Efficient Parallel Testing with Docker

Laura Frank

Engineer, Codeship

dockercon 16

CODESHIP

dockercon 16

CenturyLink™

panamax™

IMAGE LAYERS

HP Helion

Agenda

1. Parallel Testing: Goals and Benefits
2. DIY with LXC
3. Using Docker and the Docker Ecosystem

Parallel Testing

GOAL

Create a customizable, flexible test environment that enables us to run tests in parallel

Why?

- Deploy new code faster
- Find out quickly when automated steps fail

If you're still not sure why testing is important, please talk to me at the Codeship booth.

Where?

- For local testing, e.g. unit and integration tests run by a development team
- On internal CI/CD systems
- As part of a hosted CI/CD solution (like Codeship)

How?

- Performance optimization for serial testing tasks is limited
- Split up testing tasks
- Use containers to run multiple tests at once

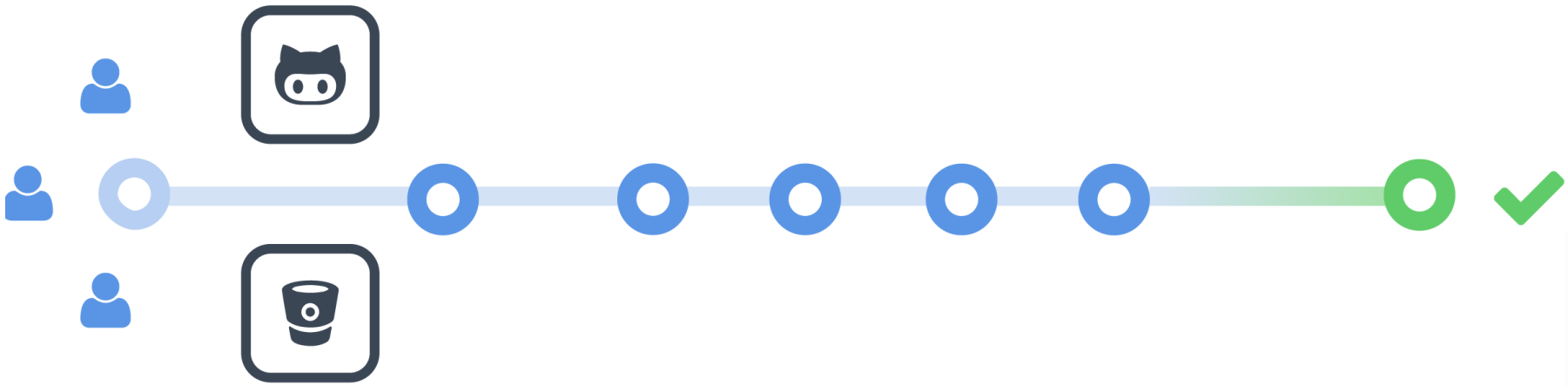
TASK PARALLELISM

Run tasks across multiple processors in parallel computing environments

Distributed Task Parallelism

A distributed system of containerized computing environments takes the place of a single multiprocessor machine

A container is a process, not a small VM



Goal: Shorter Feedback Cycles

Spend less time waiting around for your builds to finish.

- **Ship newest code to production faster**
- **Be alerted sooner when tests fail**

Goal: More User Control

Developers should have full autonomy over testing environments, and the way tests are executed.

- **Move testing commands to separate pipelines**
- **Designate commands to be run serially or in parallel**
- **Declare specific dependencies for each service**

Why not VMs?

- Isolation of running builds on infrastructure
- Challenges with dependency management
- No clean interface for imposing resource limits
- Infrastructure is underutilized which makes it expensive

Containers, duh!

- Impose resource limits and utilize infrastructure at higher capacity
- Run customer code in isolation
- Provide consistent build environment across many build runs
- Run testing tasks in parallel 👍

DIY with LXC

Codeship has been powered by containers since the very beginning

2011: A Brief History Lesson

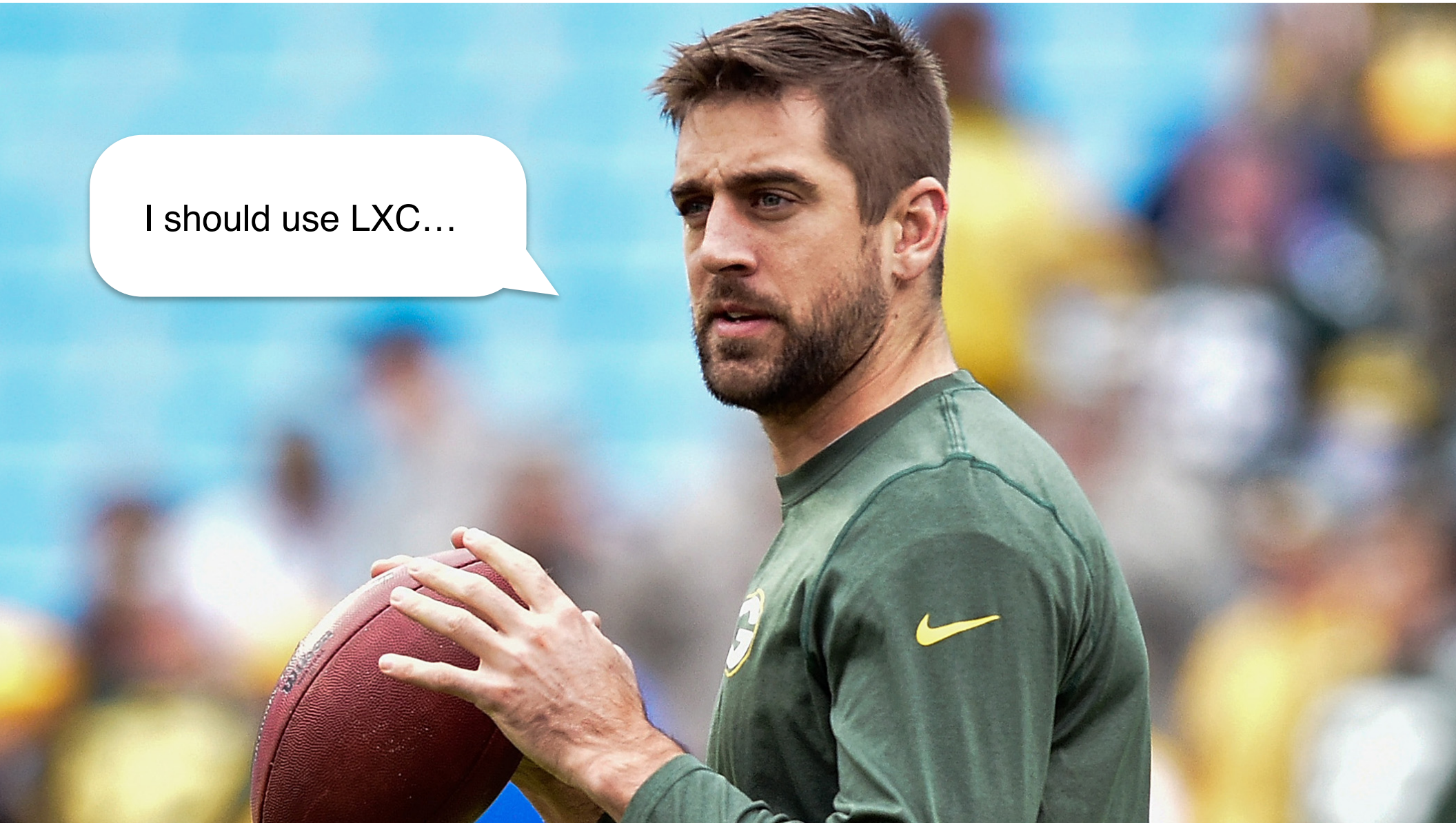
Flowing salty water on Mars

International Year of Forests

Preparations ahead of 12.04 Precise release shipped to support LXC improvements

Codeship was founded

Green Bay Packers won Super Bowl XLV


I should use LXC…

Why LXC?

- Impose resource limits and utilize infrastructure at higher capacity
- Run customer code in isolation
- Provide consistent build environment across many build runs
- Enable parallel testing jobs
- Can programmatically automate creation and deletion

Checkbot (Codeship Classic)

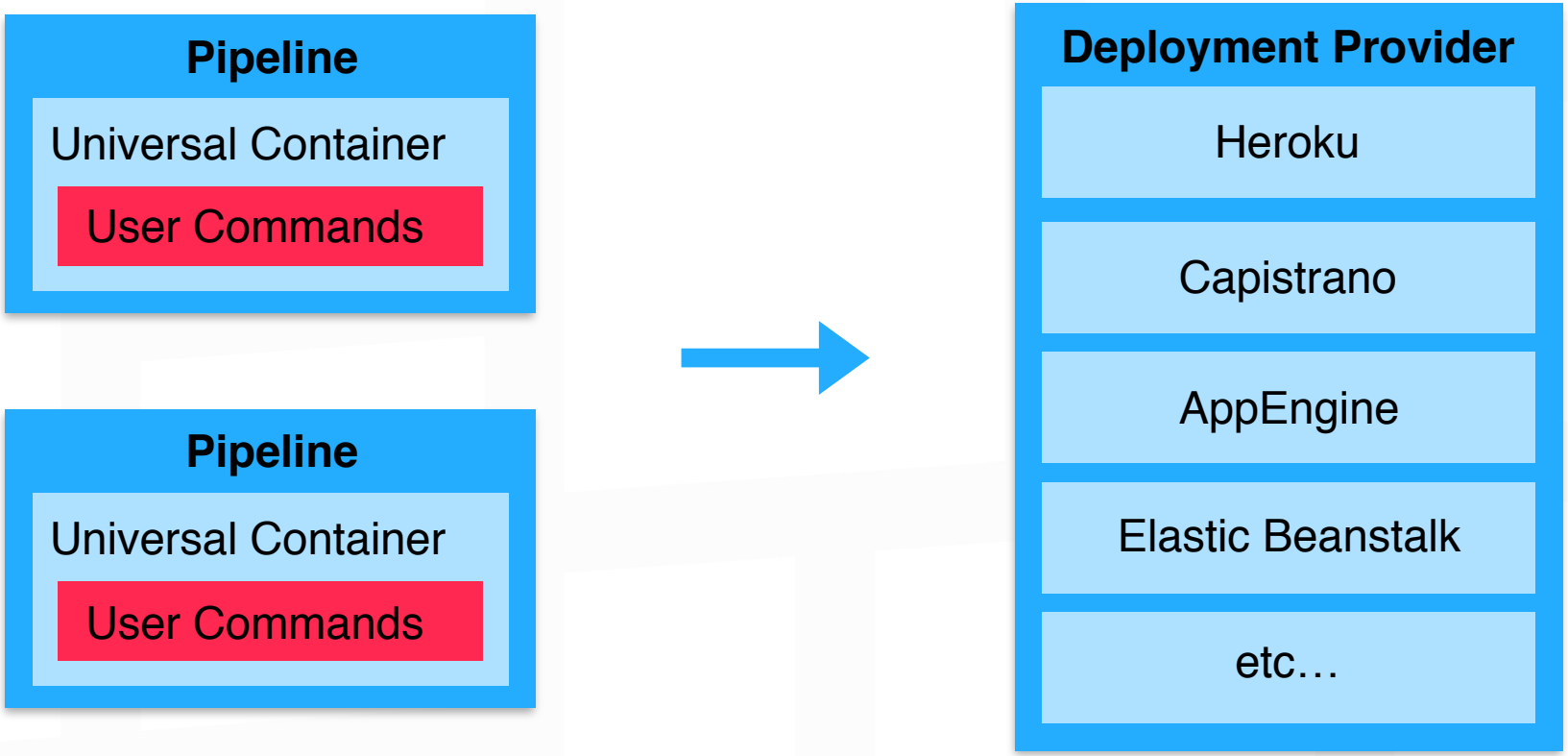
- Still running in production as our classic infrastructure
- Well-suited for users who want 1-click test environments without much customization
- Compromise flexibility for ease of use

Checkbot

39K builds per day
7.8M builds per year

Architecture

- Universal Container with provided dependencies
- Run builds in isolation from one another
- Implement parallel testing pattern using *pipelines* with ParallelCI
- Users can have N pipelines running in isolation during a build


Pipeline
Universal Container
User Commands
Pipeline
Universal Container
User Commands
Deployment Provider
Heroku
Capistrano
AppEngine
Elastic Beanstalk
etc…

Limitations

- Parity between dev and test
- Can’t really debug locally
- No useable interface between user and container
- Resource consumption is too high

While using straight-up LXC solved some of our technical problems, it didn't solve any of our workflow problems

We weren’t able to provide the best, most efficient product to our customers (or ourselves)

Using Docker and the Docker Ecosystem

dockercon 16

GOAL

Create a customizable, flexible test environment that enables us to run tests in parallel

Big Wins with Docker

Even before 1.0, Docker was a clear choice

- Support and tooling
- Standardization
- Community of motivated developers

Using Docker allowed us
to build a much better testing platform
than with LXC alone

Codeship Jet

A Docker-based Testing Platform

- Development started in 2014
- First beta in 2015
- Official launch February 2016

A Docker-based Testing Platform

Built with Docker

in order to support Docker workflows

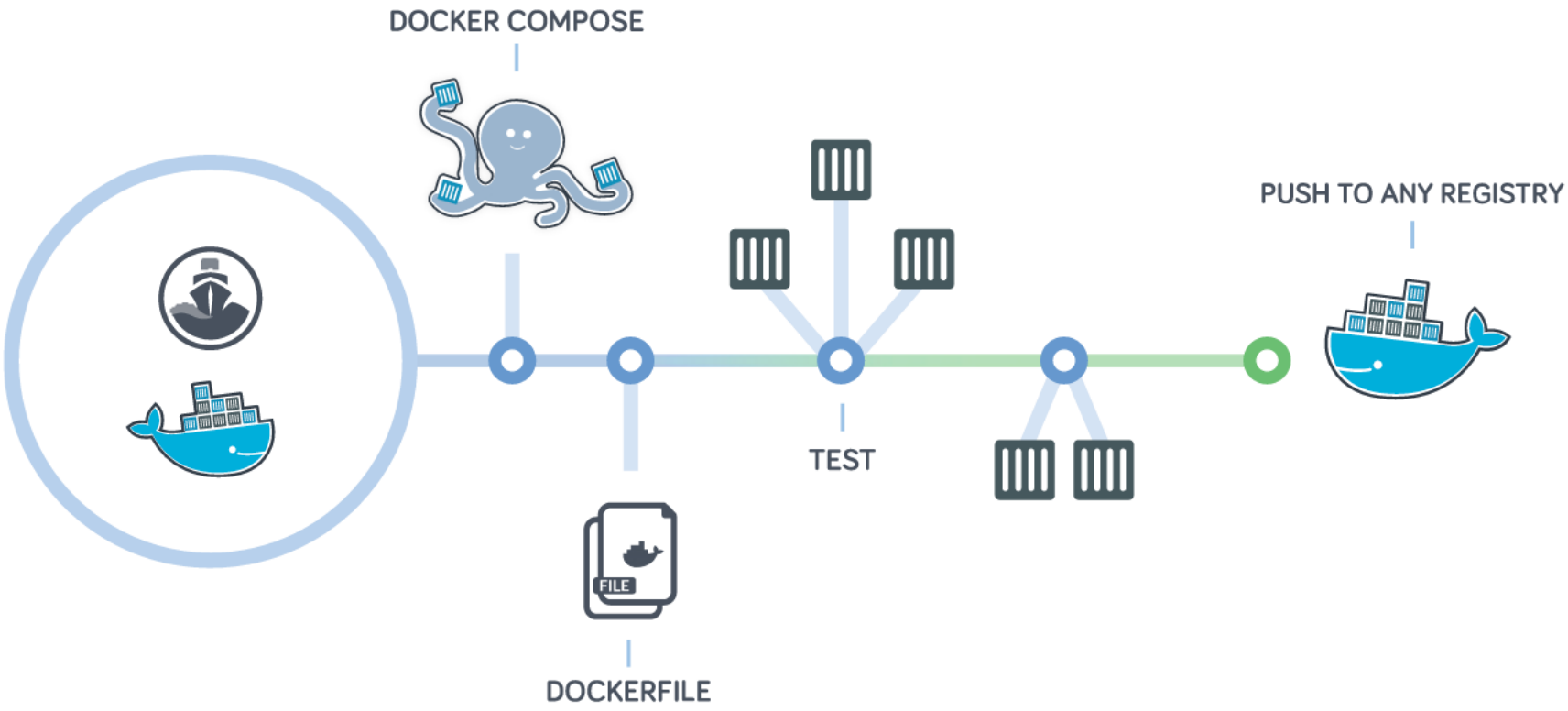
Codeship Jet

2.3K builds per day
~250K total builds

Why Docker?

- Docker Compose: service and step definition syntax
- Docker Registry: storage for images; remote caching*
- Docker for Mac and Windows: give users ability to reproduce CI environments locally

*not for long!


DOCKER COMPOSE
PUSH TO ANY REGISTRY
TEST
FILE
DOCKERFILE

Docker Compose

- Provides simplicity and a straightforward interface
- Developers can use existing `docker-compose.yml` files with Codeship
- Ensure parity in dev, test, and production
- Use similar syntax for testing step definitions to get users up and running faster

The workflow tools provided by Docker are indispensable

Parallel Testing with Docker

dockercon 16

Managing containers with Docker allowed us to improve our parallel testing workflow

A New Parallel Workflow

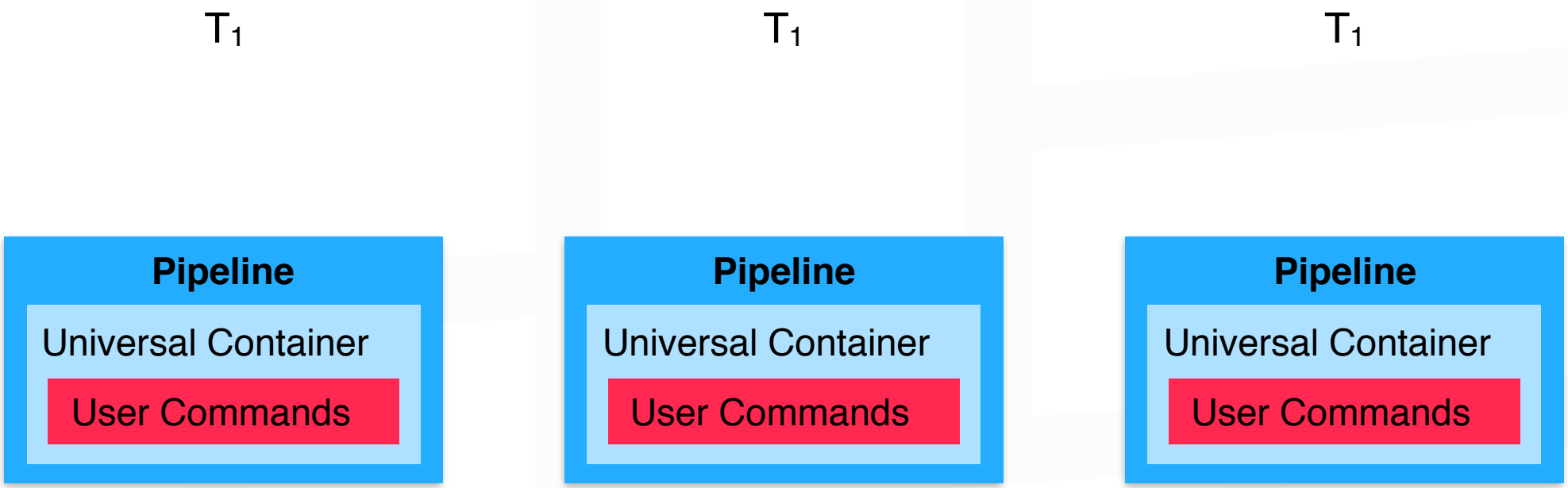
- Introducing services adds additional layer of flexibility
- Loosen coupling between steps and services — execute N steps against M services
- Parallel and serial steps can be grouped and ordered in any way

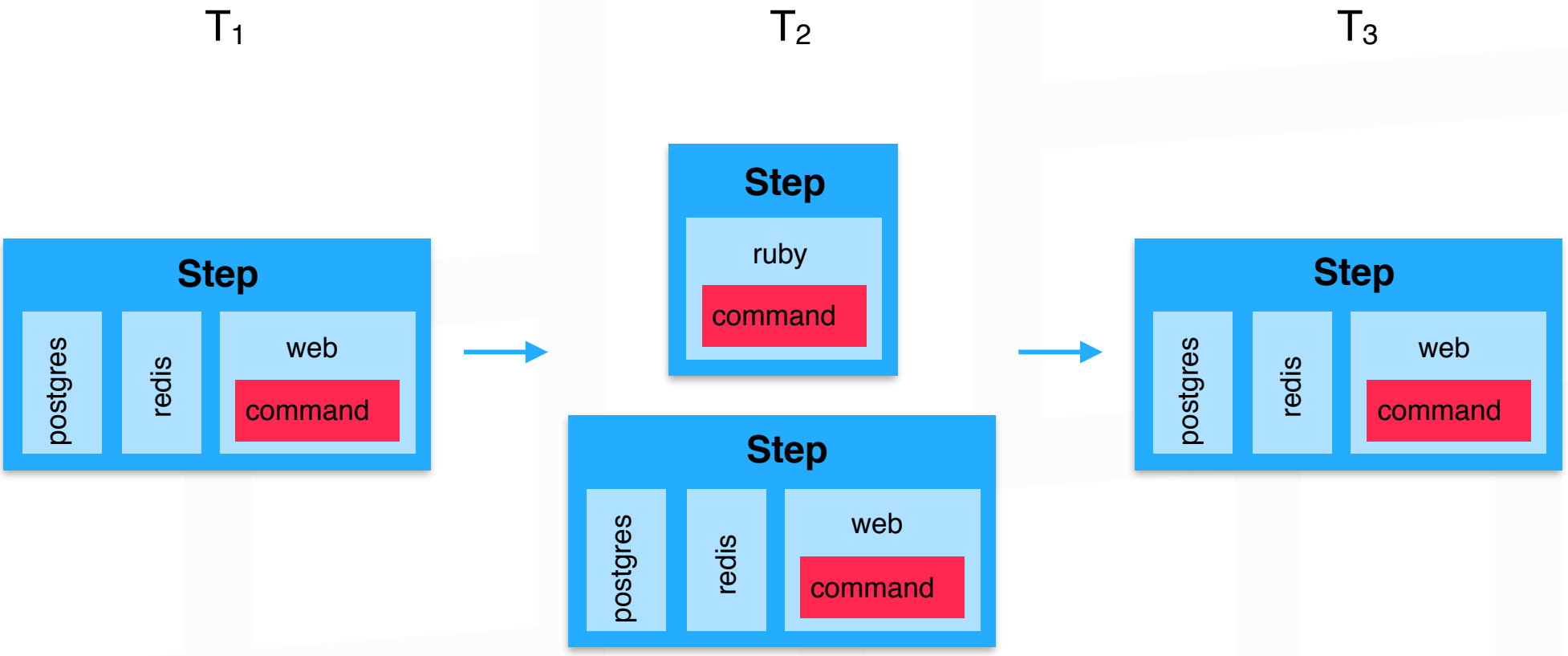
Services

- Pull image from any registry or build from Dockerfile
- Optimize service for testing tasks
- Fully customizable by the user

Steps

- Each step is executed in an independent environment
- Can be nested in serial and parallel groups
- Two functions
  - Run: execute a command against a service
  - Push: push image to registry
- Tag regex matching to run steps on certain branches or tagged releases


T1
T1
T1
Pipeline
Universal Container
User Commands
Pipeline
Universal Container
User Commands
Pipeline
Universal Container
User Commands


T1
T2
T3
Step
postgres
redis
web
command
Step
ruby
command
Step
postgres
redis
web
command
Step
postgres
redis
web
command

codeship-services.yml

★
```
db:
  image: postgres:9.5
```

★
```
app:
  encrypted_dockercfg_path: dockercfg.encrypted
  build:
    image: user/some-image
    dockerfile: Dockerfile.test
  cached: true
  links:
    - db
```

★
```
deploy:
  encrypted_dockercfg_path: dockercfg.encrypted
  build:
    dockerfile: Dockerfile.deploy
```

codeship-steps.yml

```
- type: serial
  steps:
  - type: parallel
    steps:
    - name: rspec
      service: app
      command: bin/ci spec
    - name: rubocop
      service: app
      command: rubocop
    - name: haml-lint
      service: app
      command: haml-lint app/views
    - name: rails_best_practices
      service: app
      command: bin/railsbp
```

```
- service: deploy
  type: push
  image_name: rheinwein/notes-app
  tag: ^master$
  registry: https://index.docker.io/v1/
  encrypted_dockercfg_path: dockercfg.encrypted
```

ProTip: Your push or deploy step should never be part of a parallel step group

Demo!

Docker for Mac and Windows

- All users can test locally
- Jet CLI is available at http://bit.ly/codeship-jet-tool
- Don’t have a Docker for Mac/Windows invitation yet? Totally cool, Docker Toolbox also rocks
- HUGE advantage over our previous LXC implementation

Engineering Challenges

Infrastructure

Build allocation

- Customers can choose specs for their build machines
- Machine provisioning used to be part of the build process
- Now we pool build machines
- Allocation time is ~1 second!

Performance

Image Caching

- Old way: rely on the registry for caching
- A pull gave us access to each parent layer; rebuilding the image used the local cache
- 1.10 content addressable breaking change

Performance

Image Caching

- Great news: 1.11 restores parent/child relationship when you save the images via docker save
- ETA: 1 month
- Double-edged sword of relying on external tools
  ¯_(ツ)_/¯

What's Next?

Docker Swarm

- Jet was born pre-Swarm
- We manage build machines on AWS via our own service
- Previous concerns about security — single tenancy
- Swarm (and services like Carina) are promising for the future

libcompose

- Currently use APIs directly for container-level operations (Jet was also born before Fig was popular)
- Minimal change for our users and builds, but much easier for our engineers
- Preliminary work has been completed (thanks Brendan!)

TL;DR

You can create a highly efficient parallel testing platform with LXC alone, but using Docker tools makes it better

Thank you!