not just an install package. This frees us from having to maintain a configuration server such as Puppet or Chef, and makes it very easy to roll back and forward between versions of a service - you just kill instances of a version you don't want and spin up instances with a different version. Netflix is all about continually trying out new features and rolling forward (preferably) or back fast if needed.

We maintain a base AMI that encapsulates our selected o/s - currently CentOS, soon to be Ubuntu - plus a host of other basic services that provide monitoring, logging, web and application servers (Apache HTTPD and Tomcat) and much more. We have an application called the Bakery (soon to be open sourced) which handles installation of Netflix services onto a base AMI to create a custom AMI for each service.

Once the AMIs are ready, we deploy them into AWS Auto Scaling Groups using Asgard, our open source management console. Asgard is a fantastic piece of work that goes way beyond what the standard AWS Management Console can do, and I encourage any of you who use AWS to check it out.

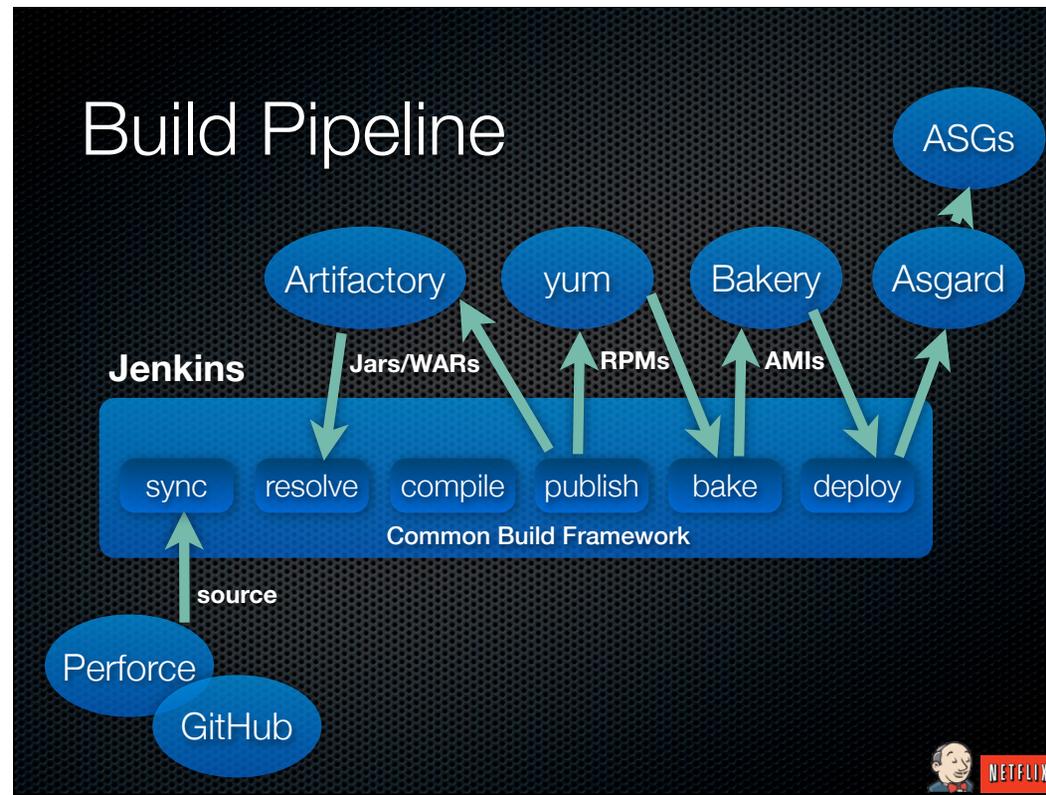
Note that a key aspect of using so many shared services is that each service team has to rebuild often in order to pick up changes from the other services that they depend on. This is the CONTINUOUS part of continuous integration and is where Jenkins comes in.

# Getting Built



Here are a few details on how we build all those cloud services.

# Build Pipeline



Here are the tools that form our continuous build and deployment pipeline. Jenkins jobs can provide the entire workflow.

We wrote a Common Build Framework, based on Ant with some custom Groovy scripting, that's used by all our development teams to build different kinds of libraries and apps. (If you think Ant is a bit long in the tooth, so do we; we're working on a brand new build framework using Gradle, but that will take a while to finish up and get everyone converted over).

For the continuous integration to run all those builds, we picked Jenkins because it's very feature rich, easy to extend, and has a very active and helpful community (thanks, guys!).

We use Perforce for our version control system as it's arguably the best centralized VCS available. But we're making increasing use of Git; for example, our many open sourced projects are all hosted on GitHub, and we use Jenkins to build them.

We publish library JARs and application WAR files to the Artifactory binary repository tool. This lets us access and add to the metadata for each published artifact, and allows us to add Apache Ivy to Ant to handle dependency resolution. So each project only has to know about its immediate dependencies.

JARs and WARs are then bundled into RPM packages for installation.

I mentioned the bake and deploy process for the finished artifacts just now. The Common Build Framework includes Ant tasks that access the Bakery and Asgard APIs to run these operations. Being able to do that workflow from a Jenkins job can save a lot of time; for example, our API team was able to move from one deployment every two weeks to multiple deployments per week.

It's worth mentioning that the full automated deployment cycle is mainly used in our test environment; deploying to production needs extra checks and balances that tend to be specific to the service being deployed, many of those aren't yet fully automated.

# Jenkins Setup for a New Build Job

Project Details

Workspace (client)

Let Jenkins Create Workspace

Let Jenkins Manage Workspace View

Client View Type

Stream

View Map from File

View Map

```
//depot/commonlibraries/nf-deploy-utils/main/...  
//depot/Tools/build/...
```

Build

Invoke Ant

Ant Version

Targets

Build File



Here is all you need to do in Jenkins to set up a typical project's build job. You just tell Jenkins where to find the source code and add in the Common Build Framework, then specify where your Ant build file is and what targets to call from it.

Even this minimal setup can be tedious when repeated for a large number of jobs, so we're working on a plugin to make it even more trivial; more on that later.

## Ant Setup for a New Build Job

### build.xml

```
<project name="helloworld">  
  <import file="../../Tools/build/webapplication.xml"/>  
</project>
```

### ivy.xml

```
<info organisation="netflix" module="helloworld">  
  <publications>  
    <artifact name="helloworld" type="package"  
      e:classifier="package" ext="tgz"/>  
    <artifact name="helloworld" type="javadoc"  
      e:classifier="javadoc" ext="jar"/>  
  </publications>  
  <dependencies>  
    <dependency org="netflix" name="resourceregistry"  
      rev="latest.${input.status}" conf="compile"/>  
    <dependency org="netflix" name="platform"  
      rev="latest.${input.status}" conf="compile" />  
    ...  
  </dependencies>  
</info>
```



And here is a simple project's Ant build file and Ivy publish & dependency definition file.

You can see the Ant code simply pulls in one of the standard framework entry points like, library, webapplication, etc.

The Ivy publications section just defines what type of artifacts you want to publish; the Common Build Framework has the knowledge of how to create those artifacts.

# Jenkins at Netflix



NETFLIX

Let's take a closer look at how we use Jenkins as the core of our build infrastructure, plus a few other interesting uses we've come up with.

# Not Just a Build Server

- Monitoring of our Cassandra clusters
- Automated integration tests, including bake and deploy
- Housekeeping and monitoring of
  - the build / deploy infrastructure
  - our AWS environment



At its heart Jenkins is just a really nice job scheduler, so we've found lots of other uses for it. Here are some of the main ones.

Our Cassandra team has their own Jenkins instance for monitoring the health of our Cassandra clusters and performing automated repairs on them. This one is a great testimonial for Jenkins' ease of use; Engineering Tools just gave the Cassandra guys a basic Jenkins master and some slaves, then they set up their own jobs, a custom dashboard, and even turned the blue balls for build success to green ones.

Housekeeping jobs usually use system Groovy scripts for access to the Jenkins runtime. We effectively use Jenkins to manage Jenkins and its associated apps, in a similar way to the Infrastructure jobs on <https://ci.jenkins-ci.org>.

We've posted a few of these scripts to the public Scriptler repository and will be adding more. Please check them out and feel free to make changes.

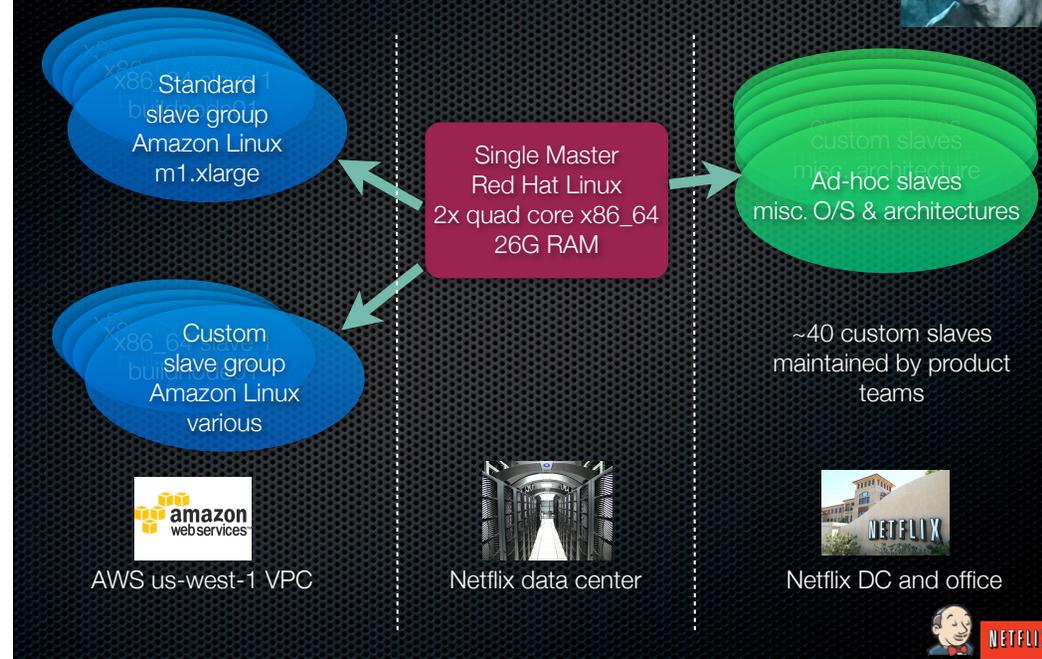
# Jenkins Statistics

- 1600 job definitions, 50% SCM triggered
- 2000 builds per day
- Common Build Framework changes trigger 800 builds
- 2TB of build data
- Over 100 plugins



The CBF build floods run in 2 stages: the first main one is triggered by the SCM change, then downstream test and deploy jobs are triggered by Jenkins.

# Current Jenkins Architecture (One Master to Build Them All)



Our Jenkins master runs on a physical server in our data center.

We use the master strictly as a build orchestrator and a runner for housekeeping scripts; we don't run regular builds on the master to avoid overloading it.

Slave servers are used to execute the actual builds. Our standard slaves can each run 4 simultaneous builds. Custom slave groups are set up for requirements such as C/C++ builds or jobs with high CPU or memory needs.

We make extensive use of slave labelling to control the type of slave that each job executes on, and we have housekeeping jobs that make sure that build jobs are using the correct labels.

We vary the number of standard slaves from 10 to 30 depending on demand. This is currently a manual operation (but all you have to do is run a Jenkins job !); we're working on autoscaling.

Our cloud slaves are set up in an AWS Virtual Private Cloud (VPC), which provides common network access between our data centre and AWS. Amazon's us-west-1 region is physically located close to our data centre, so latency is not an issue.

Ad-hoc slaves in our DC or office are used by individual teams if they need an O/S variant other than those on our standard slaves, or a specific tool or licensed app. Typical uses are for our Partner Product Development team, which makes the Netflix clients that run on TVs, DVD players, smartphones, tablets, and soon - I'm reliably informed - toasters.

We keep our standard slaves updated by maintaining a common set of tools (JDKs, Ant, Groovy, etc.) on the master and syncing the tools to the slaves when they are restarted. Custom slaves can also use this mechanism if they choose.

The single master has definitely reduce maintenance overhead and, thanks to continuing performance improvements from the Jenkins community, it's handled our expanding workload pretty well up to now. Still, it's a single point of failure, and we're starting to run up against a number of scaling issues with this architecture.

That's the present - now I'll hand it over to Brian to take us into the future.

# Scaling Challenges - and How We Met Them



Thanks Gareth. I'm going to talk about some of the scaling challenges we've faced in building and growing our Jenkins infrastructure, and what we've done to address them.

# Jenkins Scaling Challenges

- Thundering herds clogging Jenkins, downstream systems
- Making global changes across jobs
- Making sure plugins can handle 1600 jobs
- Testing new Jenkins and plugin versions
- Backups

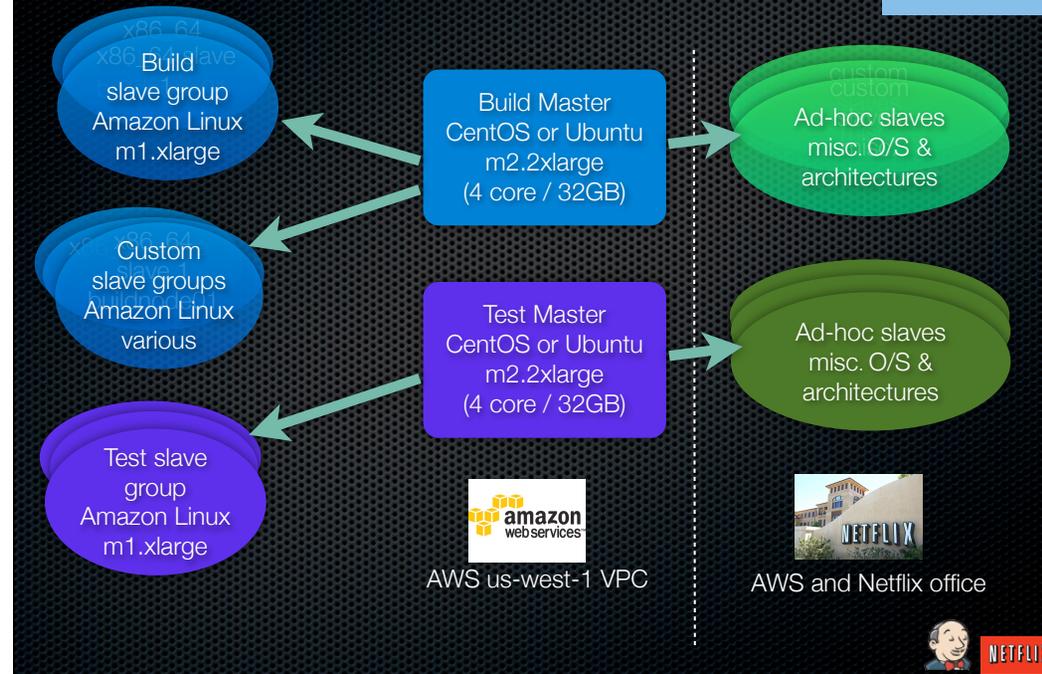


We've run into a number of scaling challenges as we've evolved our build pipeline: Thundering herd problems, modifying and managing 1600 jobs, making sure those 1600 jobs work from Jenkins version to Jenkins version, plugin version to plugin version, and so on.

A good example problem: when we update our build framework, this often triggers all SCM triggered jobs that use the CBF. We can scale our slave pool up, but the slaves push to artifactory, resolve deps from artifactory, and so on, so more slaves means more artifactory load. It's a balancing act.

Our goal, of course, is to have one button build/test/deploy with as little human intervention as possible, and make the developer's life as pain-free as we can. All of these scaling issues get in the way of that. Here are some of the ways we're working on eliminating them.

# New Architecture (Cloudy with a Chance of Builds)

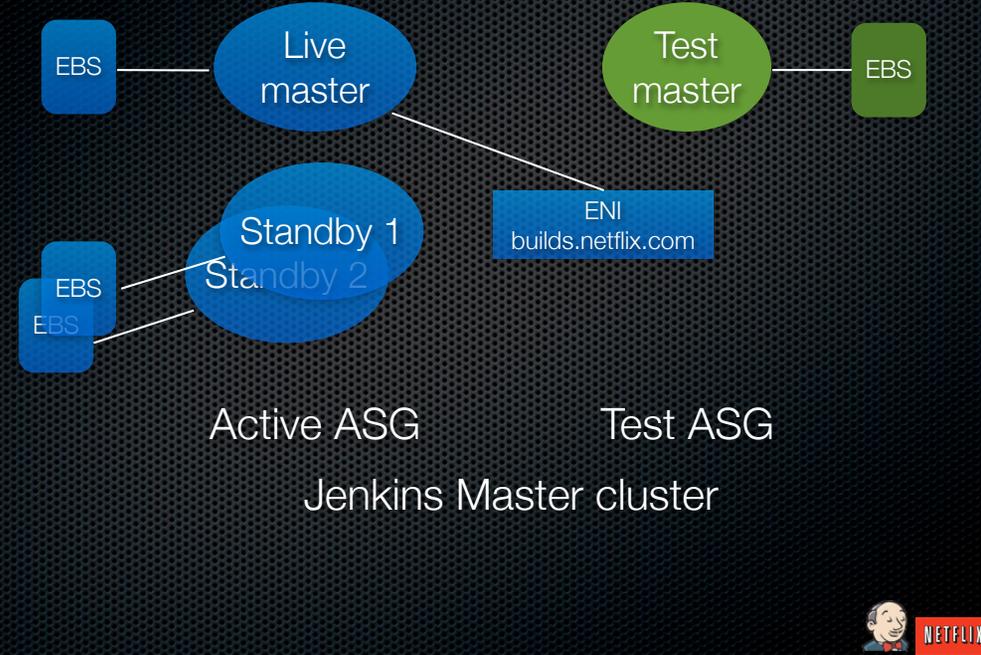


To start, we're working on a sharded model for our builds. Our current thought is we'll split the current master into multiple AWS instances dedicated to different types of work: standard CI builds, integration tests, non-Java builds for our device clients, etc.

The set of plugins for each master can be streamlined to fit the type of job. Isolate long-running jobs so we don't bounce Jenkins for maintenance in the middle of someone's 10 hour integration test, schedule regular planned maintenance for the long-running cluster, etc.

We'll use the same bake & deploy architecture that powers our streaming service to run the Jenkins masters and slaves.

# Deploying a New Master



To deploy the Jenkins master, we'll use the same "Red / Black deploy" principle that our streaming service uses in production. We launch and test the master in a manner that prevents it from taking on traffic. When we're satisfied with where things are, we flip a switch, and the test becomes the real master.

To go into more detail:

The live ASG will have a master that's running builds, plus one or more standby instances with builds disabled. We store job data on EBS volumes--the Amazon term for persistent storage that doesn't disappear with cloud nodes. The job data is synced regularly from the live master to the standbys.

The test ASG has a similar job data setup, but uses a representative subset of jobs to give us confidence in the new master and plugin versions.

If we want to switch to a new master version, or fail over to one of the hot standbys, we bring down the live master, attach the live job data volume to the new master, and finally do a network switch to make the new master live. We're planning to use Elastic Network Interfaces (ENIs) to do the network switchover; these are virtual network interfaces which can be moved from one EC2 instance to another.

We haven't fully worked out how we will do job data replication and network switchover yet, so we'd be interested to hear your experiences if you have a similar setup. CloudBees' HA plugin might be an option here; we'll definitely be looking at that.

# Building a New Master

- Jenkins build watches <http://mirrors.jenkins-ci.org/war/latest/jenkins.war> and downloads the new Jenkins WAR file
- Download latest versions of our installed plugins (from a list stored externally)
- Create an RPM containing the Jenkins webapp
- Bake an AMI with the RPM installed under Netflix Tomcat



We've set up a Jenkins build to make us a new master image as soon as a new Jenkins release is available.

We could have used the standard Jenkins RPMs, but this way lets us deploy to the same Netflix Tomcat used by production services and get things like monitoring, logging, Apache proxy etc. for free from our base AMI.

# Job Configuration Management

- [Configuration Slicing plugin](#)
  - Great for the subset of config options that it handles
  - Don't want to extend it for each new option
- [Job DSL plugin](#)
  - Set up new jobs with minimal definition, using templates and a Groovy-based DSL
  - Change the template once to update all the jobs that use it

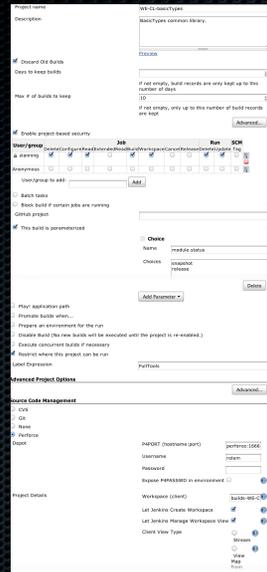


How do we change everyone to a new version of a tool, add a new plugin to everyone's job or make other common configuration changes to a large number of jobs ?

The Configuration Slicing plugin is really nice as long as you want to change one of the subset of options that it handles.

Justin from our team, who presented on "Getting to Your Third Plugin" earlier, has written a job DSL plugin that will allow us to create job templates and simplify the process of configuring new jobs, using a Groovy DSL to define the configuration. Here's a quick peek at how easy that is:

Old and busted...



The screenshot shows a complex web form for configuring a Jenkins job. It includes sections for 'Job' (Name, SCM, Triggers), 'Build' (Steps, Environment), and 'Advanced' (Options, Credentials). The form is filled with various input fields, checkboxes, and dropdown menus, representing a verbose configuration process.



```
job {
  name 'DSL-Tutorial-1-Test'
  scm {
    git('git://github.com/jgritman/aws-sdk-test.git')
  }
  triggers {
    scm('*/*15 * * * *')
  }
  steps {
    maven('-e clean test')
  }
}
```



The new hotness...



etc.

On the left, a typical job configuration page. On the right, a peek at the groovy DSL you'd use to set up a job. This can be made even smaller using templates--pull out the job name and repo location, and you can have a very tiny, simple, reusable representation of a much more complex object, allowing you to do things in a very repeatable, consistent fashion.

# More Maintenance

- System Groovy scripts to clean up old job configs and associated artifacts:
  - Disable jobs that haven't built successfully for 60 days
  - Set # of builds to keep if unset
  - Mark builds permanent if installed on active instances
  - Delete unreferenced JARs from Artifactory



We also employ a number of system groovy scripts for housekeeping both inside Jenkins itself and outside. Jobs will disable abandoned jobs that have been failing for 60 days, ensure everyone's retaining some build history, flagging builds as permanent if they get used in production, and so on...

# Keeping Slaves Busy

- System Groovy scripts to monitor slave status and report on problems
- The Dynaslave plugin: slave self-registration



To help us keep builds moving through the system, we need to keep our slave fleet in shape.

We have system groovy scripts keeping watch over the slaves that complain when something comes up that requires attention. For example, if a specialized slave goes offline for an extended period of time, the team that owns the slave will be notified so they can take action.

We also wrote the dynaslave plugin to help manage slave registration, isolation, and cleanup after disconnect. A quick note for those who cannot make it to the lightning talk--the dynaslave plugin, very simply, allows for slave self-registration without having to tell Jenkins anything about their actual infrastructure. We'll be open sourcing this plugin, so check out the lightning talk or look for our slides on the subject after the conference.

We currently manage some aspects of the dynaslave system using groovy scripts as well. For those that saw our colleague Justin Ryan's presentation earlier, always reach for Groovy first when you feel the temptation to play with plugin internals :)

## Further Reading

<http://techblog.netflix.com>

<http://www.slideshare.net/netflix>

<https://github.com/netflix>

<http://jobs.netflix.com>

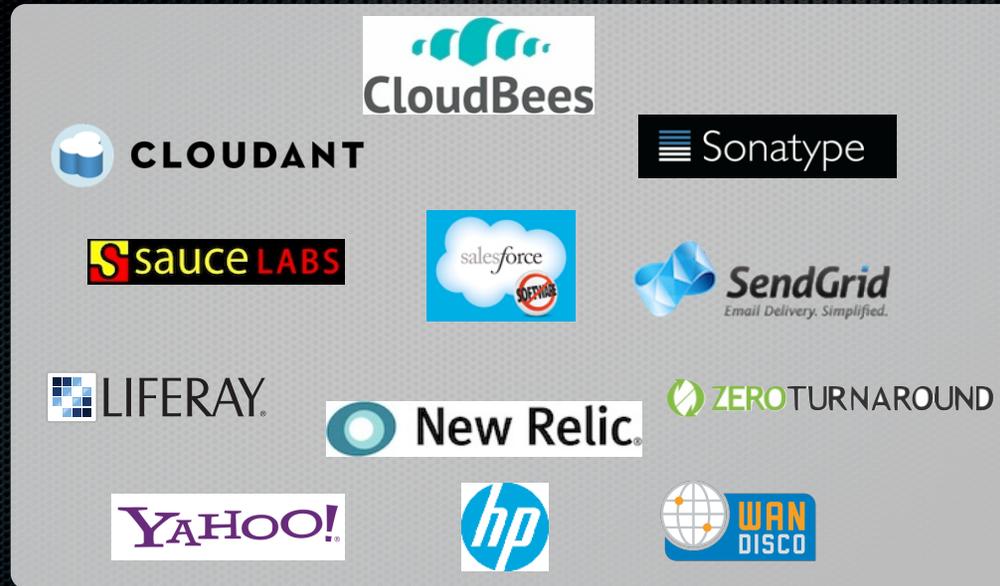
 @netflixoss



If you found that interesting, you can read more at these links, especially the jobs link :)

Check out the Netflix OSS twitter account for announcements around our open source efforts.

# Thank You To Our Sponsors



A big thanks to Cloudbees and the sponsors of the conference for allowing us the opportunity to speak to everyone

Thank you

Questions?

 @bmoyles @garethbowles



And thanks to you all for listening. Questions?