Continuous Delivery

Moritz Lenz <moritz@faui2k3.org>

https://perlgeek.de/

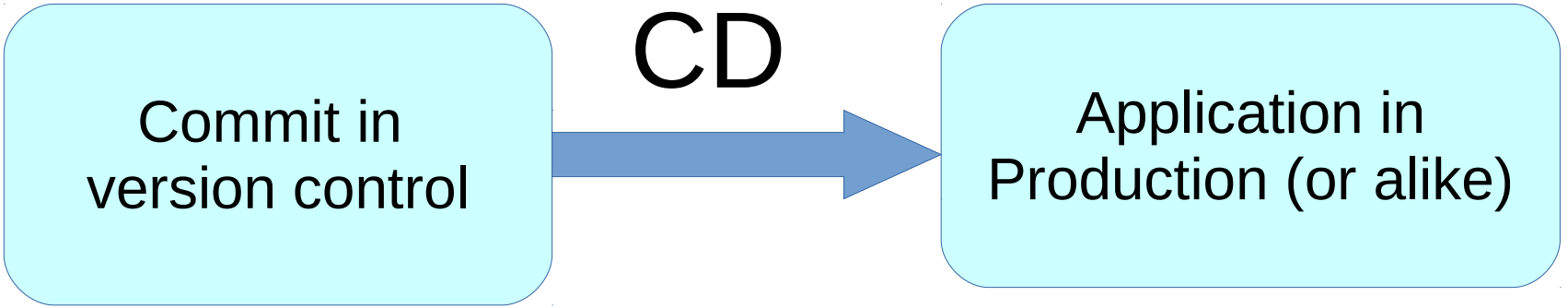
https://deploybook.com/

Contents

- What is Continuous Delivery
- Why?
- Requirements
- Anatomy of a Delivery system
- Example project
- Experience

What is Continuous Delivery?

Deploying every commit that is good enough to production or a production-like environment



What is Continuous Delivery?

- Building
- Determine which commits are “good enough” (CI, Various testing stages)
- Deployment to environments
- Actual deployment to production can still be a business decision

Why?

Why CD? – Save time

How much time do you spend

- Preparing releases?
- Doing the deployments?
- Fix up problems afterwards?
- Spend time in meetings on how to avoid the problems caused by deployments?

Boring, repetitive, error-prone tasks should be automated!

Why CD? – Allow Scaling

Manual deployments don't scale to:

- Several environments
- Many machines
- Many applications

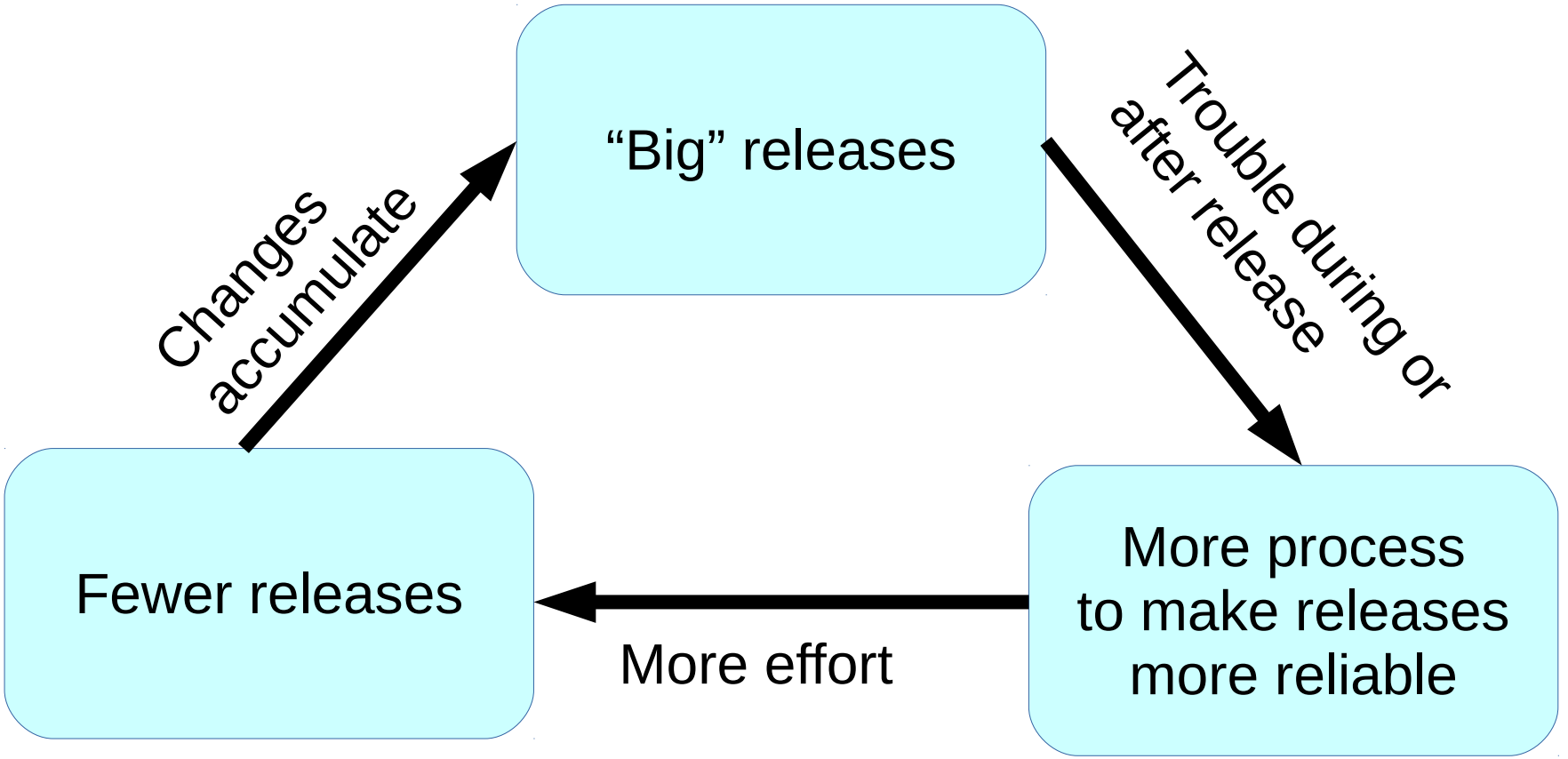
Why CD? – Time to Market

- Release cycles put a lower bound on the time from idea (or bug report) to value delivered
- Applies to **everything,** even one-line CSS changes
- (unless you bypass the normal deployment process for “small” fixes, which gives you other problems)

Why CD? – Shorter feedback cycle

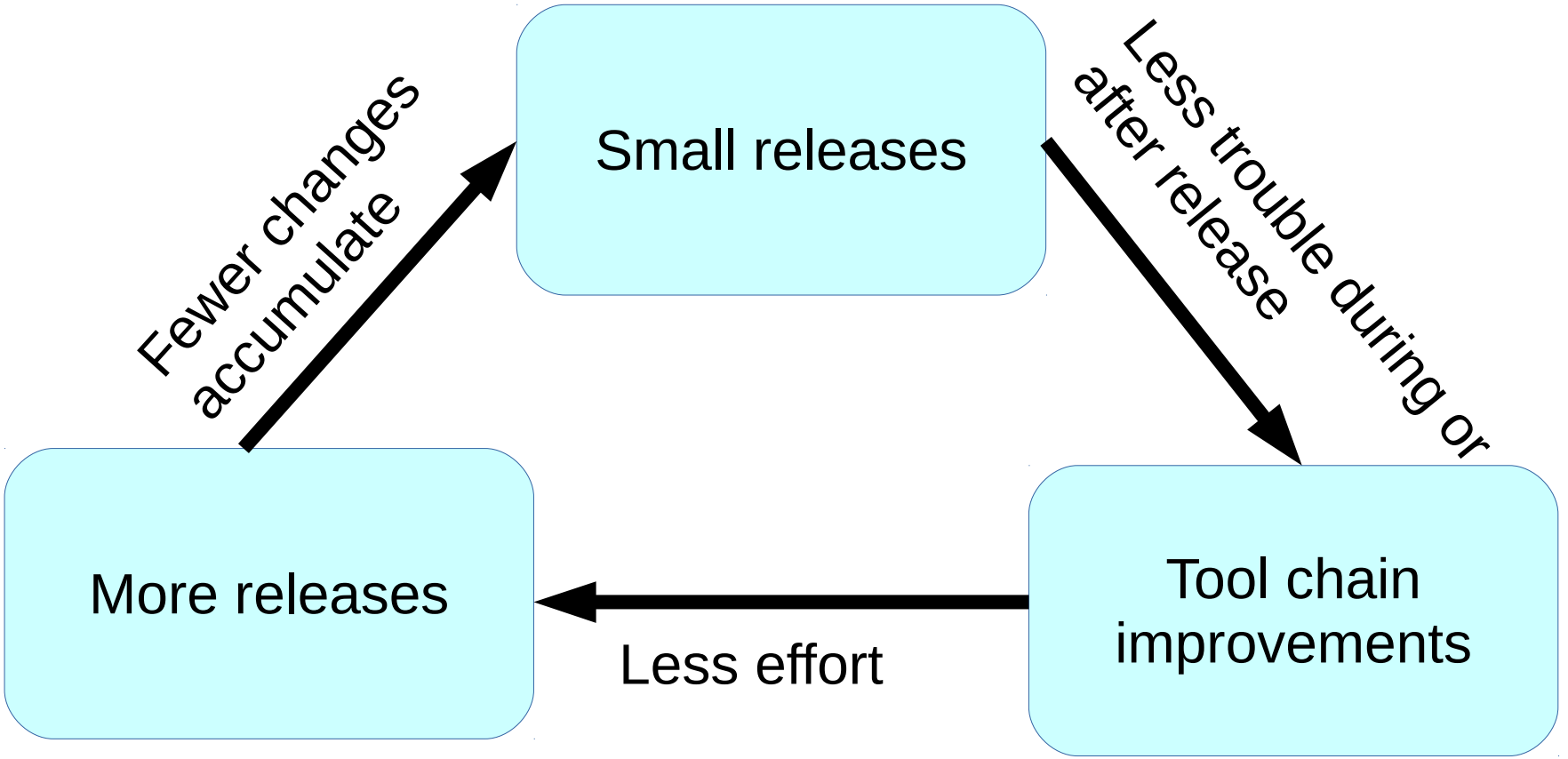
- Specifications are never perfect. Only multiple iterations of feedback and improvements produce good products
- The shorter the feedback cycle, the more feedback you can get
- If the feedback cycle is too long, users won't bother to give feedback

Why CD? – Reliability



The Vicious Cycle of big, manual releases

Why CD? – Reliability



The Virtuous Cycle of small, automated releases

Why CD? – Architectural Freedom

The more work a deployment is, the more pressure you will feel to put a new feature into an existing component.

Microservices: dozens or hundreds of separately deployed components.

Why CD? – Fancy QA

You could:

- Deploy two version of a service
- Route requests to both versions, compare results
- Record differences and timings

… but not practical if you deploy it all manually

Every good commit to production?

- Regulations might prevent that
- Schedules in contracts might prevent that
- The need for manual review might prevent that

You can still do Continuous Delivery by deploying to a staging environment.

Can I do Continuous Delivery?

Requirements: Version Control

- Your code must be under version control
- Your build tools must be under version control
- Your requirements and environment should be under version control (but you can start without it).

Requirements: Infrastructure

You must have or create the infrastructure to

- Build your software
- Test it
- Deploy it

If it's not there yet, make sure to talk to the operators!

In case of lock-down: deploy to a non-locked-down environment at least.

Requirements: Tests

You need automated tests. Enough to make you feel confident that a new version doesn't break everything/too much/anything.

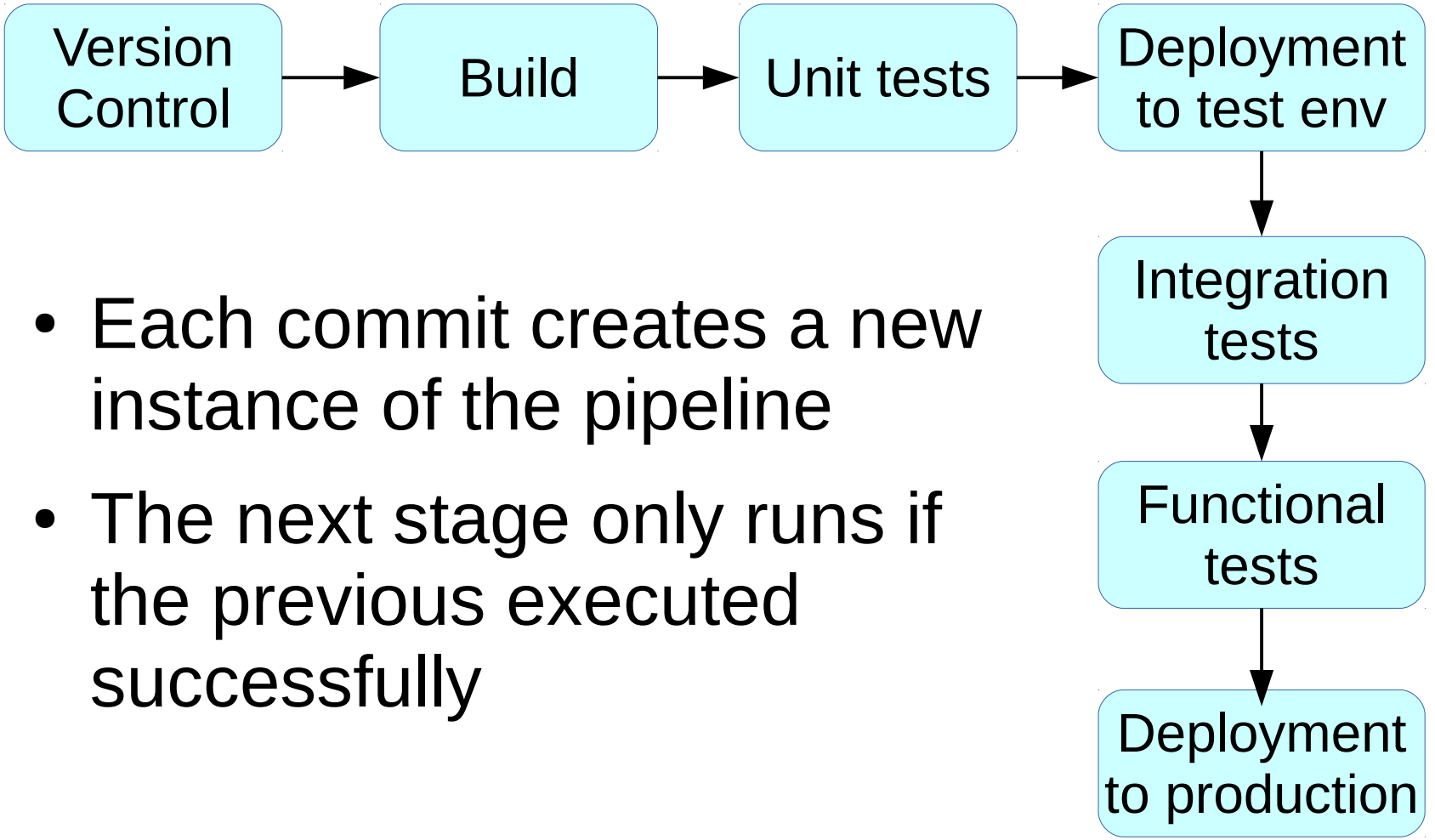
Integration tests and functional tests are very valuable here.

Soft requirements: Server software

- It is easier to deploy a server-side application (web app, service).
- If users install your software on their own machines, you can only deploy to a repository and have the software auto-update.

Anatomy of a Deployment System

Architecture: Pipeline



- Each commit creates a new instance of the pipeline
- The next stage only runs if the previous executed successfully

Technology: Packaging Format

- Use something that resolves dependencies for you (not just plain tar balls/rsync). Or fat-package them.
- If you have operators: something that your operators understand

  Good choices: System packages (.deb, .rpm, .msi), Containers

Technology: Repository

- Dependency resolution requires a pull model
- Repository tooling depends on the actual package format
- Aptly (.deb), Pulp (.rpm, .deb, python, …), CPAN::Mini + CPAN::Mini::Inject, ...

Technology: Installation

- “apt-get install” “zypper install” or “cpanm” or …
- Or a tool that automates those installers: ansible, rex, salt

Technology: Integration tests

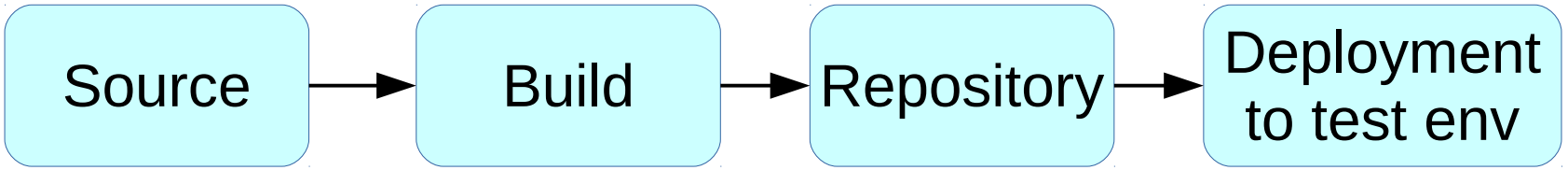
- WWW::Mechanize::Test
- wget
- curl
- expect
- Various monitoring plugins
- ...

Technology: Pipeline Organization

- Go Continuous Delivery
- Jenkins with plugins
- Atlassian Bamboo

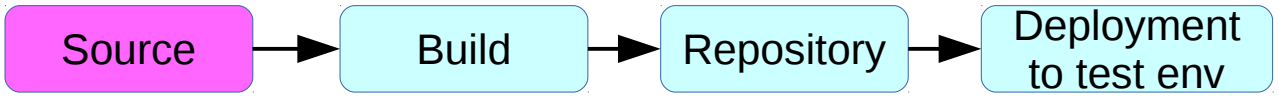
An Example Project

https://github.com/moritz/package-info/



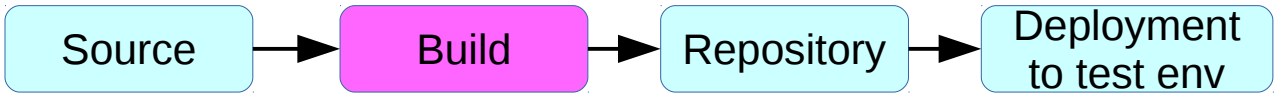
Example Project: Code

```
#!/usr/bin/perl
use Mojolicious::Lite;

plugin 'Config';

get '/' => sub {
    my $c = shift;

    $c->render(text => scalar qx/dpkg -l/, format => 'text');
};

app->start;
```



Debian packaging

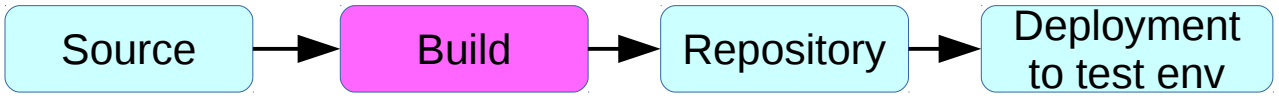
- dh_make --createorig -p package-info_0.1
- debian/install: Files to be copied to package
- debian/control: Meta data (name, deps)
- debian/postinst: Create user here
- debian/$package.service: system service file



Building

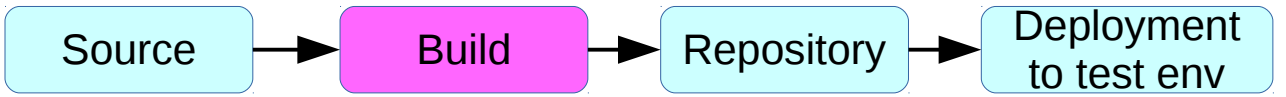
debuild -b -us -uc

- Problem: version (from debian/changelog) recycled



Building without version reuse

```
#!/bin/bash

set -e
set -o pipefail
version=$(git describe --long | sed 's/-g[a-f\d]*$//')
# Remove commit hash with leading "g"
version=$(sed 's/-g[A-Fa-f0-9]*$//' <<< "$version")
version="$version.${GO_PIPELINE_COUNTER:-0}.${GO_STAGE_COUNTER:-0}"

DISTRIBUTION=${DISTRIBUTION:-jessie}

echo $version
echo $version > ../version

DEBFULLNAME='Go Debian Build Agent' DEBEMAIL='go-noreply@example.com' debchange
--newversion=$version  --force-distribution -b  --distribution="${DISTRIBUTION:-jessie}" 'New
Version'
debuild -b -us -uc
```



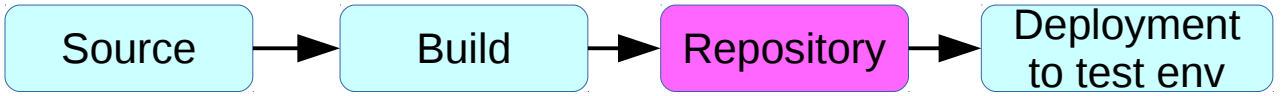
Repo Management: aptly

- Initial setup:

```
aptly repo create -distribution=jessie
-architectures=amd64,all,i386 staging

aptly publish repo staging
```

- For each package:

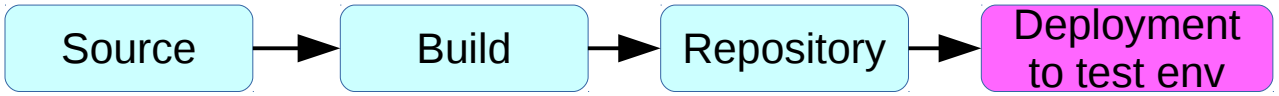
```
aptly repo add staging $debfile

aptly publish update jessie
```



Package Installation: ansible

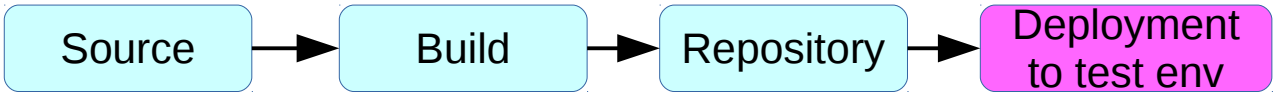
- Inventory (“hosts”) file:

```
[web]
www01.staging.example.com
www02.staging.example.com
[all:vars]
ansible_ssh_user=root
```



Package Installation: ansible

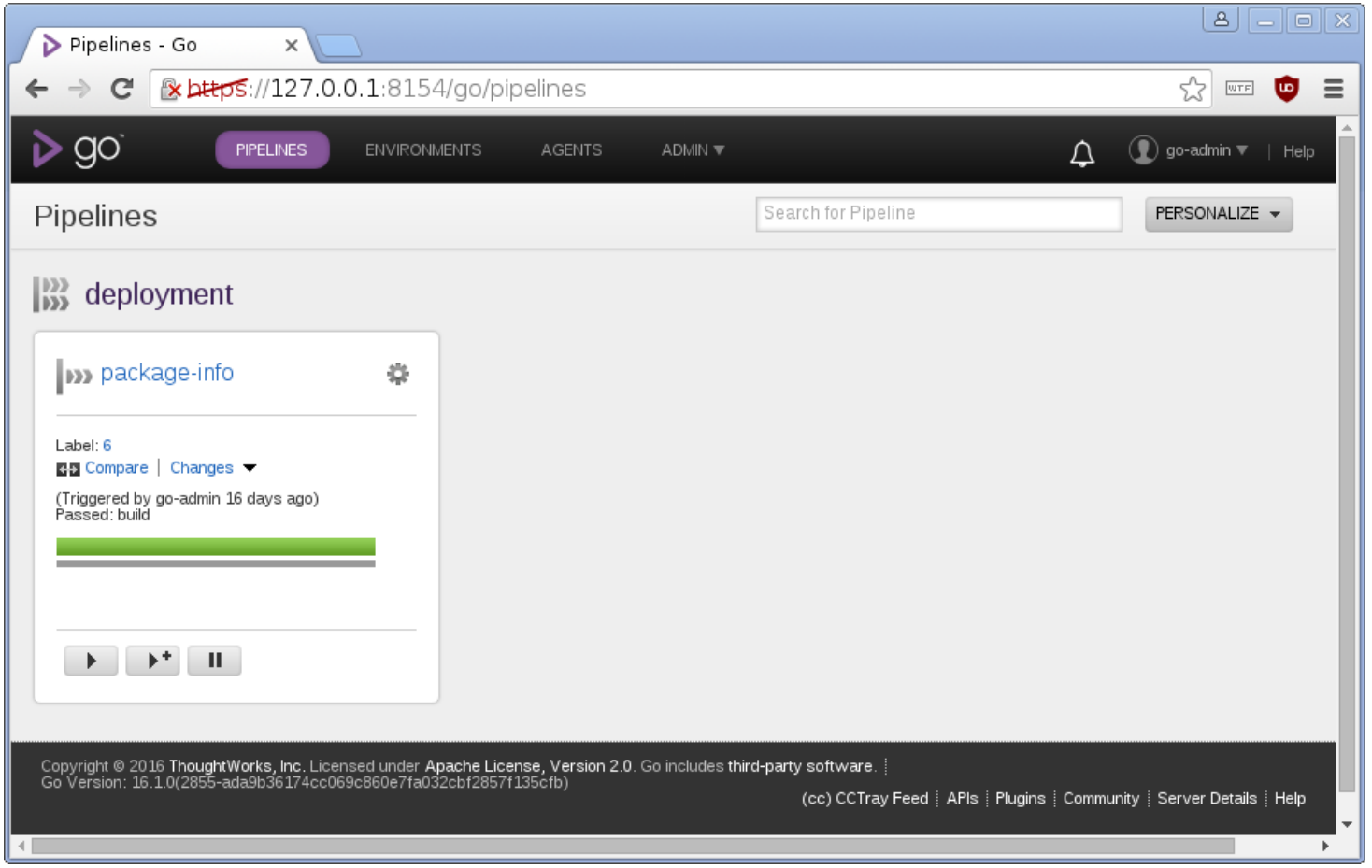
```
ansible -i staging web  \
 -m apt \
 -a 'name=package-info update_cache=yes
state=latest'
```



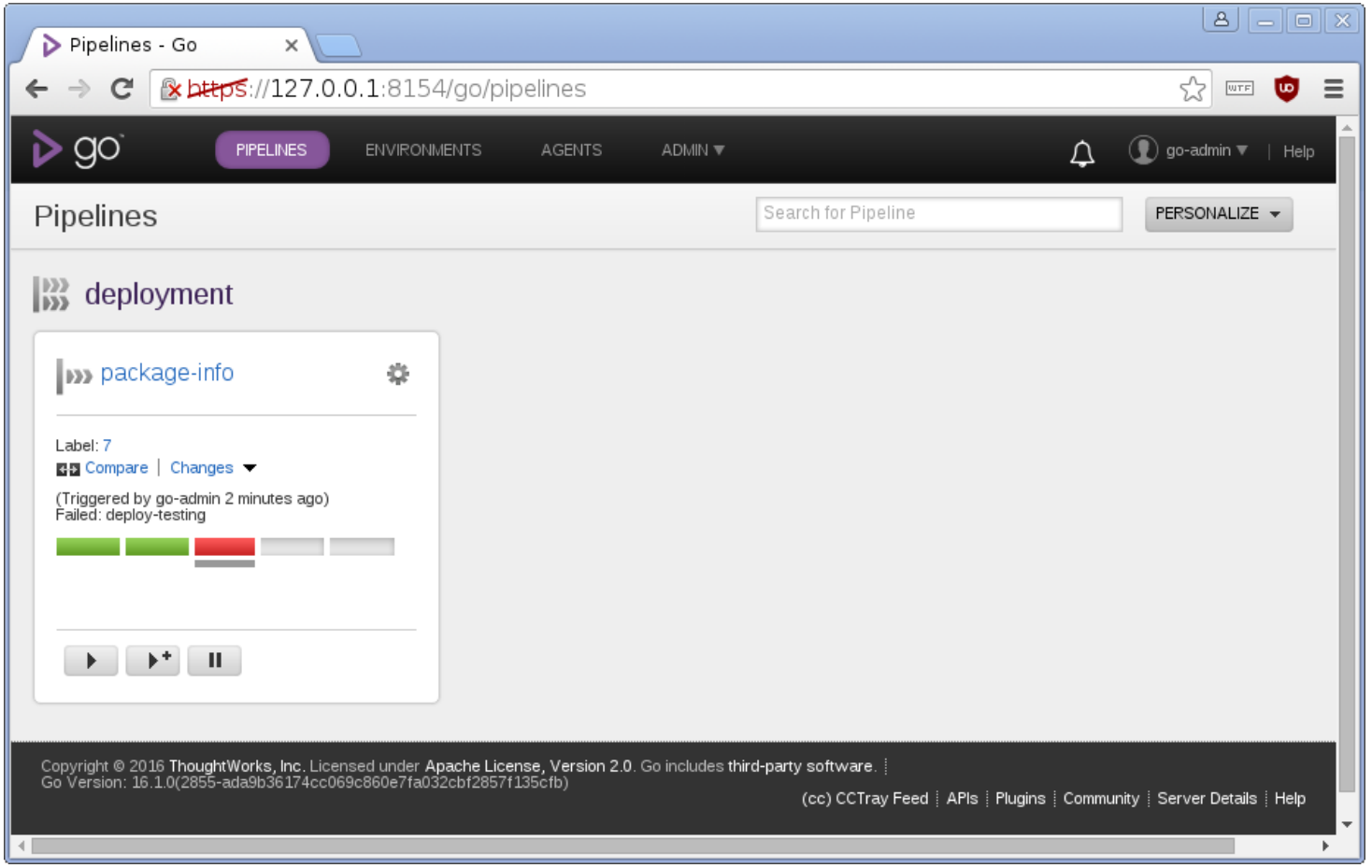
Coordination: GoCD

```
<materials>
  <git url="https://github.com/moritz/package-info.git" dest="package-info" />
  <git url="https://github.com/moritz/deployment-utils.git" dest="deployment-utils" materialName="deployment-utils" />
</materials>
<stage name="build" cleanWorkingDir="true">
  <jobs>
    <job name="build-deb" timeout="5">
      <tasks>
        <exec command="/bin/bash" workingdir="package-info">
          <arg>../deployment-utils/debian-autobuild</arg>
        </exec>
      </tasks>
      <artifacts>
        <artifact src="version" />
        <artifact src="package-info*_*" dest="package-info/" />
      </artifacts>
    </job>
  </jobs>
</stage>
```

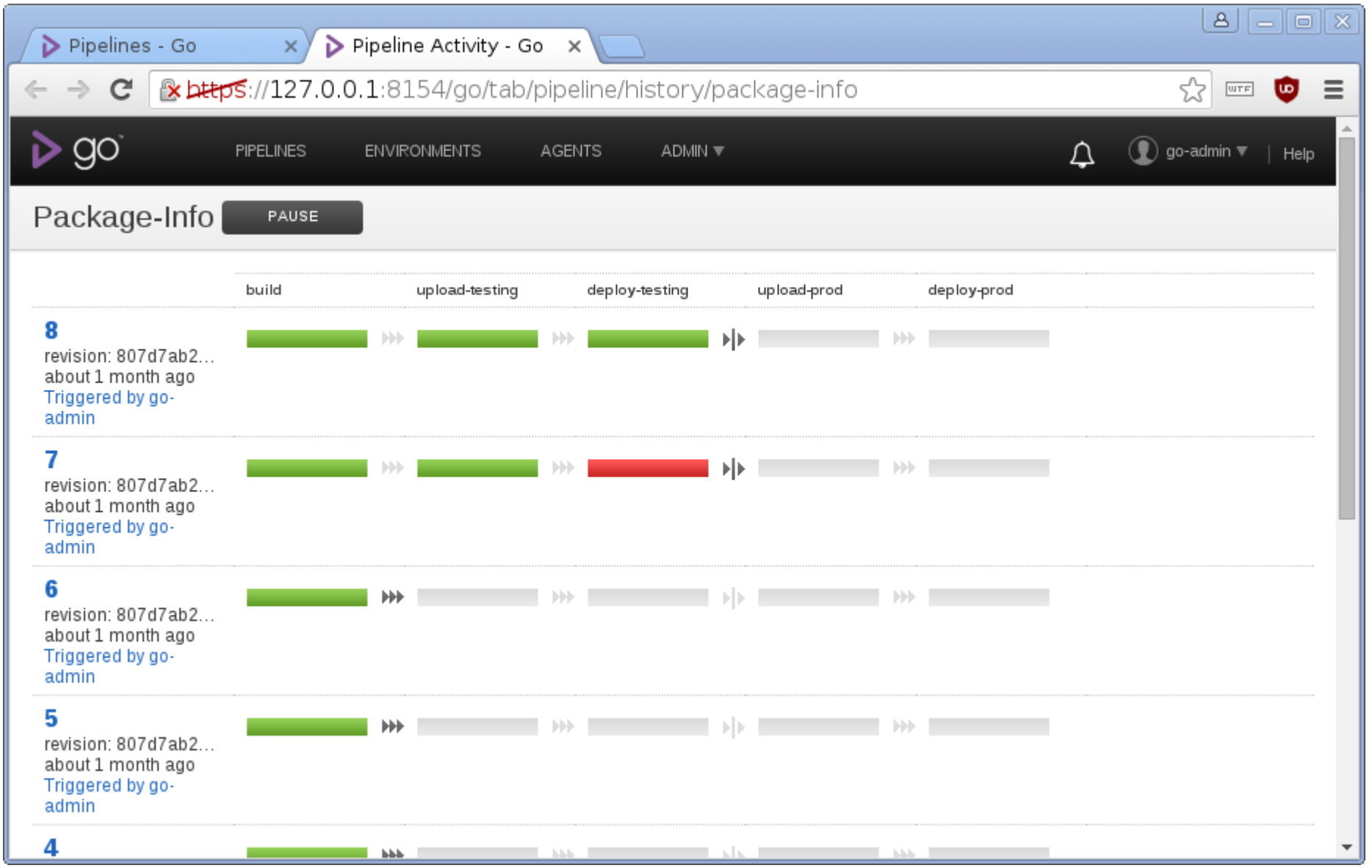
GoCD: Pipeline overview



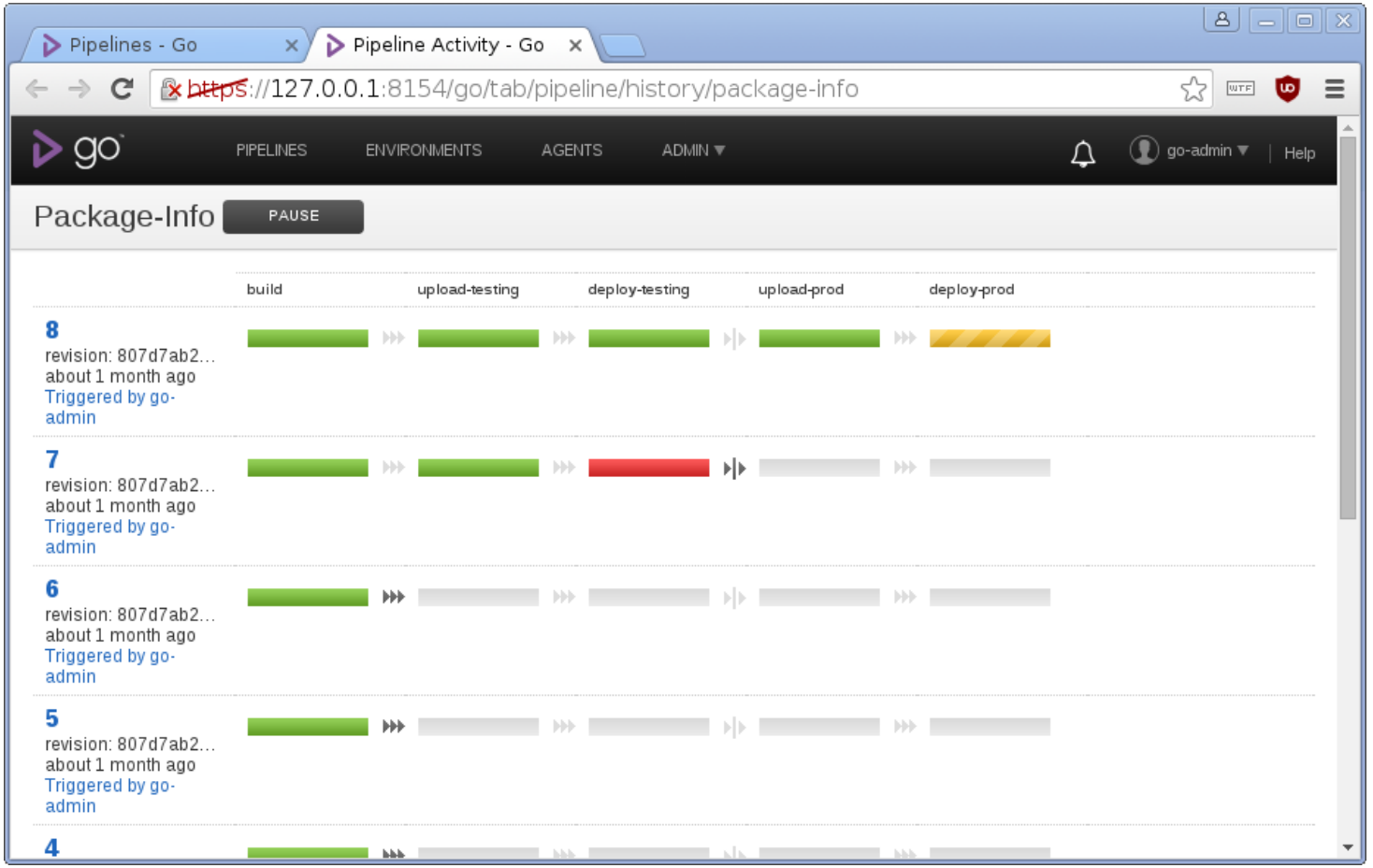
GoCD: More Stages



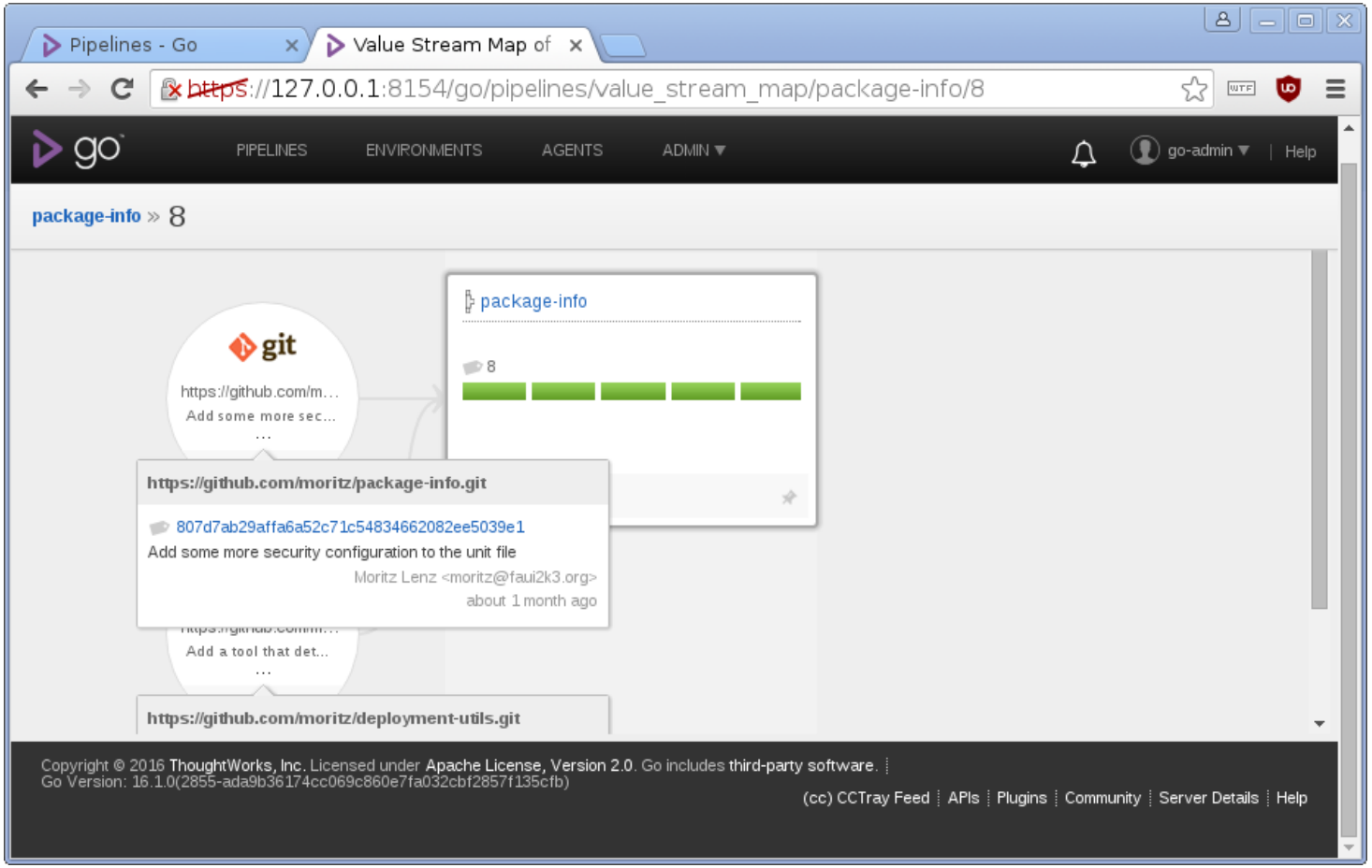
Go: History View, Manual Approval



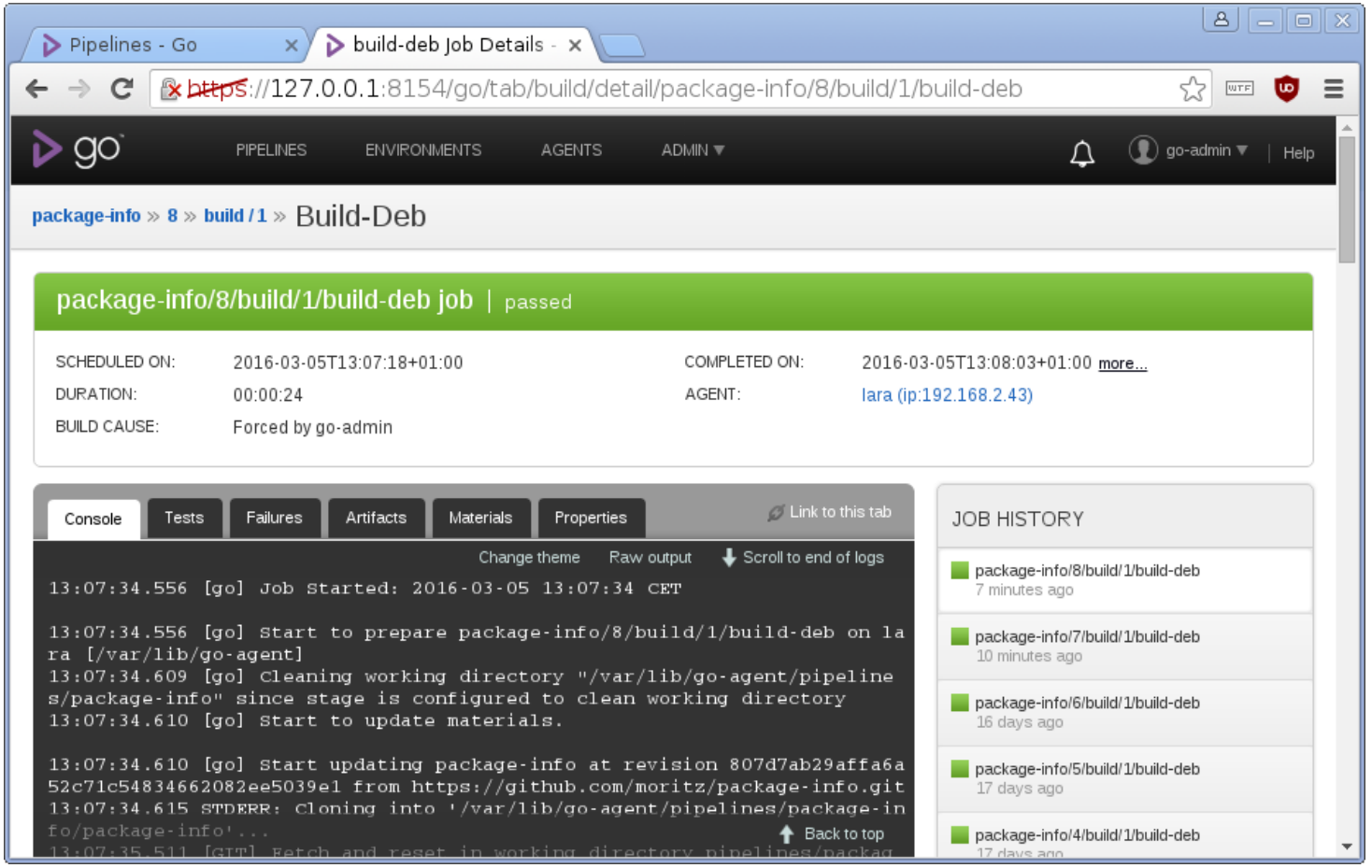
GoCD: After manual approval



GoCD: Value Stream Map



GoCD: Stage Details



Experience

CD Dreams seem unreachable

The ideals of Continuous Delivery:

- Fully automated deployments
- Fully automated rollbacks
- New environments on the press of a button

Seem to be pretty steep goals

Small steps towards CD

- Starting with automating the steps that cause most work is a good approach
- Partial automation much better than fully manual builds and deployment
- Rollbacks are really just deployments of a defined, older versions

Time to Market Matters for Bugfixes

- For bugs that affect lots of users, fixing it quickly in production is crucial
- Having a fast (minutes, not hours), automated pipeline is tremendously helpful

Managing State is HARD

- After about a year of part-time CD efforts, not yet automated at $work
- How to handle DB schema changes? Either the change or the rollback can lose data
- Needs cooperation from the development process (introduce columns first, make them `NOT NULL` later)

“Big” tools have a steep learning curve

- GoCD: Quite some time getting used to
- Initial frustration
- Many secondary benefits: version tracking, notifications, permissions, operations-friendly UI

The Advantages are real

- Time savings
- Faster feedback
- Less stress
- Scaling to more components (currently 24)
- All the successful, big companies (Netflix, Google, Facebook, Spotify, LinkedIn) do it.

A worthwhile experience.
You should try it.

Thank You

Interested? Check out
https://deploybook.com/