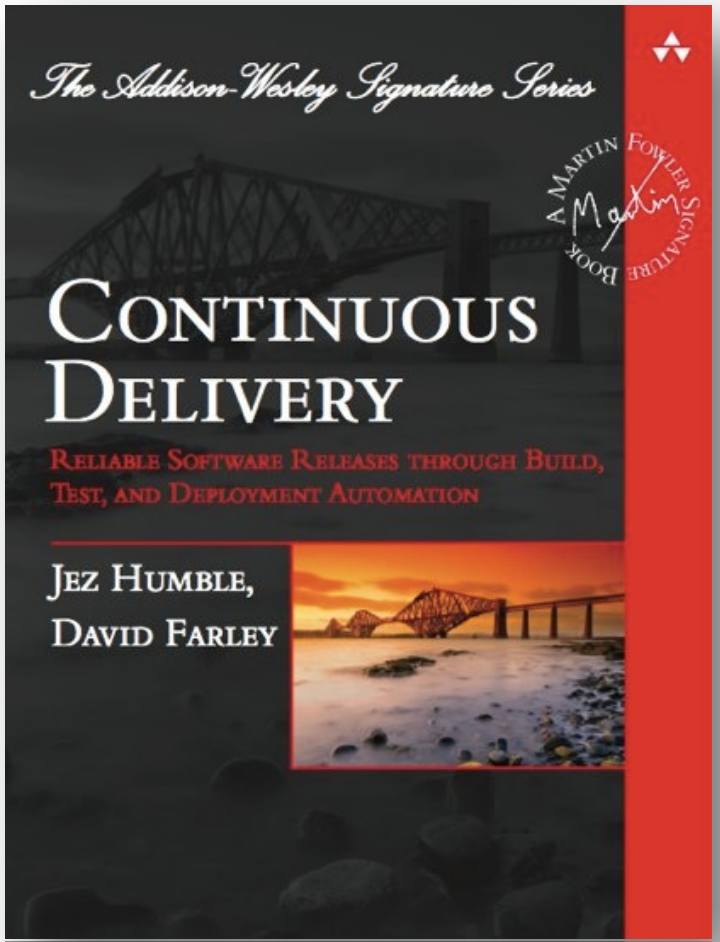
ThoughtWorks®

Continuous Delivery Workshop



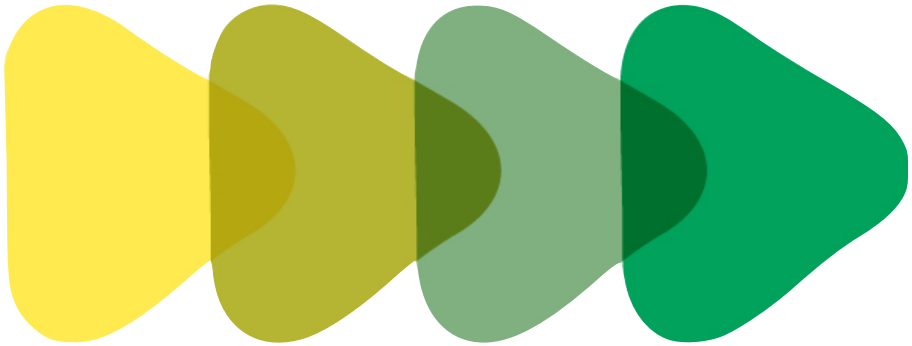
Infrastructure & Data

ThoughtWorks®

NEAL FORD

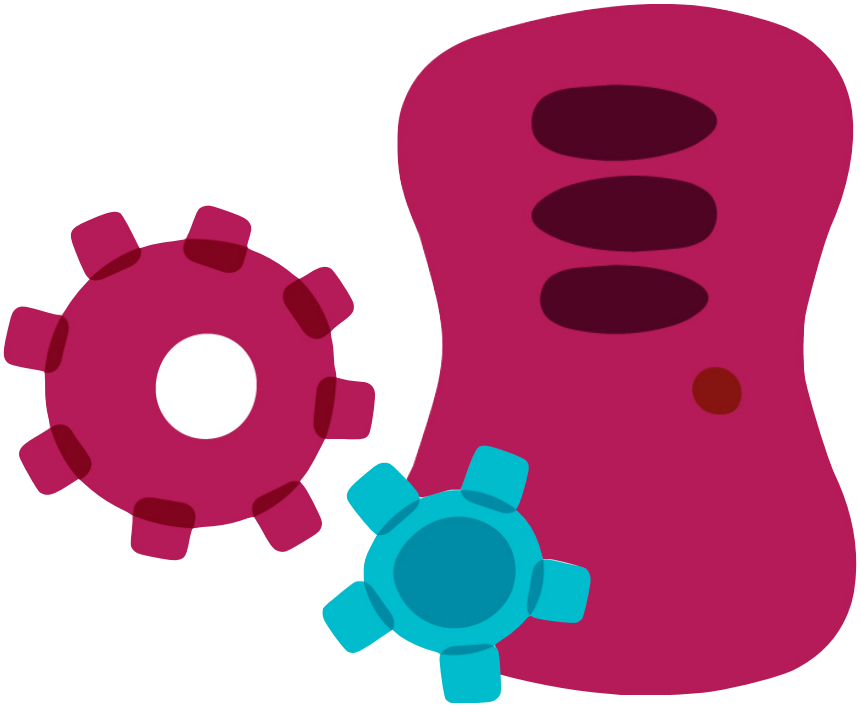
Director / Software Architect / Meme Wrangler

Workshop materials created by Jez Humble, Martin Fowler, Tom Sulston, & Neal Ford



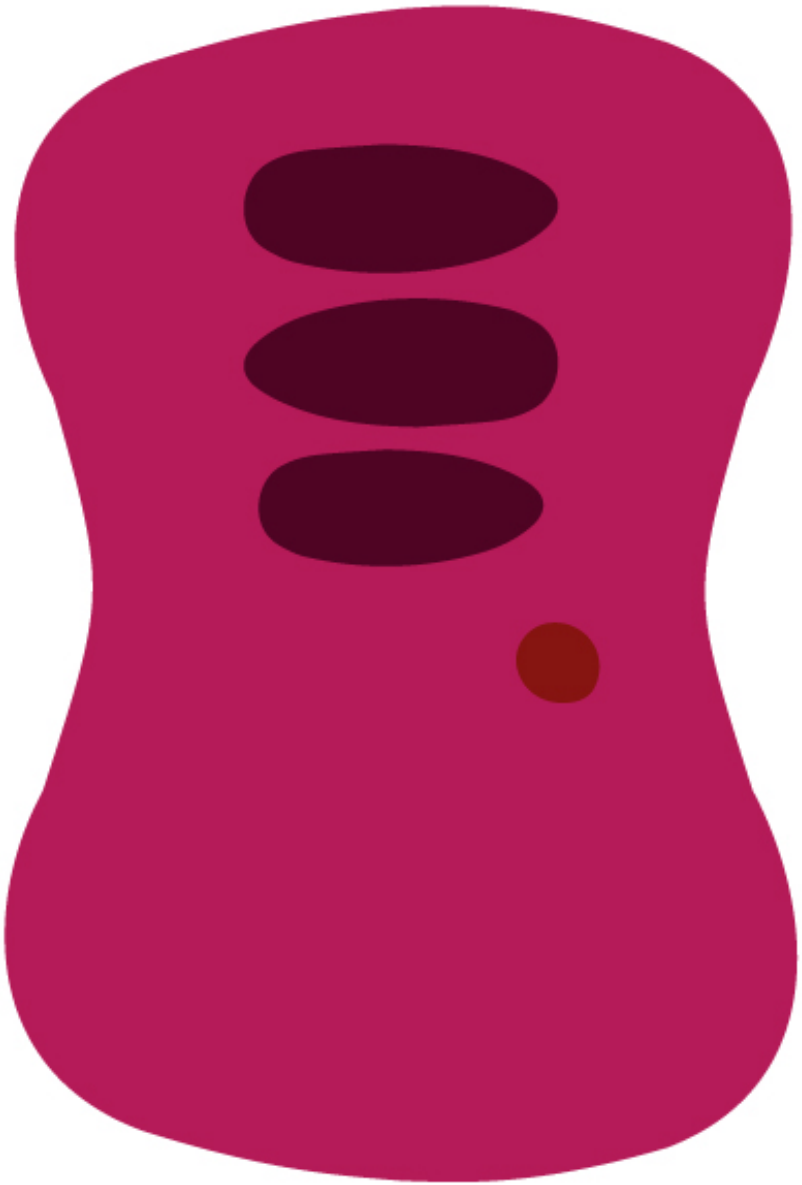
deployment pipelines

tests, synergistic practices,
incremental deployment

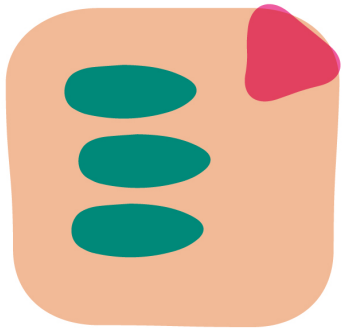


data & infrastructure

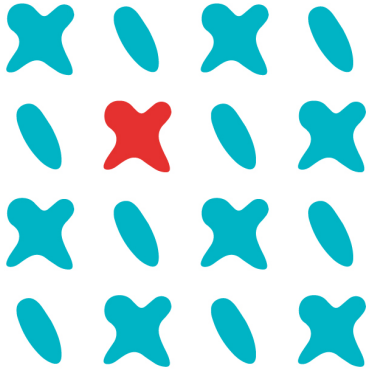
Data Management & Migration



essential complexity



persistent



impedance mismatch

bulky

accidental complexity

refactoring

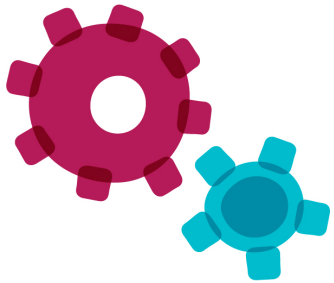
poor collaboration

late integration

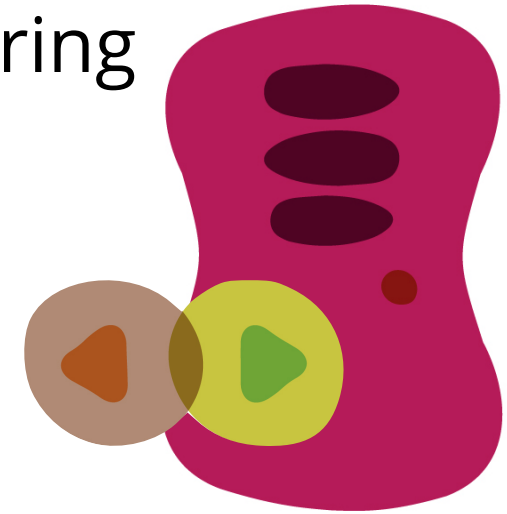
manual work

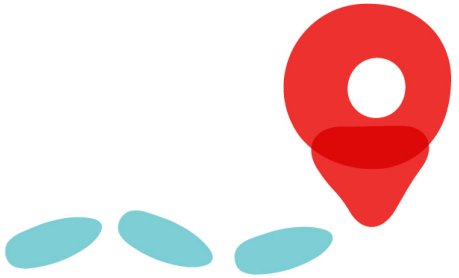
DB Evolution & Deployment

scripting all db changes incrementally



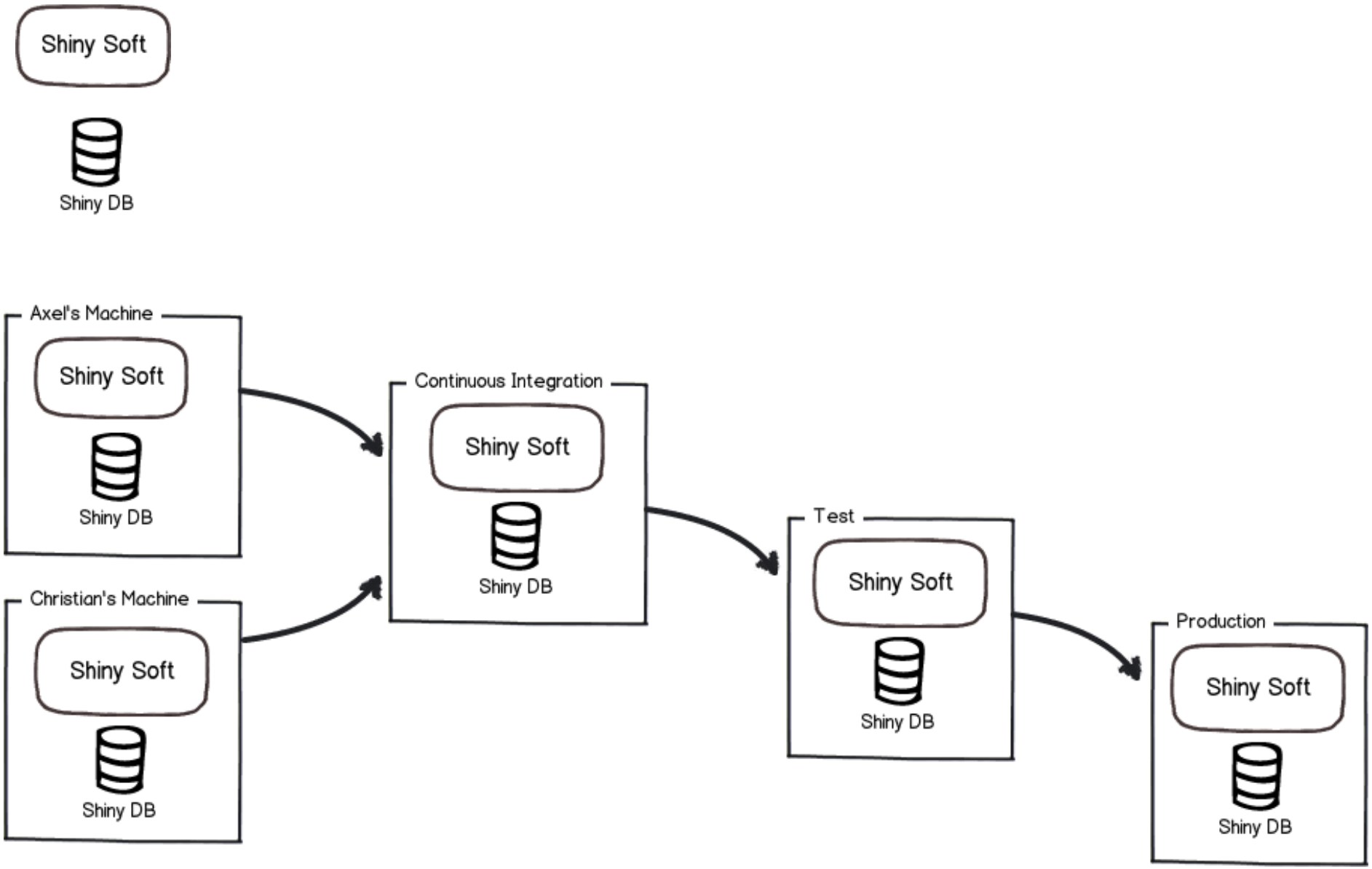
db refactoring





decouple db migration
from app migration

DbDeploy Pattern



DbBeploy Tool

db updates are code

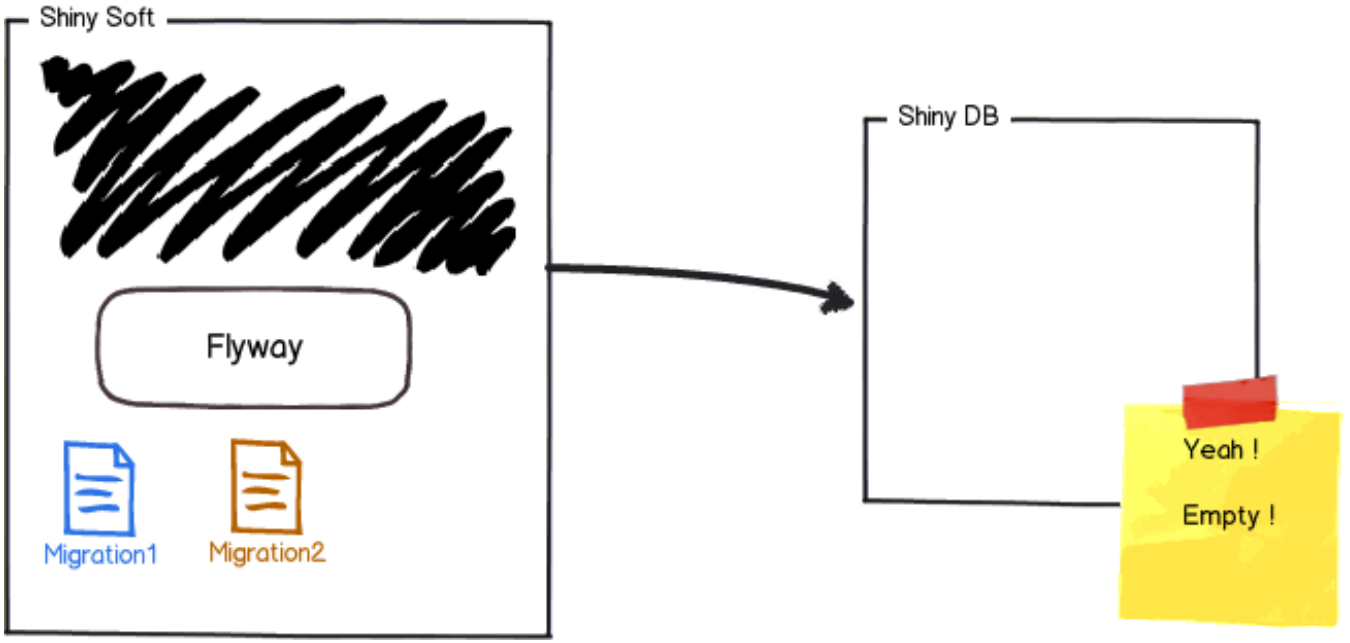
small incremental deltas

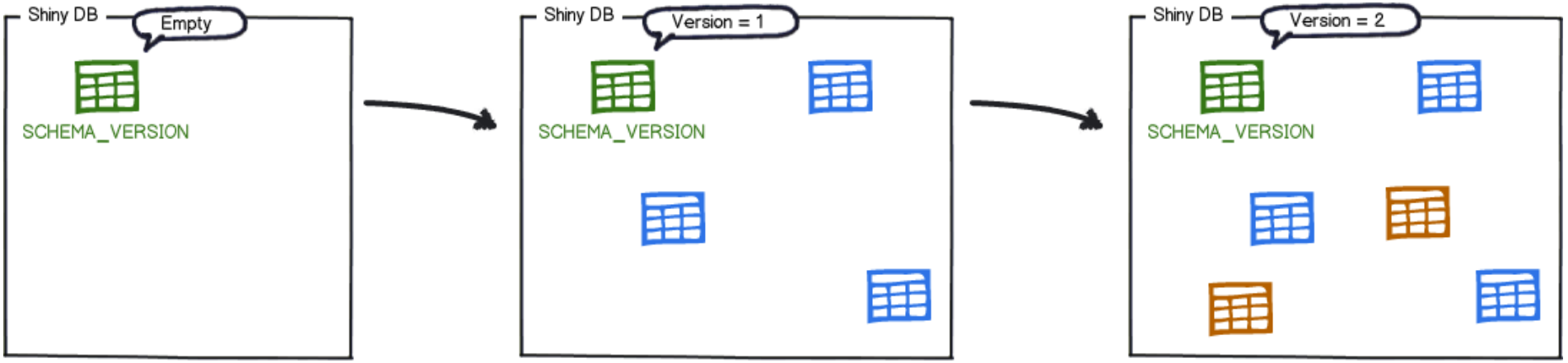
metadata in the database

fail fast

http://dbdeploy.com

http://flywaydb.org/





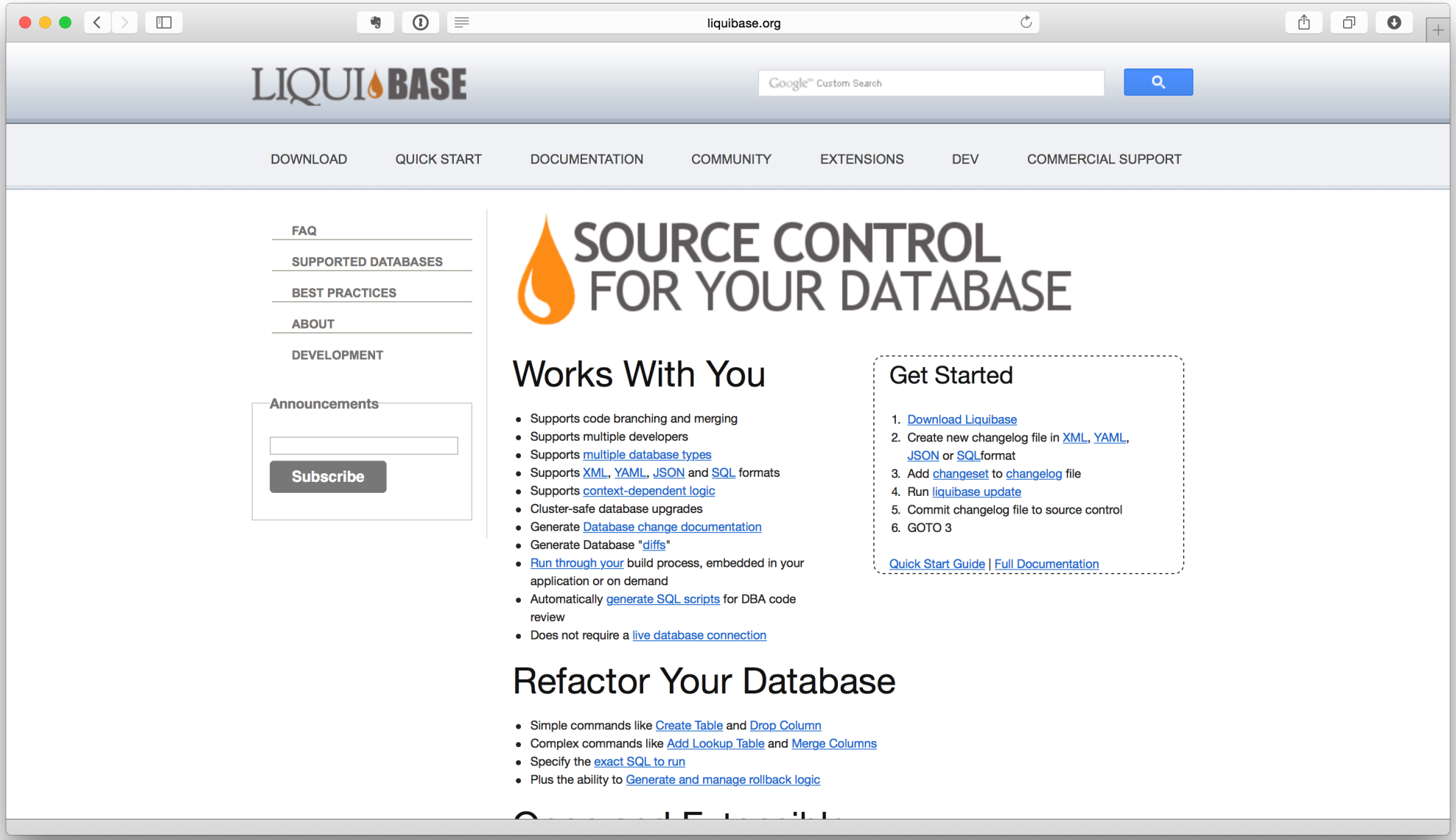
001_create_initial_tables.sql:

```
CREATE TABLE customer (
  id  BIGINT GENERATED BY DEFAULT AS IDENTITY (START WITH 1)
  PRIMARY KEY,
  firstname VARCHAR(255),
  lastname VARCHAR(255)
);
```

002_add_customer_date_of_birth.sql

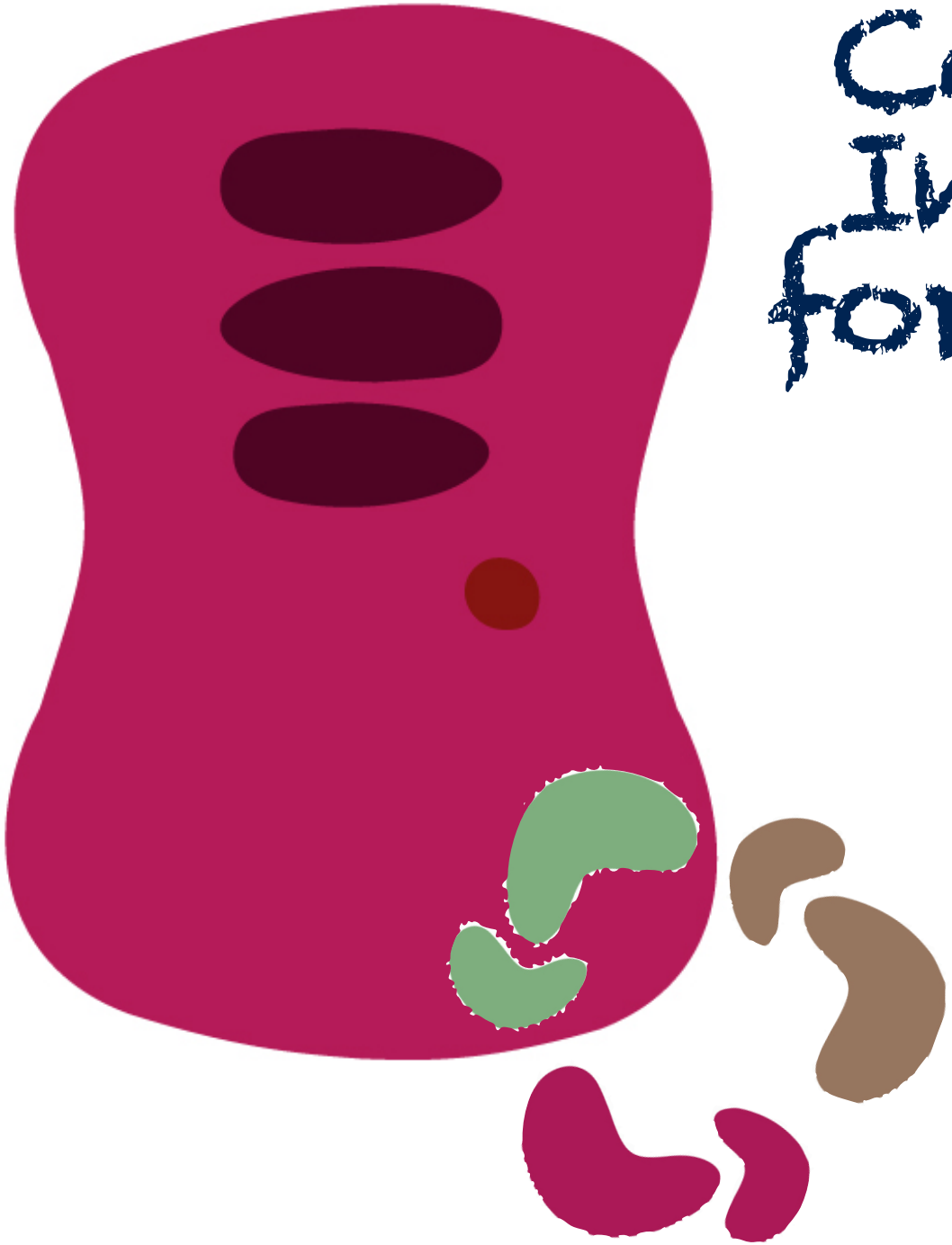
```
ALTER TABLE customer ADD COLUMN dateofbirth DATETIME;

--//@UNDO

ALTER TABLE customer DROP COLUMN dateofbirth;
```

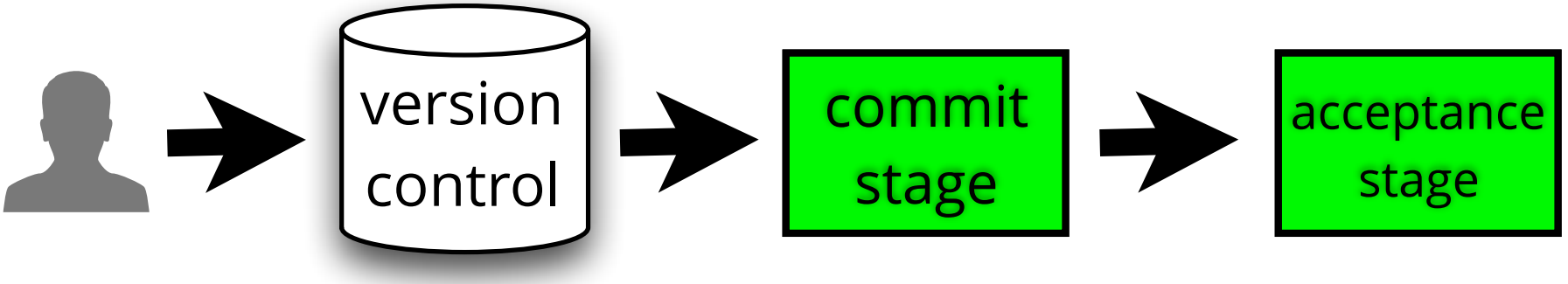
Liquibase

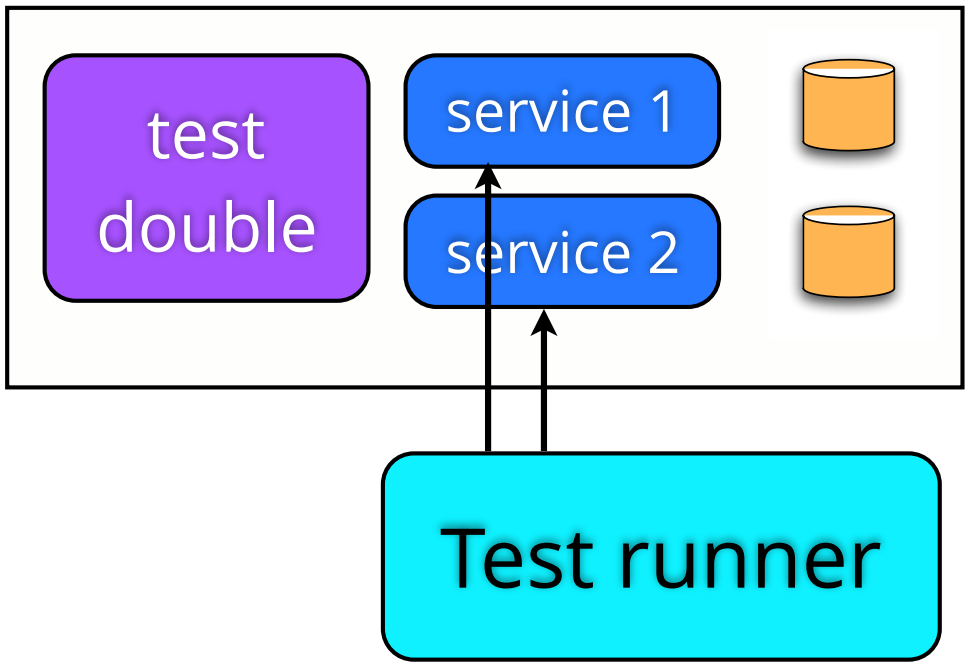


http://www.liquibase.org/

Continuous Integration for Databases







Prepare environment

Deploy app

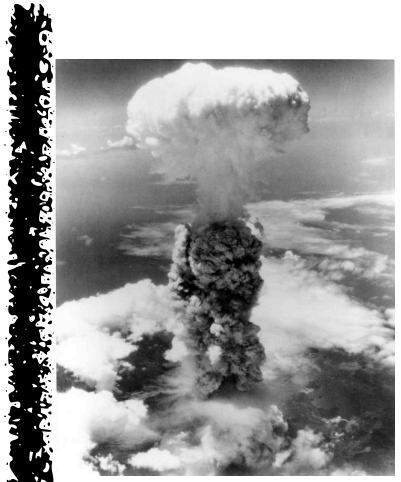
Create dbs, apply schema

Add app reference data

Run acceptance tests

For DB CI We Need To:

start with a clean database



apply changes incrementally

use the same process everywhere

be comprehensive in change management

#1: Baseline

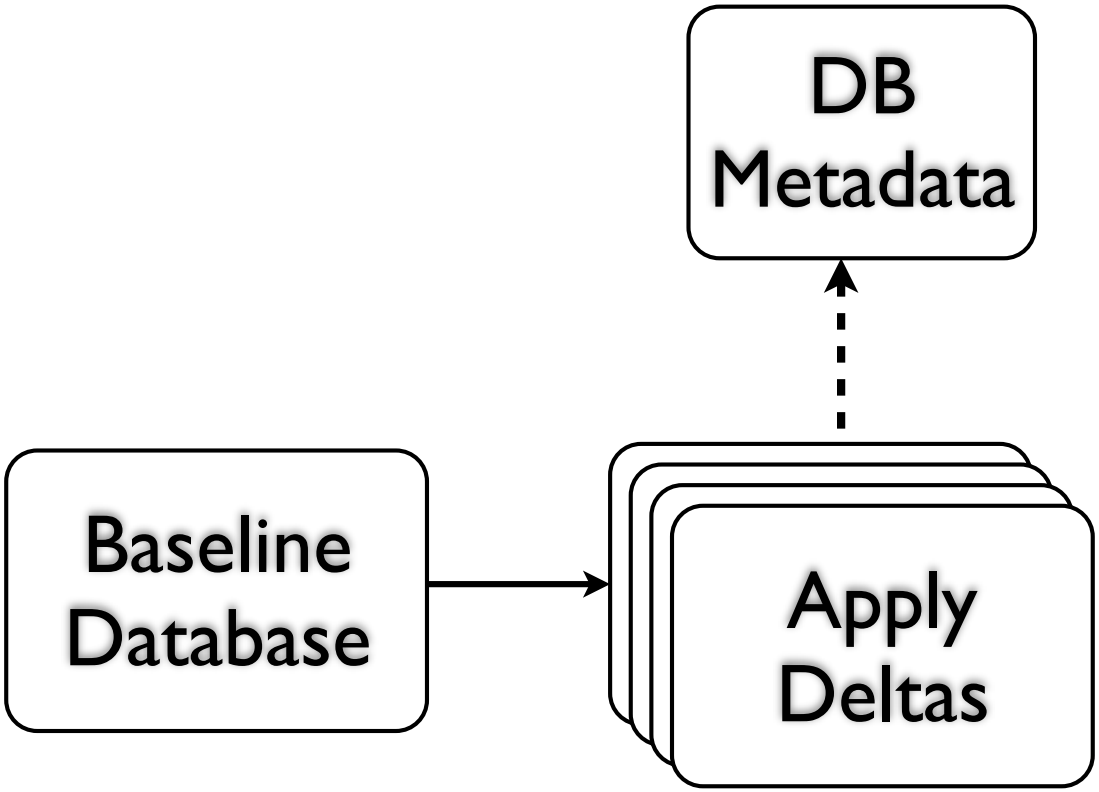
—Create database

—Add metadata table

Baseline Database

—Restore scheme & reference data to current production state

#2: Apply Deltas

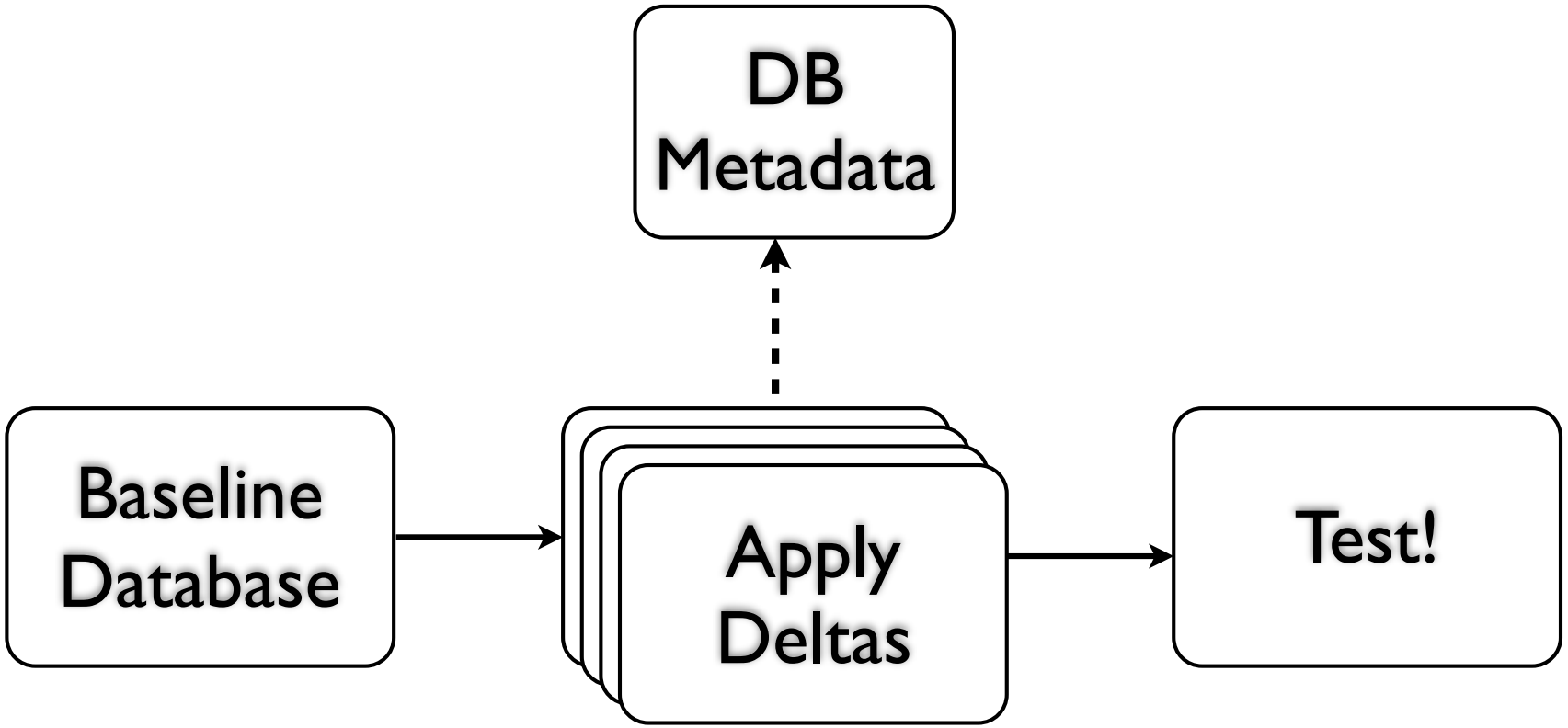


—run each delta in order

—stop the line if one fails

—record success in metadata table

#3: Run Tests



acceptance tests verify database scripts worked

Apply Deltas

run each delta in order

stop the line if one delta fails

auto-rollback if possible

record success in db metadata table

What's a Good Delta

small, self-contained change

ordered (001.sql, 002.sql ...)

deltas are immutable (mostly)

stored together

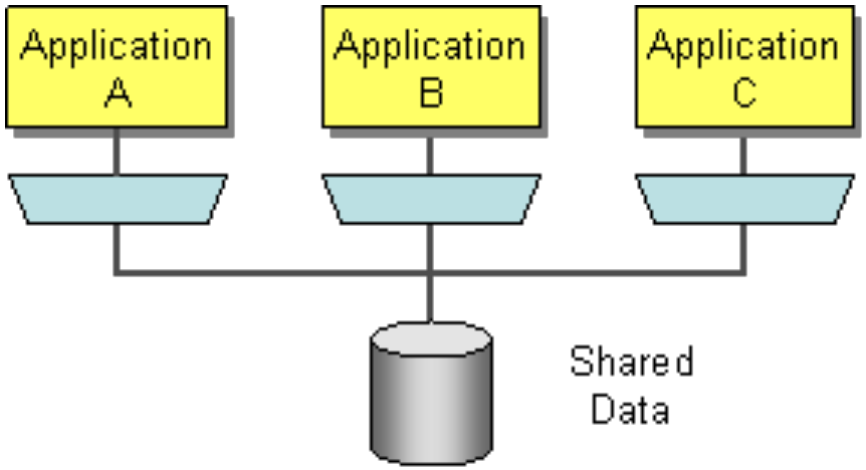
Integration in the DB

share a db delta pool

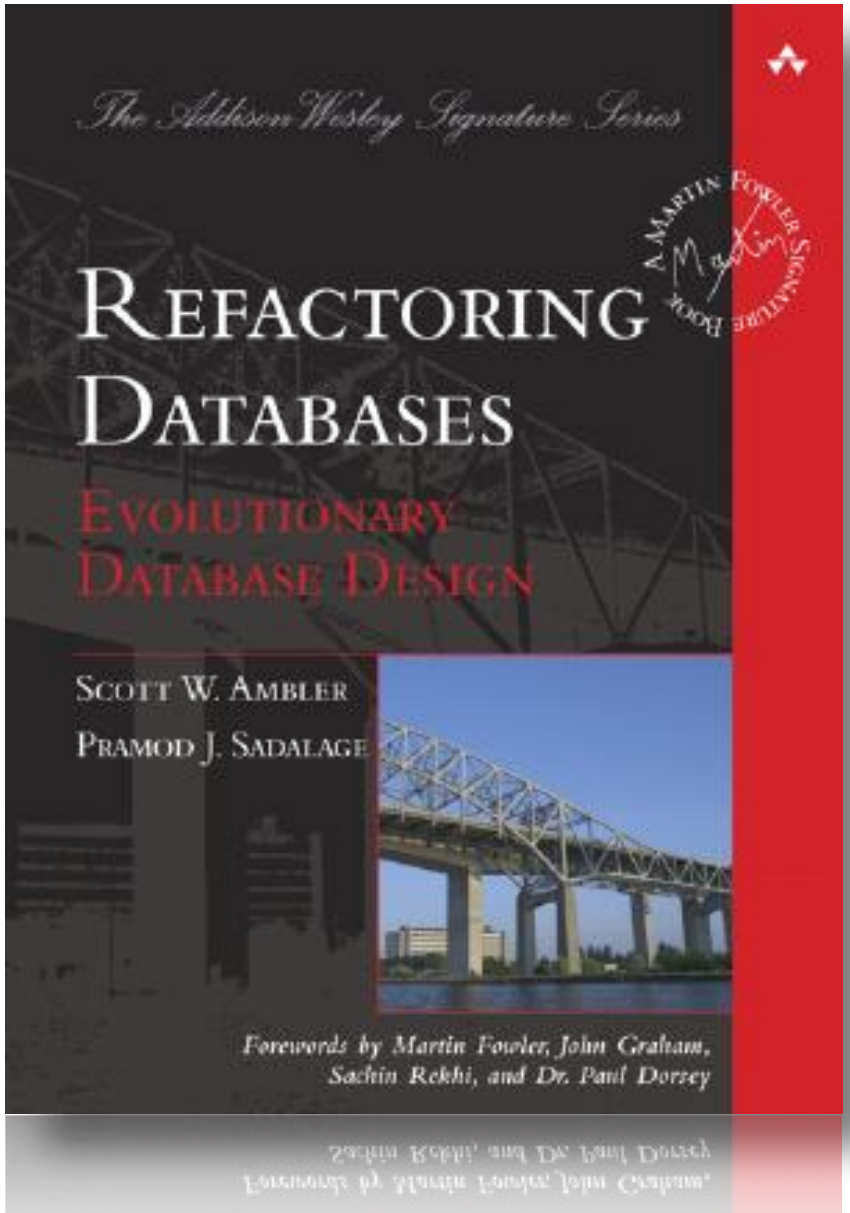
use number ranges

use empty deltas

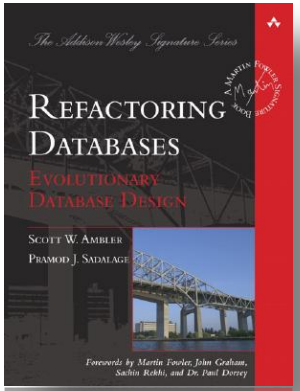
encourage refactoring

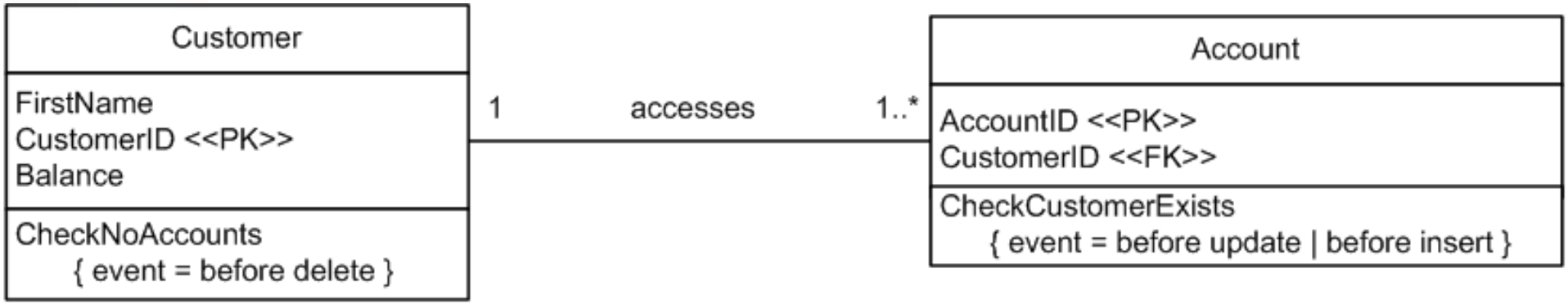


Refactoring Databases



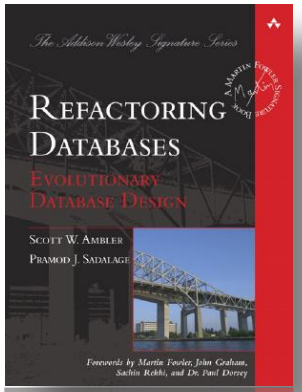
Move Column Refactoring

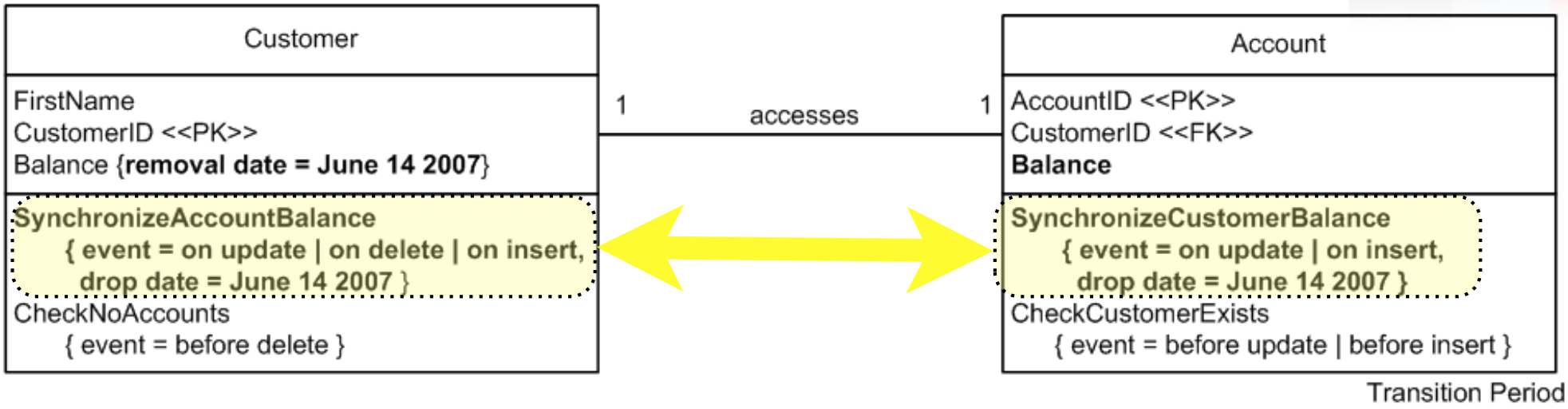




Original Schema

Transition Period

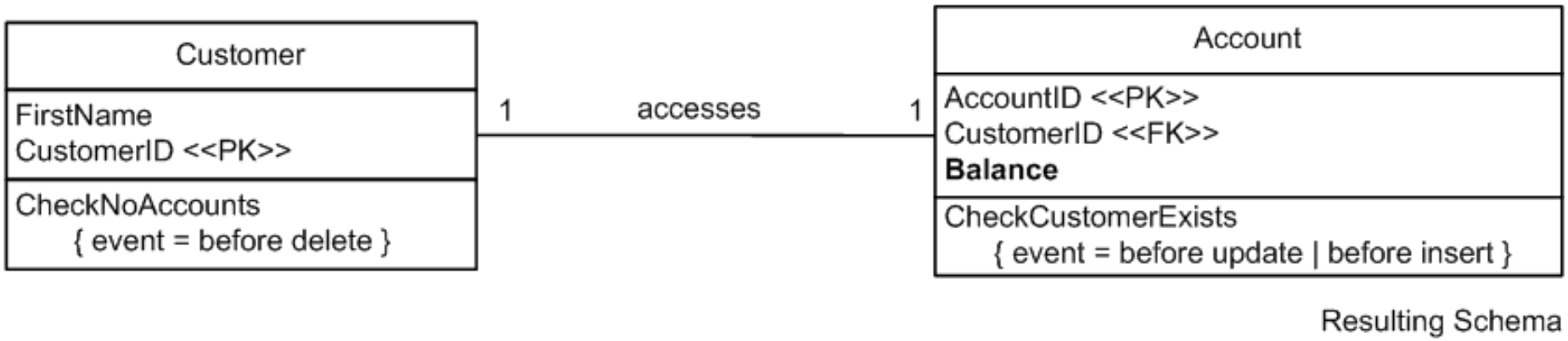




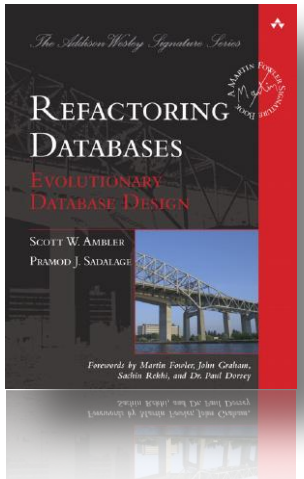
Transition Period

Move Column Refactoring

Ending Schema

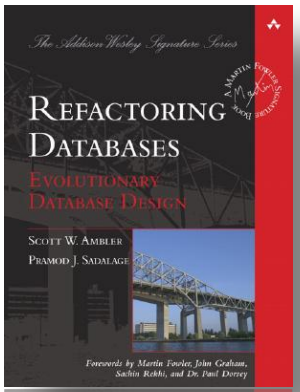


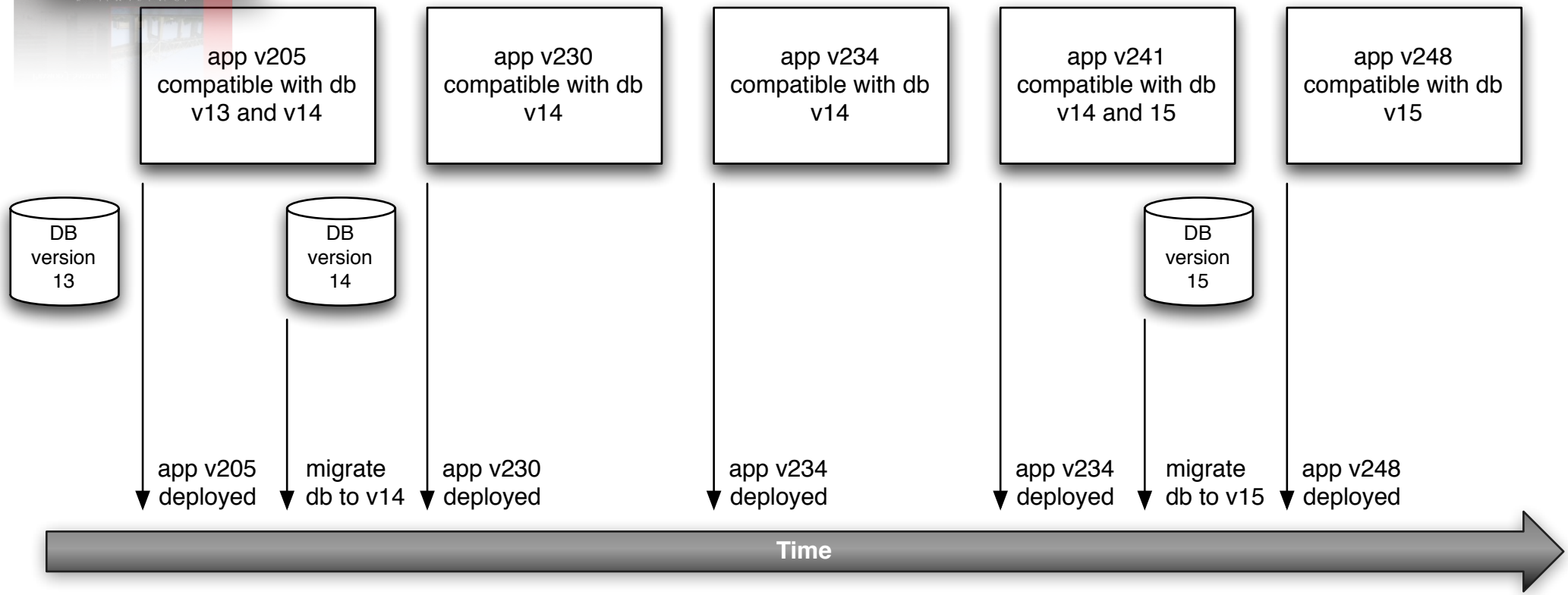
Resulting Schema



Move Column Refactoring

Decouple DB Updates: the Expand/contract Pattern





DB Deployments Still Hard

practice, practice, practice

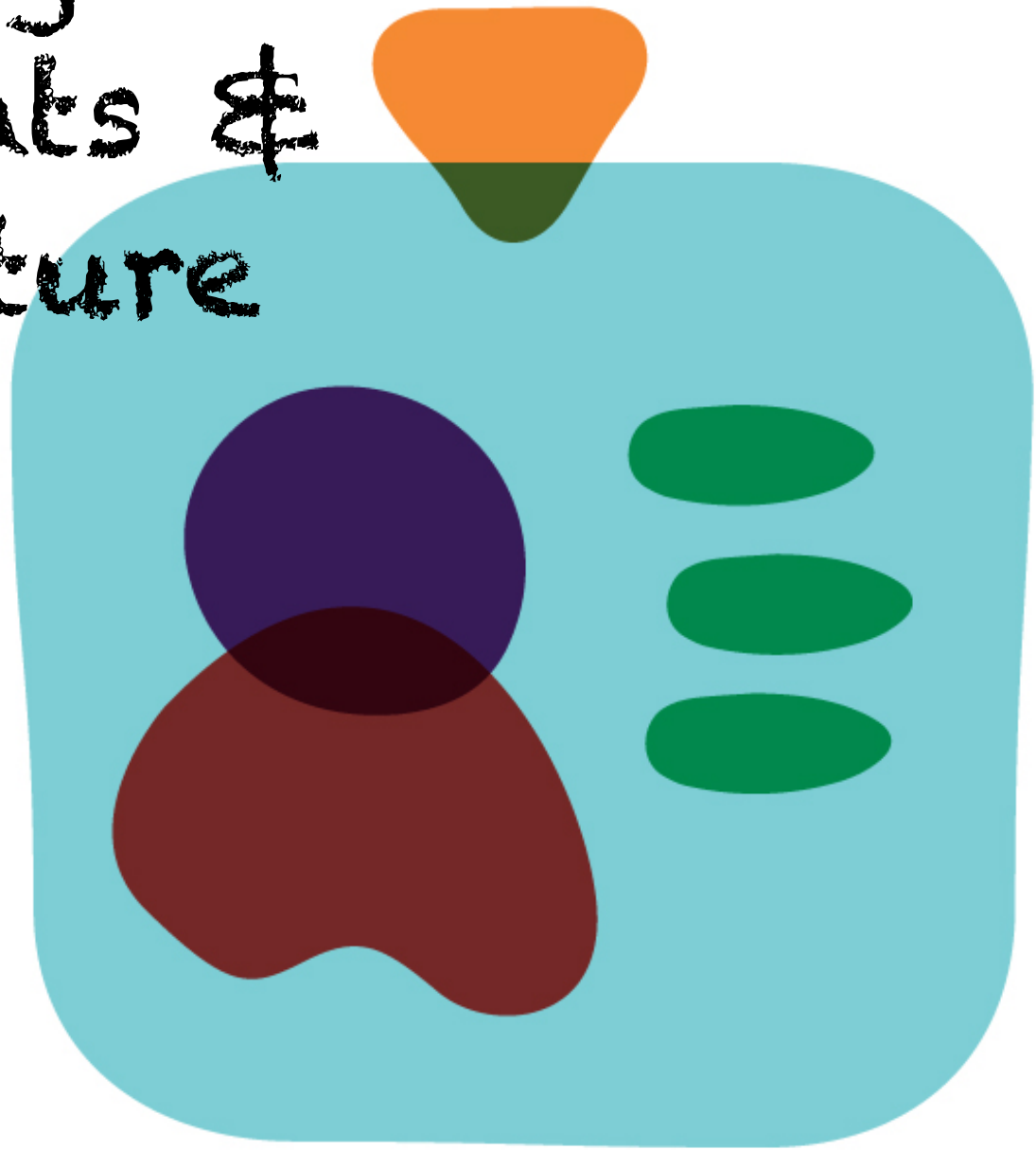
fail fast

bring the pain forward

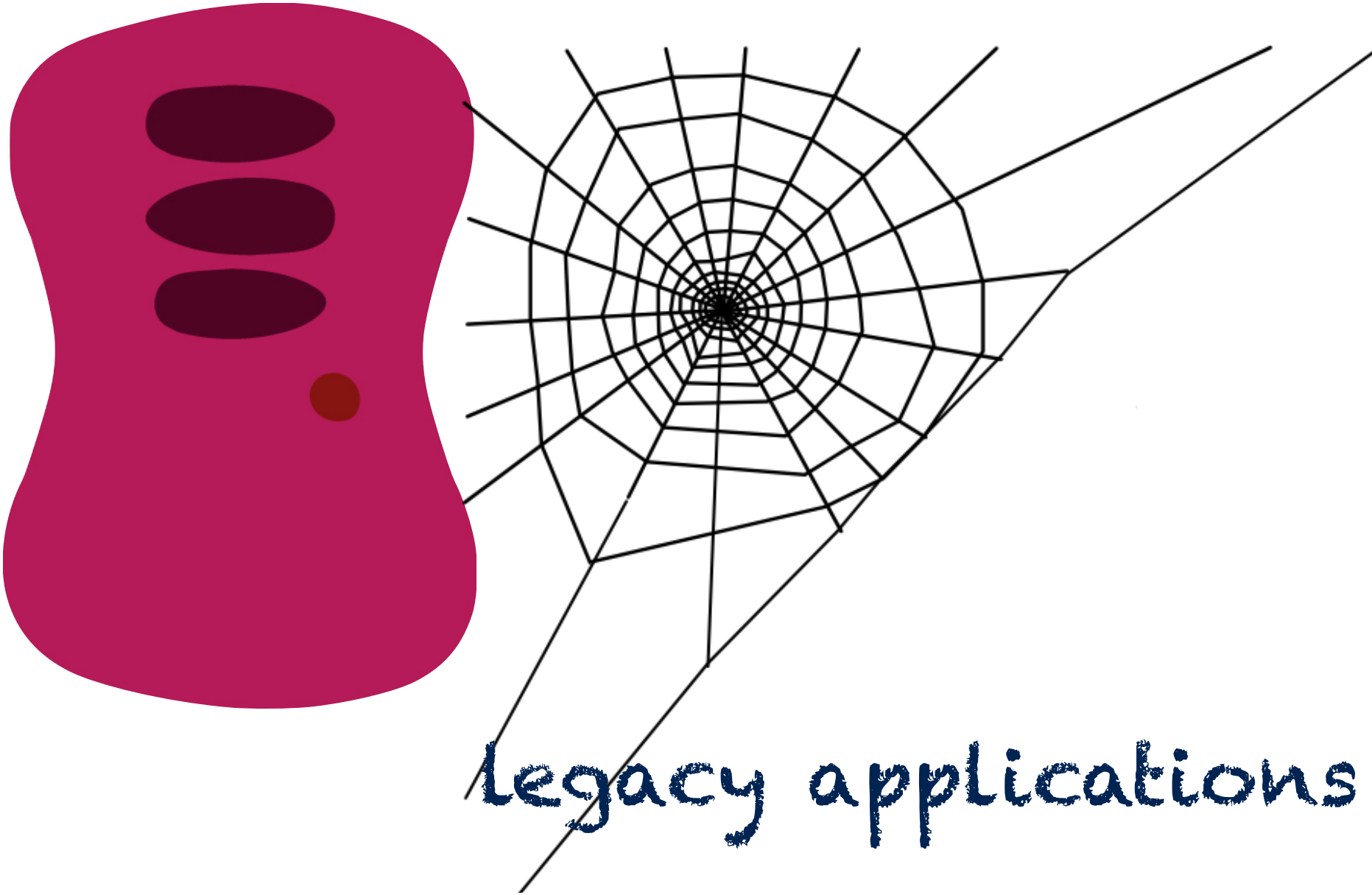
refactor the db

update engineering practices

Managing Environments & Infrastructure

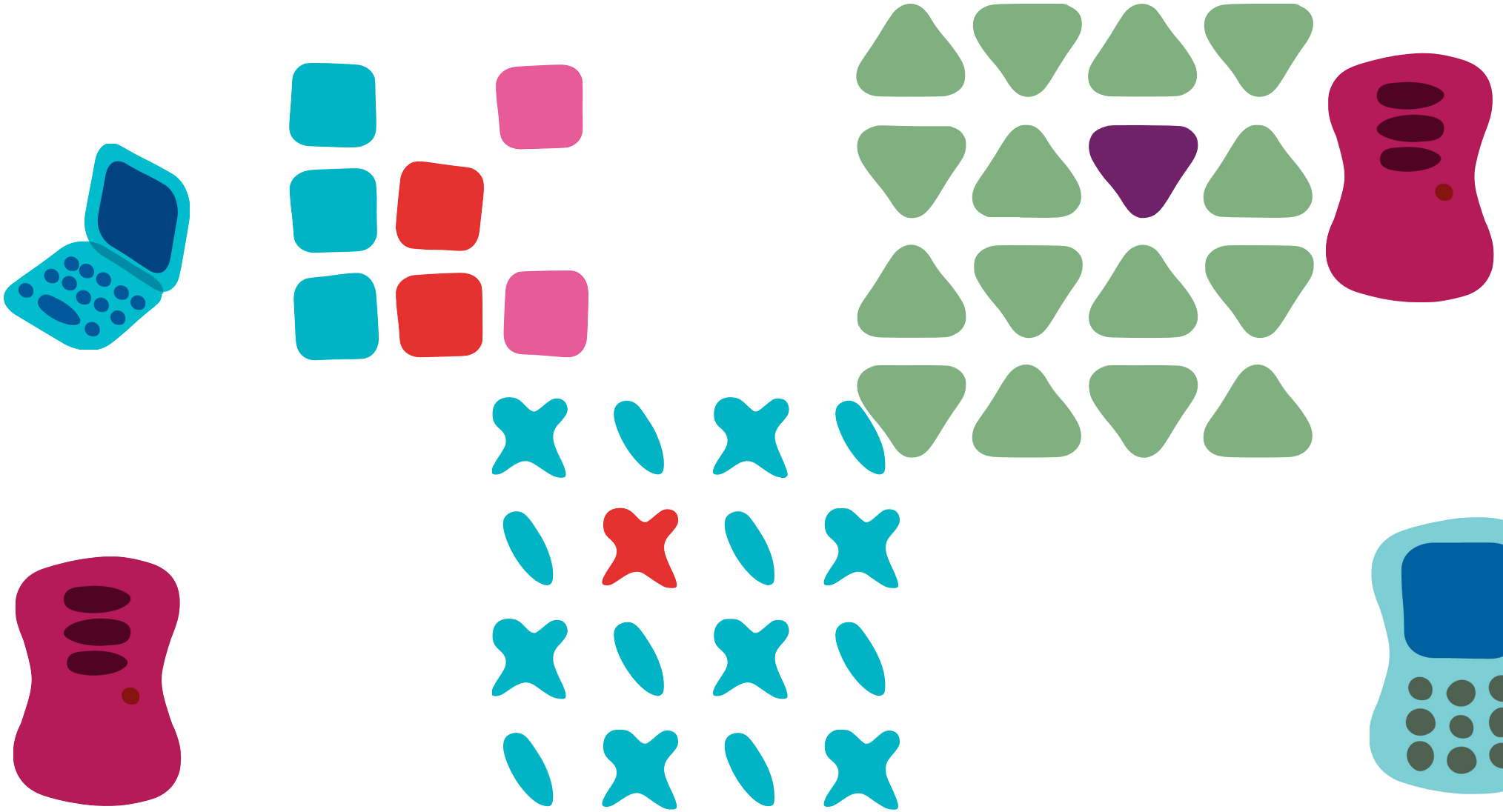


The Pain of Operations

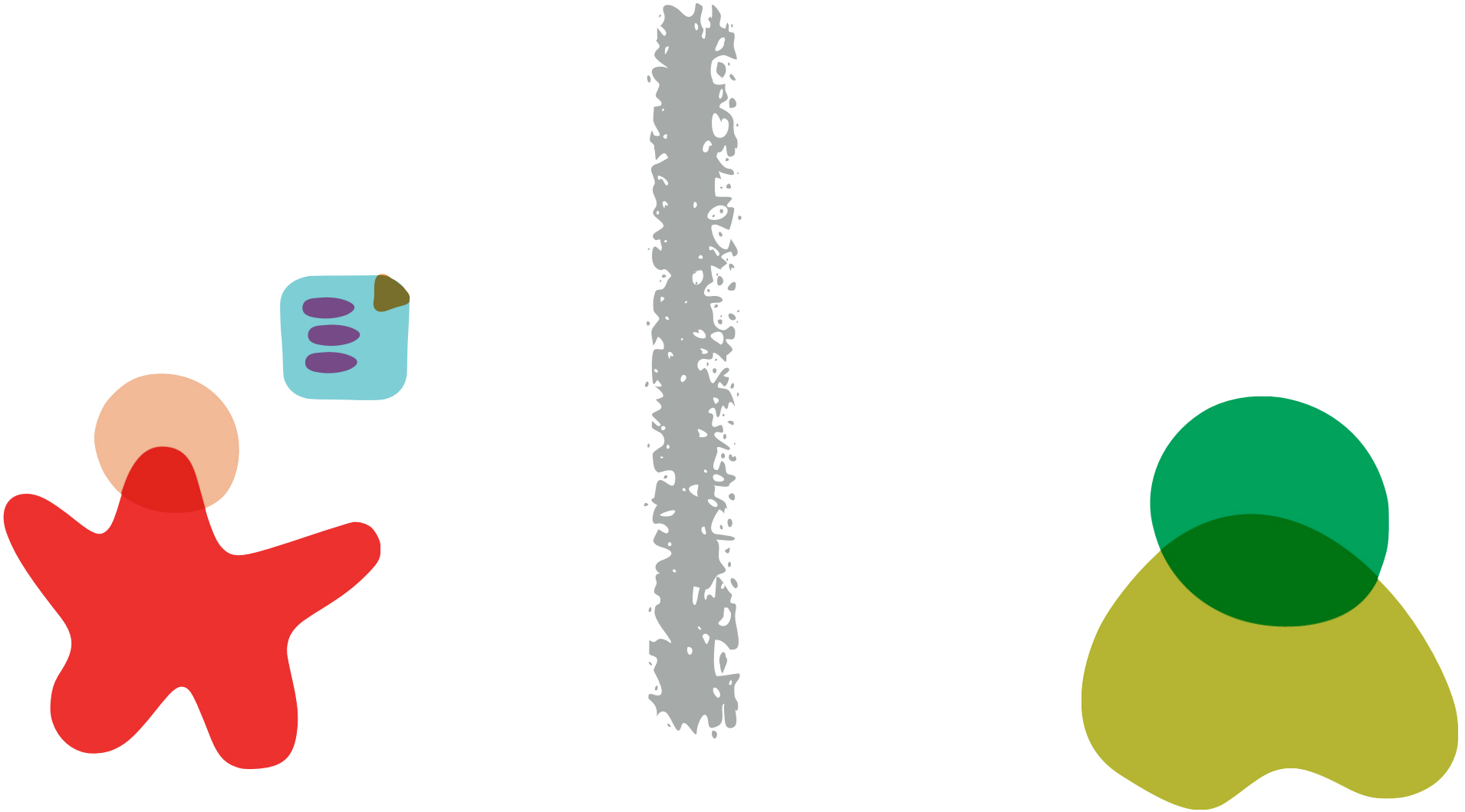


The Pain of Operations

heterogeneous platforms

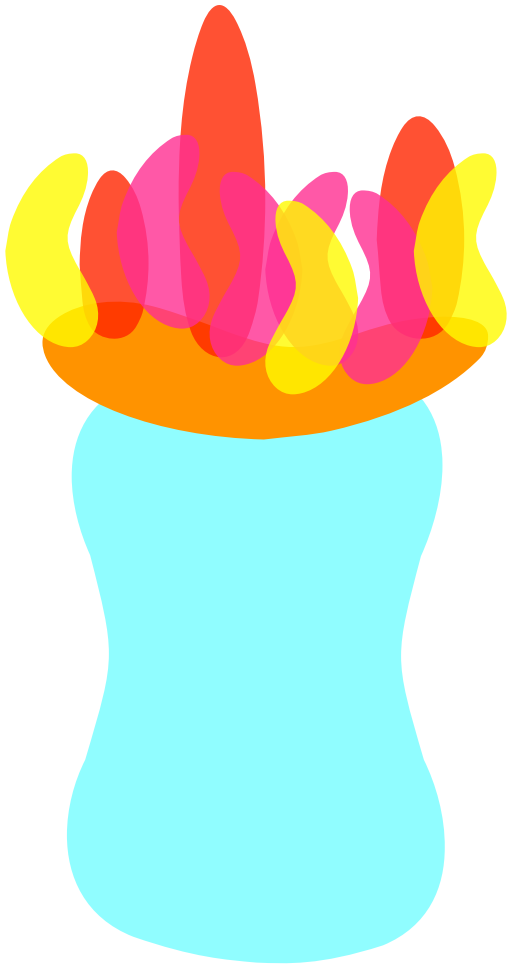


The Pain of Operations



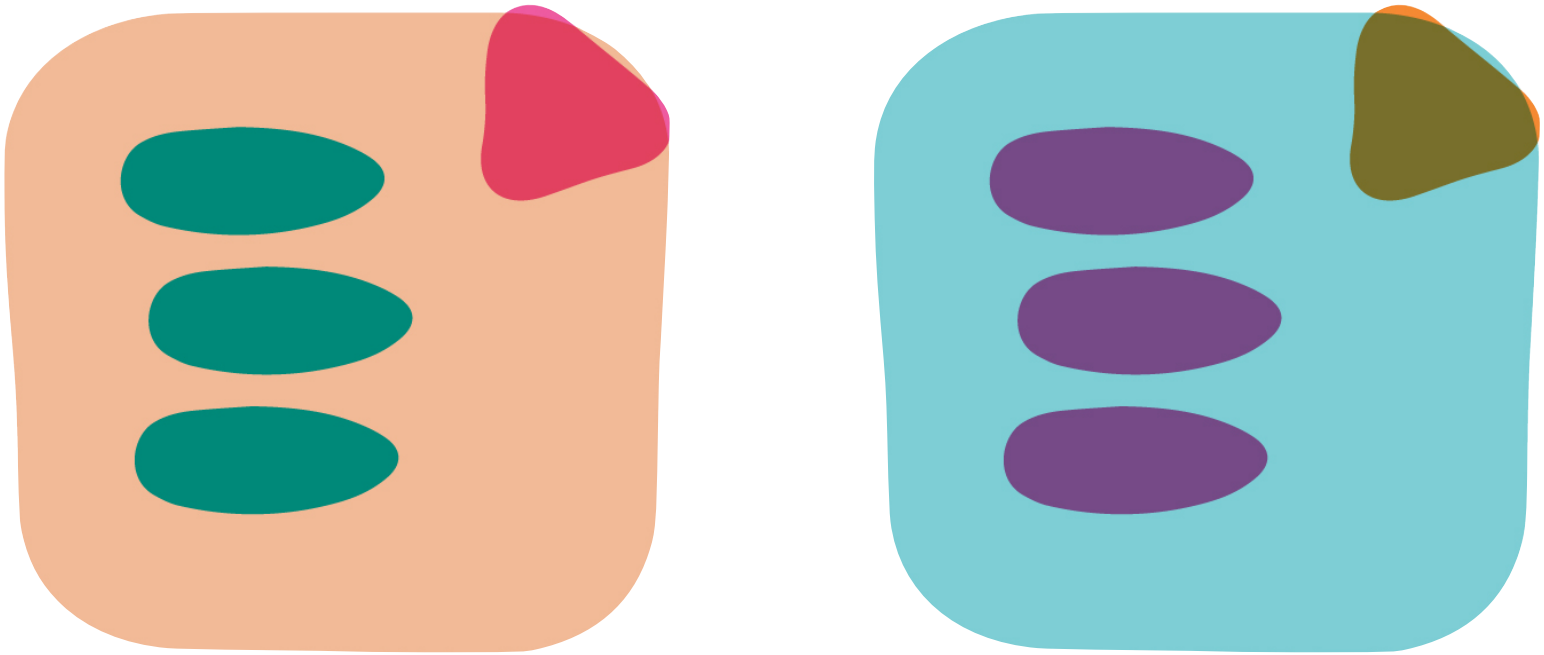
poor quality software thrown over a wall

The Pain of Operations



inordinate amount of firefighting

The Pain of Operations



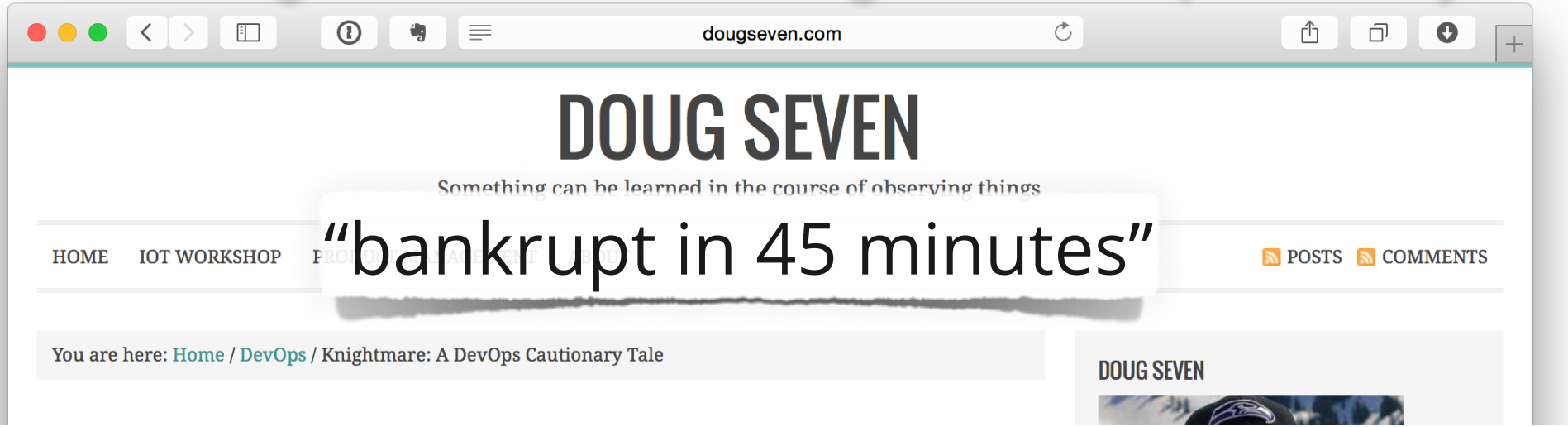
conservative, process heavy

The Pain of Operations

huge budget for operations

Horror Stories

dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/



"bankrupt in 45 minutes"

DevOps

not it’s own silo, but a liaison between operations and developers

at inceptions, showcases, retros

devs work in ops and carry pagers

devs create more deployable software

Managing Infrastructure

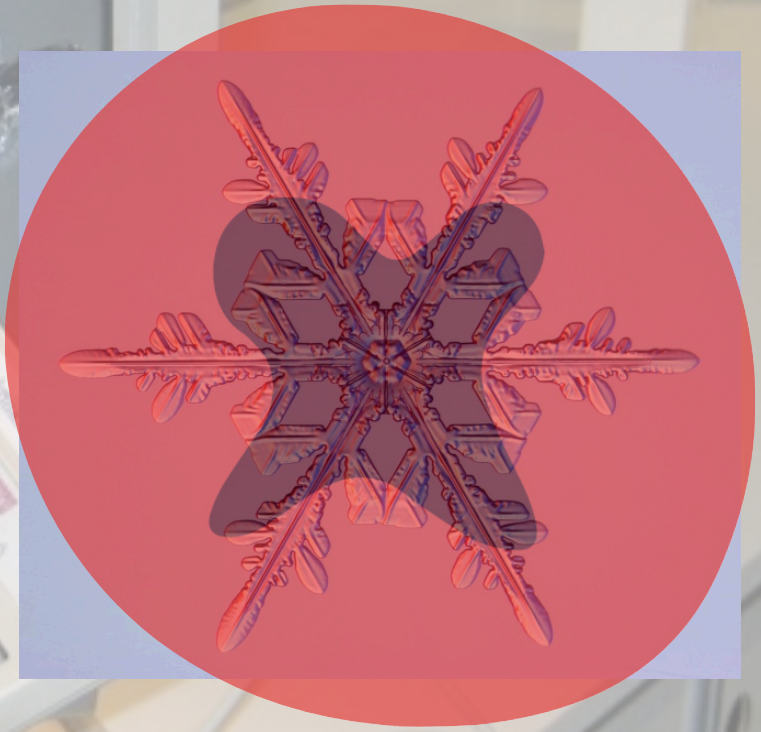
infrastructure = environments and supporting services (networking, vcs, storage, mail, dns...)

desired state specified in version control

autonomic (self-corrects to desired state)

state should be known through monitoring

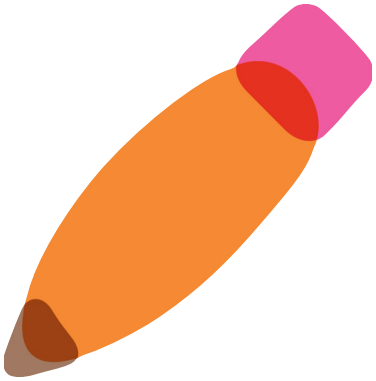
Destroy Works of Art



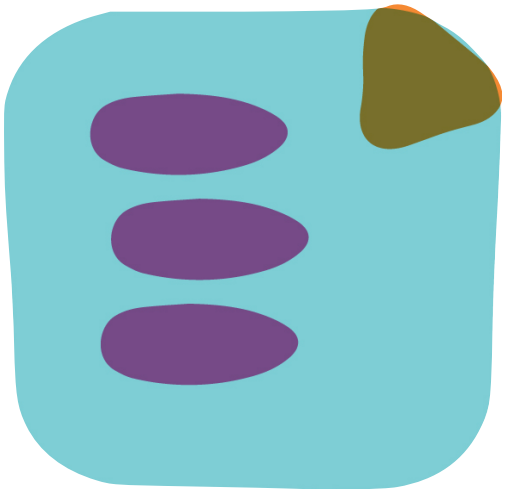
If someone threw a server out of the window, how long would it take to recreate it?

Infrastructure as Code

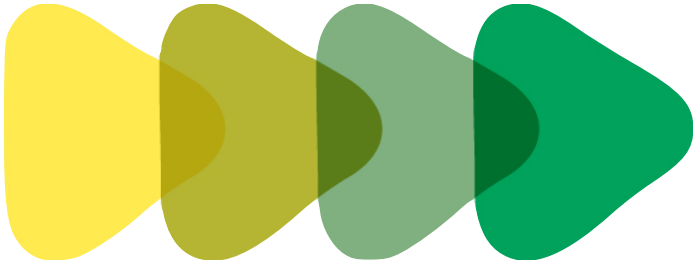
refactoring



testing

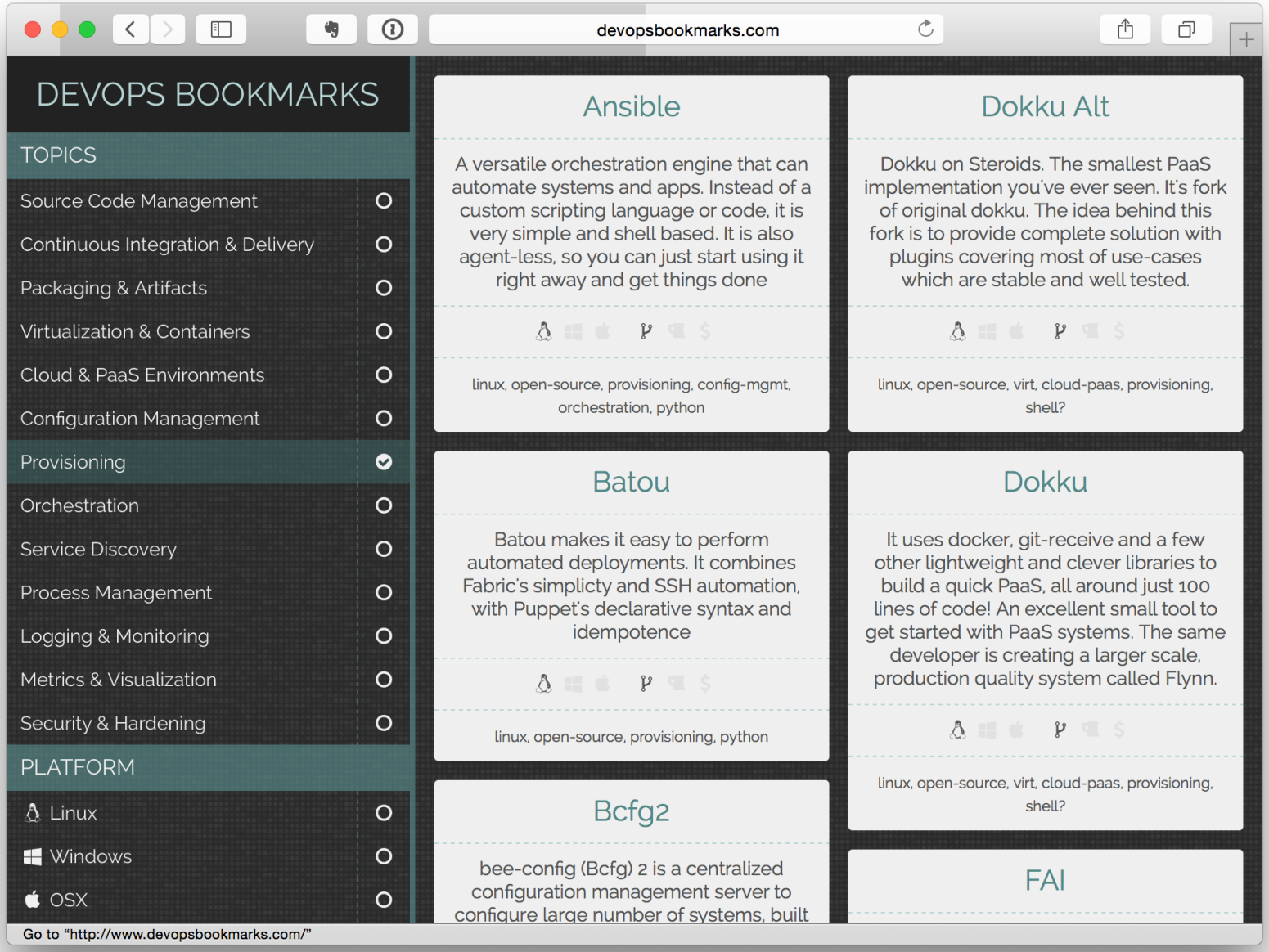


self documenting



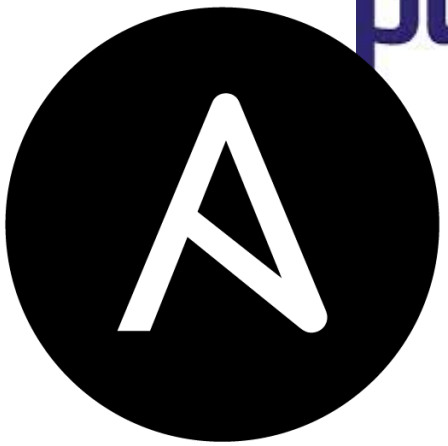
build pipeline

Tools



www.devopsbookmarks.com/

Tools



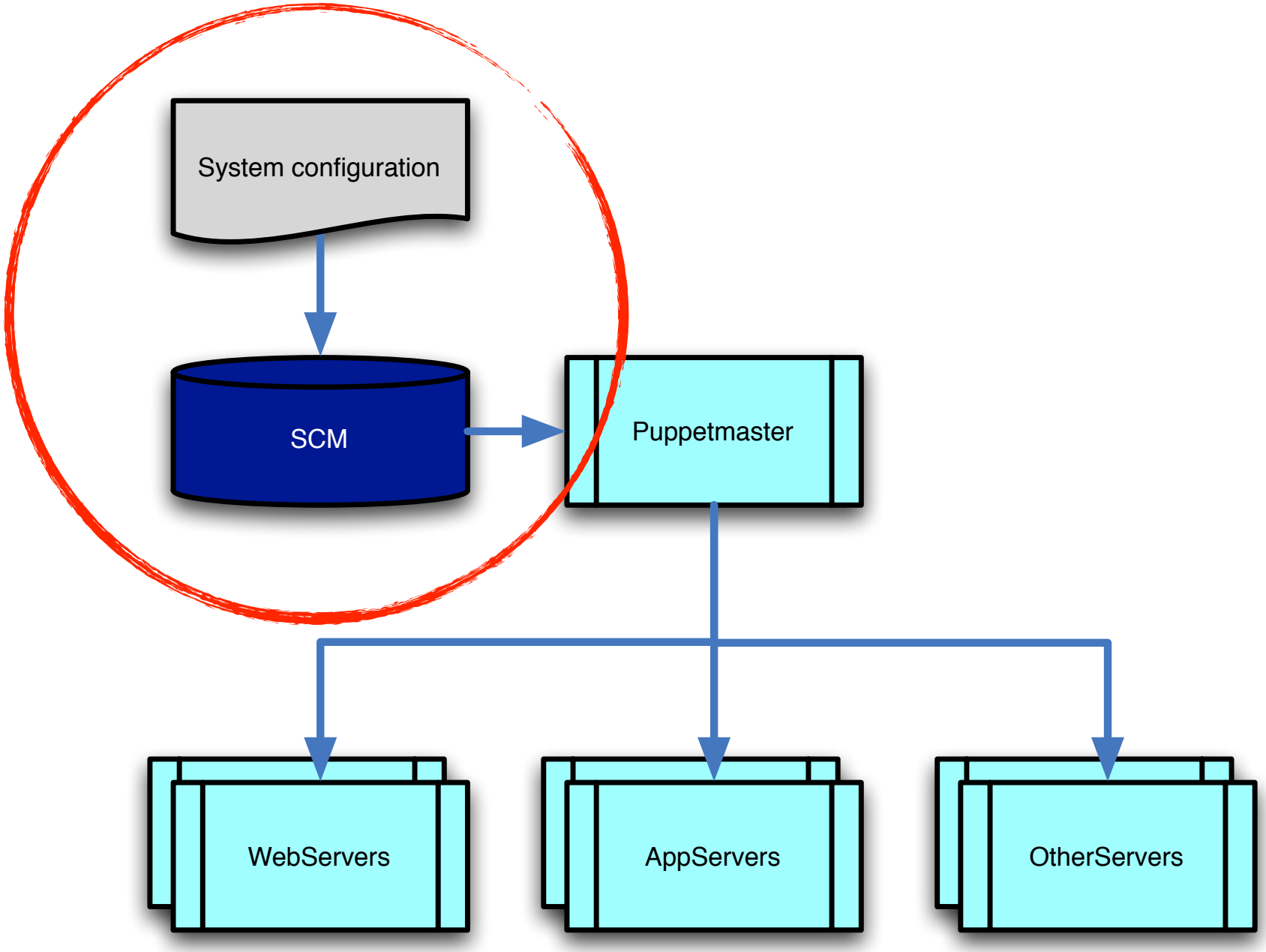
ANSIBLE

manage many systems

manage configuration

enforce consistency

treat infrastructure as code


System configuration
SCM
Puppetmaster
WebServers
AppServers
OtherServers

example (HTTP)


```
class httpd {
  package { httpd:
    ensure => latest
  }

  configfile { "/etc/httpd/conf/httpd.conf":
    source => "/httpd/httpd.conf",
    mode => 644,
    require => package["httpd"]
  }

  group { apache: gid => 48 }

  user { apache:
   comment => "Apache",
   uid => 48,
   gid => 48,
   home => "/var/www",
   shell => "/sbin/nologin"
  }

  service { httpd:
    running => true,
    subscribe => [ file["/etc/httpd/conf/httpd.conf"],
                   package["httpd"] ]
  }
}
```


```
class tomcat {

  $tomcat_port = 735
  $tomcat_password = 'badwolf'

  notice("Establishing http://$hostname:$tomcat_port/")

  Package { # defaults
    ensure => installed,
  }

  package { 'tomcat6':
  }

  package { 'tomcat6-user':
    require => Package['tomcat6'],
  }

  package { 'tomcat6-admin':
    require => Package['tomcat6'],
  }

  file { "/etc/tomcat6/tomcat-users.xml":
    owner => 'root',
    require => Package['tomcat6'],
    notify => Service['tomcat6'],
    content => template('tomcat/tomcat-users.xml.erb')
  }

  file { '/etc/tomcat6/server.xml':
     owner => 'root',
     require => Package['tomcat6'],
     notify => Service['tomcat6'],
     content => template('tomcat/server.xml.erb'),
  }

  service { 'tomcat6':
    ensure => running,
    require => Package['tomcat6'],
  }

}
```

```
define tomcat::deployment($path) {

  include tomcat
  notice("Establishing http://$hostname:${tomcat::tomcat_port}/$name/")

  file { "/var/lib/tomcat6/webapps/${name}.war":
    owner => 'root',
    source => $path,
  }

}
```

Unified Deployment

deploy the same way to all environments

separate the things that change from the things that don’t

flexible environment targeting

Virtualization

great for creating production-like test envs, highly parallel testing

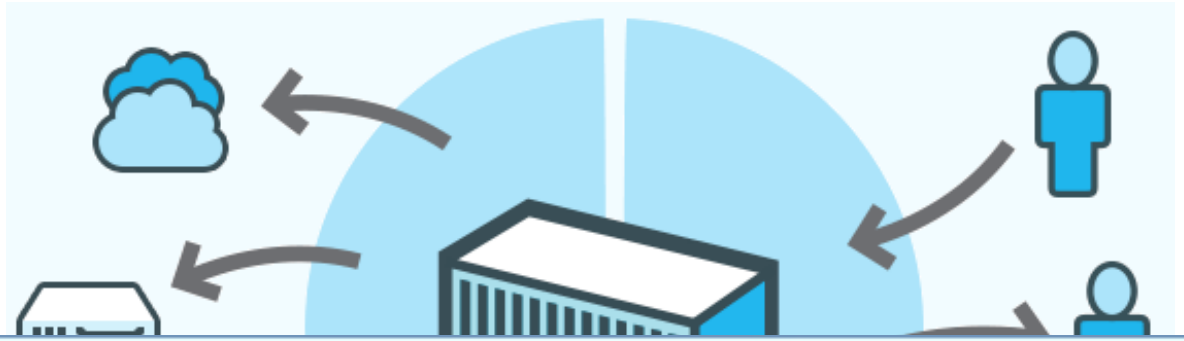
cloud is great for utility computing and scaling on demand

most real systems will be heterogeneous

use virtualization with provisioning tools

docker

www.docker.com/



Docker is a shipping container system for code

Multiplicity of Stacks

Static website | User DB | Web frontend | Queue | Analytics DB

Do services and apps interact appropriately?

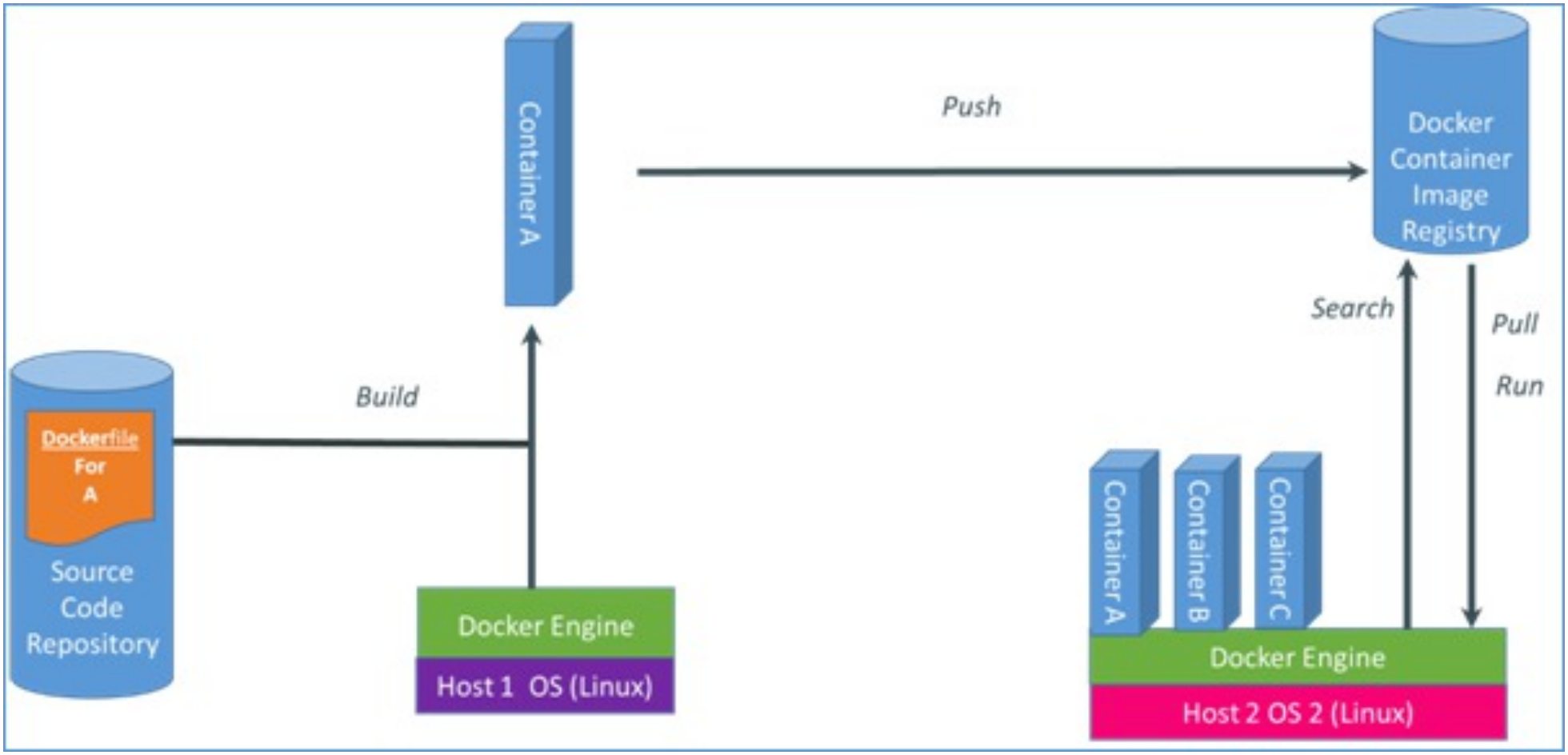
An engine that enables any payload to be encapsulated as a lightweight, portable, self-sufficient container…

…that can be manipulated using standard operations and run consistently on virtually any hardware platform

Multiplicity of hardware environments

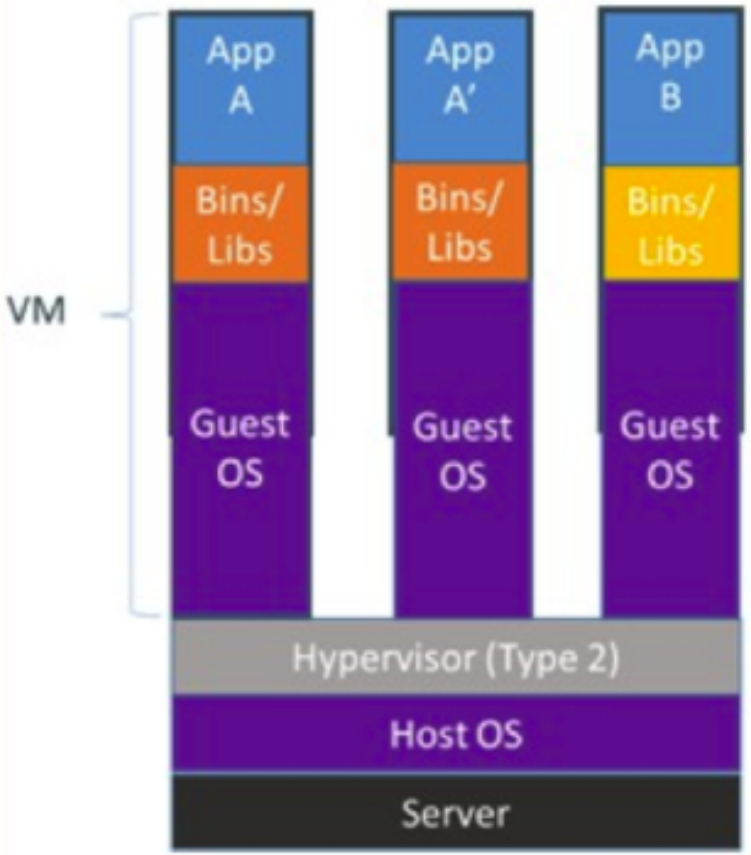
Can I migrate smoothly and quickly

Development VM | QA server | Customer Data Center | Public Cloud | Production Cluster | Contributor's laptop

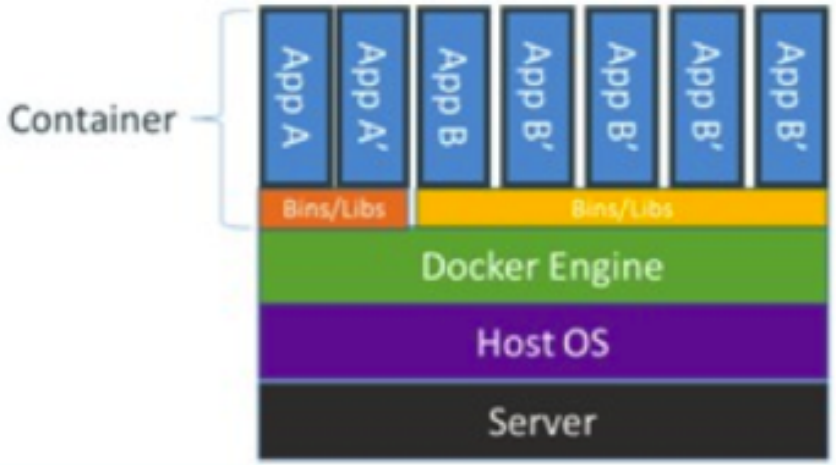

Container A
Push
Docker Container Image Registry
Search
Pull
Run
Build
Dockerfile For A
Source Code Repository
Docker Engine
Host 1 OS (Linux)
Container A
Container B
Container C
Docker Engine
Host 2 OS 2 (Linux)

docker

Containers vs. VMs



Containers are isolated, but share OS and, where appropriate, bins/libraries

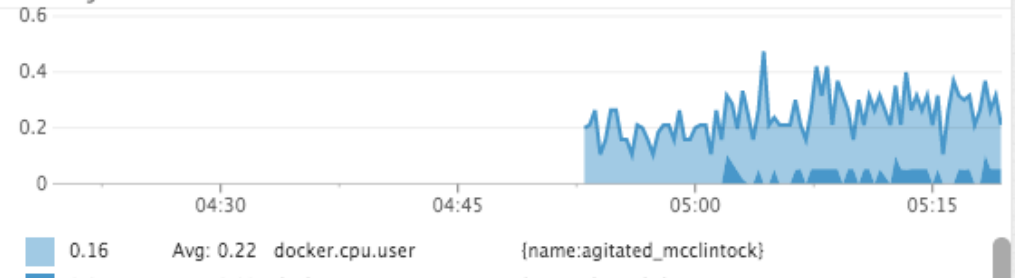


Monitor Docker with Datadog

CPU by image

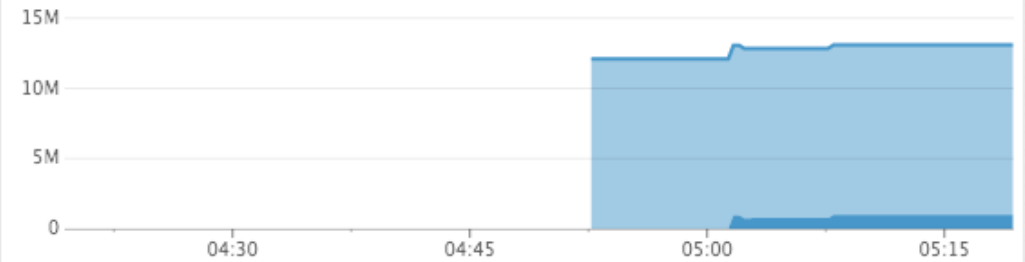
0.16	Avg: 0.22	docker.cpu.user	{image:datadog/docker-dogstatsd:latest}
0.05	Avg: 0.02	docker.cpu.user	{image:redis:2.8}

CPU by container



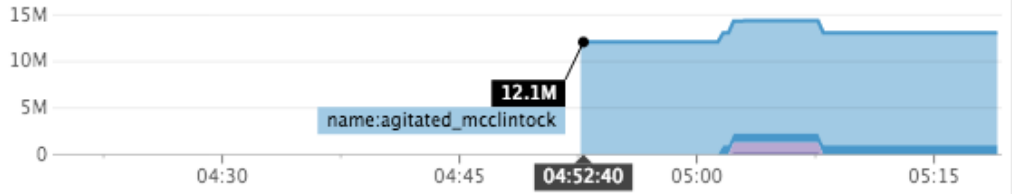
0.16	Avg: 0.22	docker.cpu.user	{name:agitated_mcclintock}
0.05	Avg: 0.02	docker.cpu.user	{name:alq-redis}
N/A	Avg: 0	docker.cpu.user	{name:alq-redis,name:determined_ptolemy/re...
N/A	Avg: 0	docker.cpu.user	{name:determined_ptolemy}

RSS by image

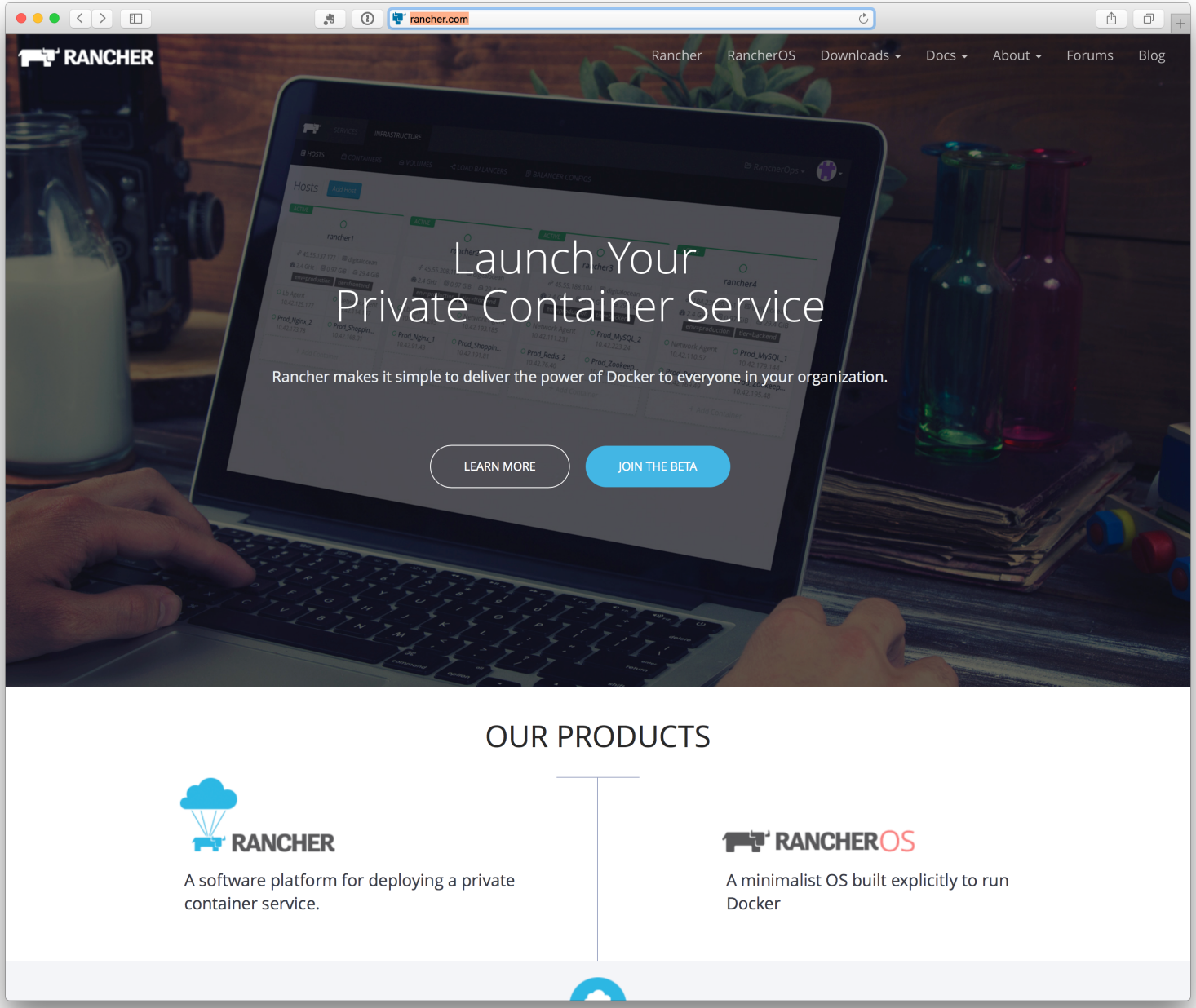


12.1M	Avg: 12....	docker.mem.rss	{image:datadog/docker-dogstatsd:latest}
1M	Avg: 0.6...	docker.mem.rss	{image:redis:2.8}

RSS by image



N/A	Avg: 12....	docker.mem.rss	{name:agitated_mcclintock}
N/A	Avg: 0.6...	docker.mem.rss	{name:alq-redis}
N/A	Avg: 0.2...	docker.mem.rss	{name:alq-redis,name:determined_ptolemy/re...
N/A	Avg: 0.0...	docker.mem.rss	{name:determined_ptolemy}


rancher.com
RANCHER
Rancher
RancherOS
Downloads
Docs
About
Forums
Blog
Launch Your
Private Container Service
Rancher makes it simple to deliver the power of Docker to everyone in your organization.
LEARN MORE
JOIN THE BETA
OUR PRODUCTS
RANCHER
A software platform for deploying a private container service.
RANCHEROS
A minimalist OS built explicitly to run Docker

Built for Robustness


techblog.netflix.com

Mer Neste blogg» Opprett blogg Logg p

NETFLIX

The Netflix Tech Blog

Tuesday, July 19, 2011

The Netflix Simian Army

We've talked a bit in the past about our move to the cloud and John shared some of our lessons learned in going through that transition in a previous post. Recently, we've been focusing on ways to improve availability and reliability and wanted to share some of our progress and thinking.

The cloud is all about redundancy and fault-tolerance. Since no single component can guarantee 100% uptime (and even the most expensive hardware eventually fails), we have to design a cloud architecture where individual components can fail without affecting the availability of the entire system. In effect, we have to be stronger than our weakest link. We can use techniques like graceful degradation on dependency failures, as well as node-, rack-, datacenter/availability-zone and even regionally-redundant deployments. But just designing a fault tolerant architecture is not enough. We have to constantly test our ability to actually survive these "once in a blue moon" failures.

Imagine getting a flat tire. Even if you have a spare tire in your trunk, do you know if it is inflated? Do you have the tools to change it? And, most importantly, do you remember how to do it right? One way to make sure you can deal with a flat tire on the freeway, in the rain, in the middle of the night is to poke a hole in your tire once a week in your driveway on a Sunday afternoon and go through the drill of replacing it. This is expensive and time-consuming in the real world, but can be (almost) free and automated in the cloud.

This was our philosophy when we built **Chaos Monkey**, a tool that randomly disables our production instances to make sure we can survive this common type of failure without any customer impact. The name comes from the idea of unleashing a wild monkey with a weapon in your data center (or cloud region) to randomly shoot down instances and chew through cables -- all the while we continue serving our customers without interruption. By running Chaos Monkey in the middle of a business day, in a carefully monitored environment with engineers standing by to address any problems, we can still learn the lessons about the weaknesses of our system, and build automatic recovery mechanisms to deal with them. So next time an instance fails at 3 am on a Sunday, we won't even notice.

Inspired by the success of the Chaos Monkey, we've started creating new simians that induce various kinds of failures, or detect abnormal conditions, and test our ability to survive them; a virtual Simian Army to keep our cloud safe, secure, and highly available.

Latency Monkey induces artificial delays in our RESTful client-server communication layer to simulate service degradation and measures if upstream services respond appropriately. In addition, by making very large delays, we can simulate a node or even an entire service downtime (and test our ability to survive it) without physically bringing these instances down. This can be particularly useful when testing the fault-

Links

- Netflix US & Canada Blog
- Netflix America Latina Blog
- Netflix Brasil Blog
- Netflix Benelux Blog
- Netflix DACH Blog
- Netflix France Blog
- Netflix Nordics Blog
- Netflix UK & Ireland Blog
- Netflix ISP Speed Index
- Open positions at Netflix
- Netflix Website
- Facebook Netflix Page
- Netflix UI Engineering
- RSS Feed

About the Netflix Tech Blog

This is a Netflix blog focused on technology and technology issues. We'll share our perspectives, decisions and challenges regarding the software we build and use to create the Netflix service.

Blog Archive

Latency Monkey

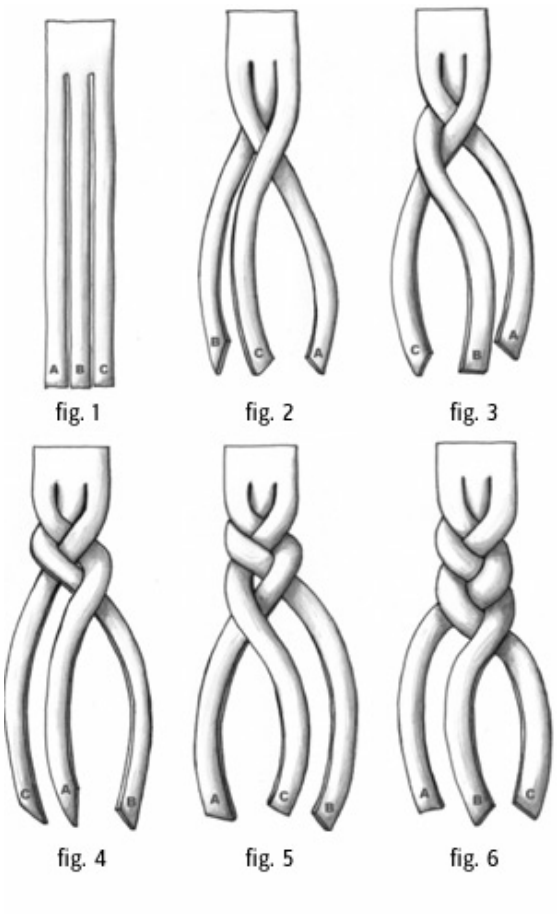
Conformity Monkey

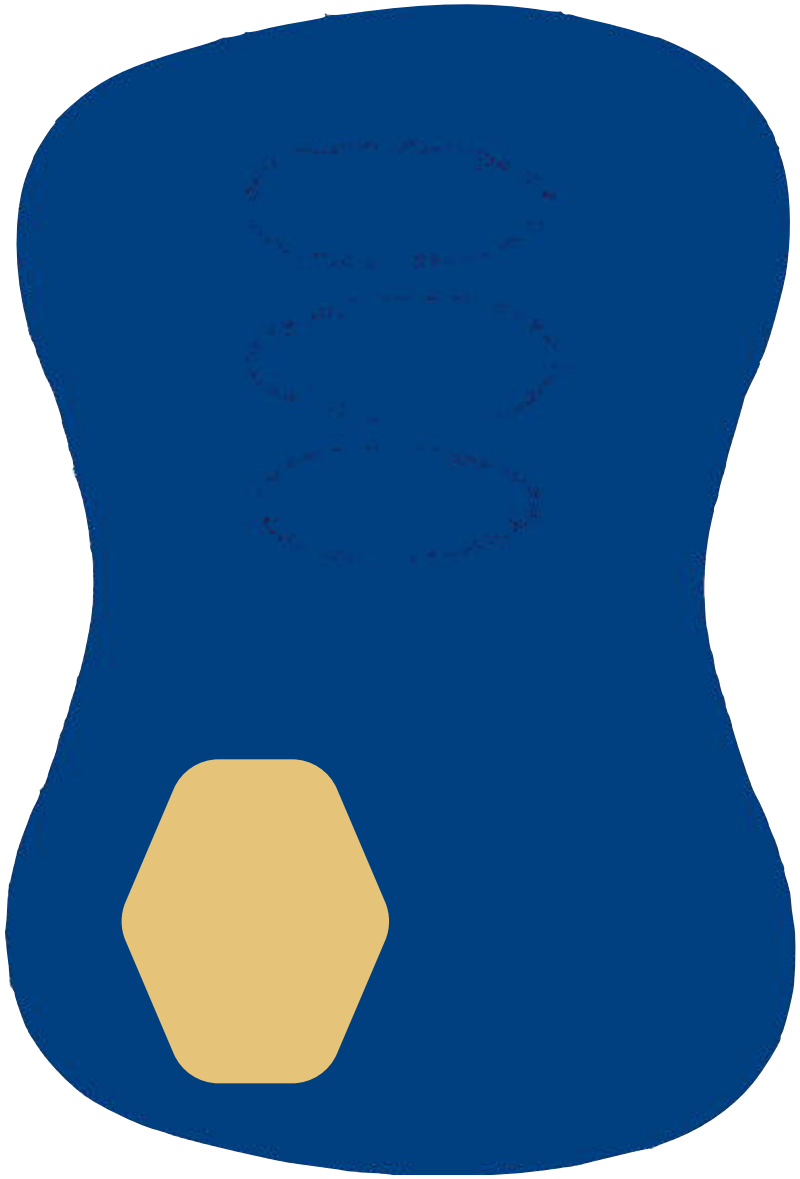
Doctor Monkey

Janitor Monkey

Chaos Gorilla

Complected Deployments





complect, *transitive verb*:
intertwine, embrace, especially
to plait together

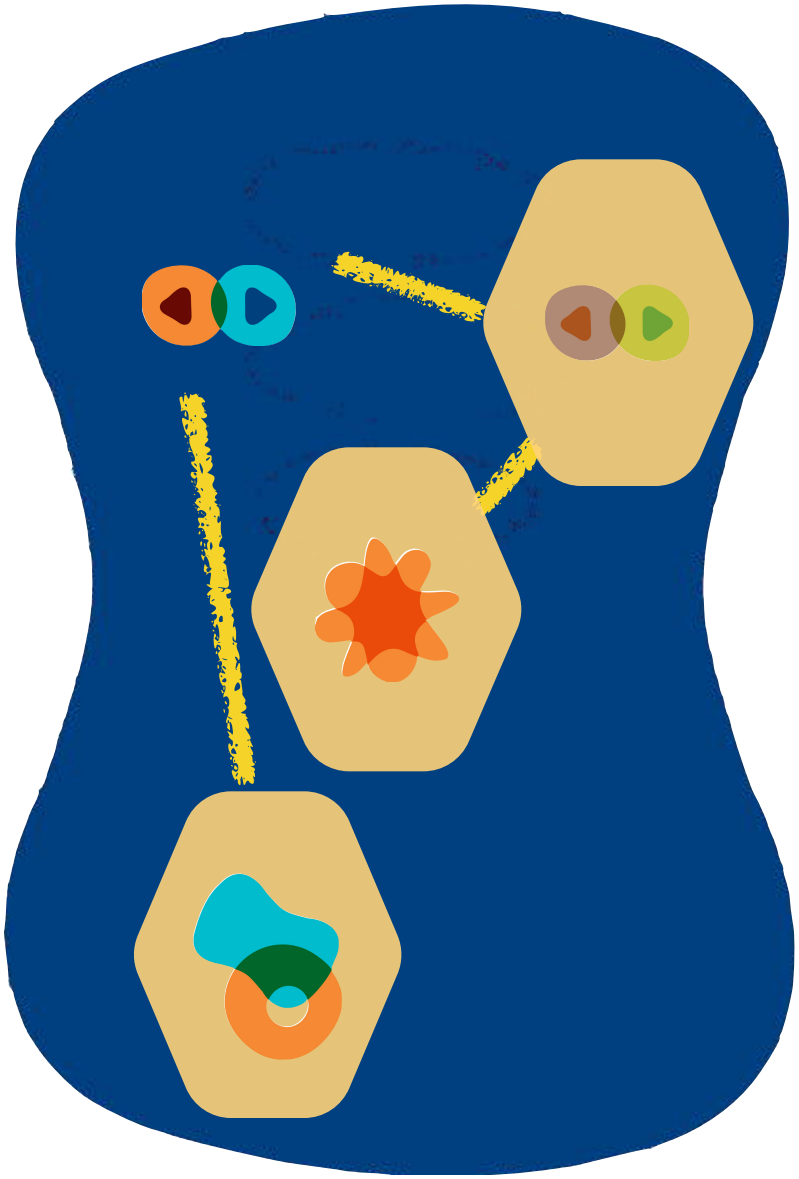
production

Evolutionary Architecture

Components are *deployed*.

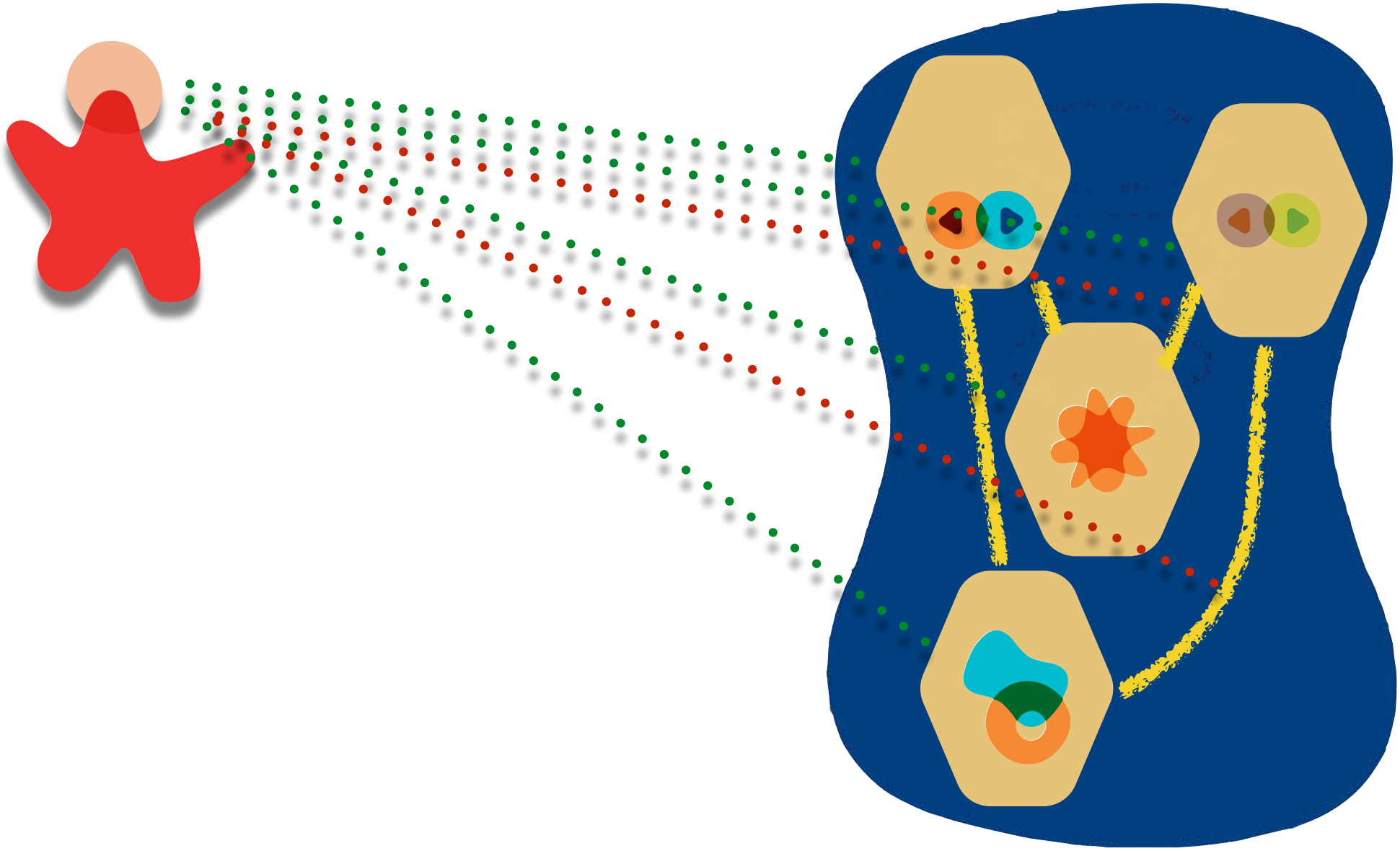
Features are *released*.

Applications consist of *routing*.



production

Evolutionary Architecture

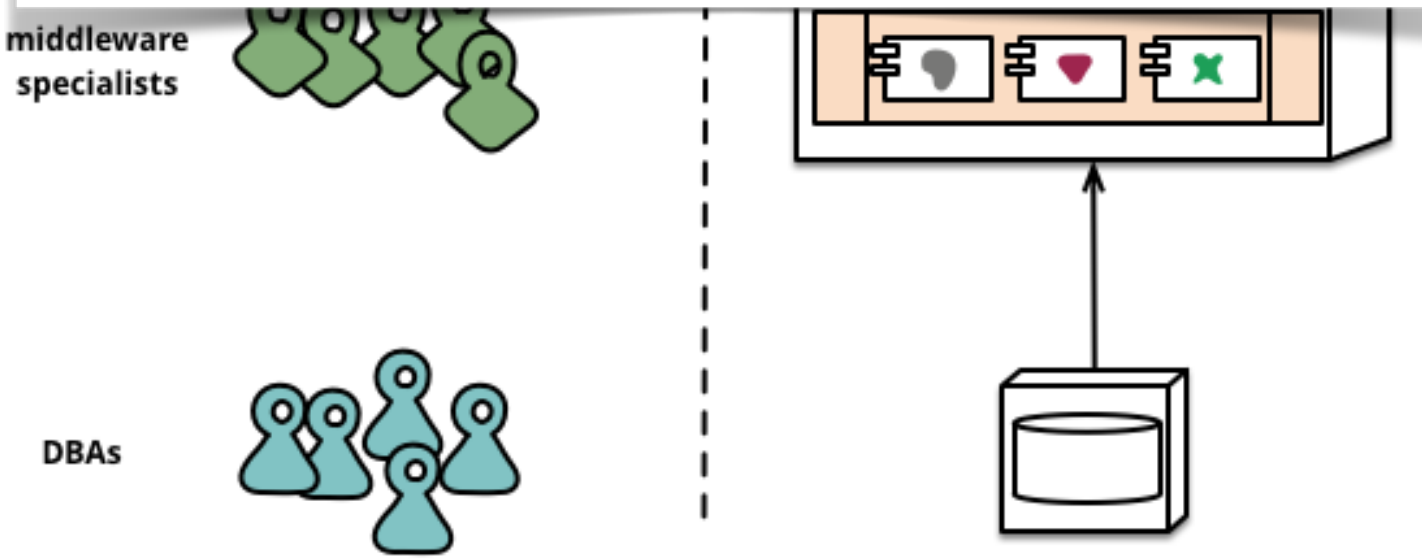


production

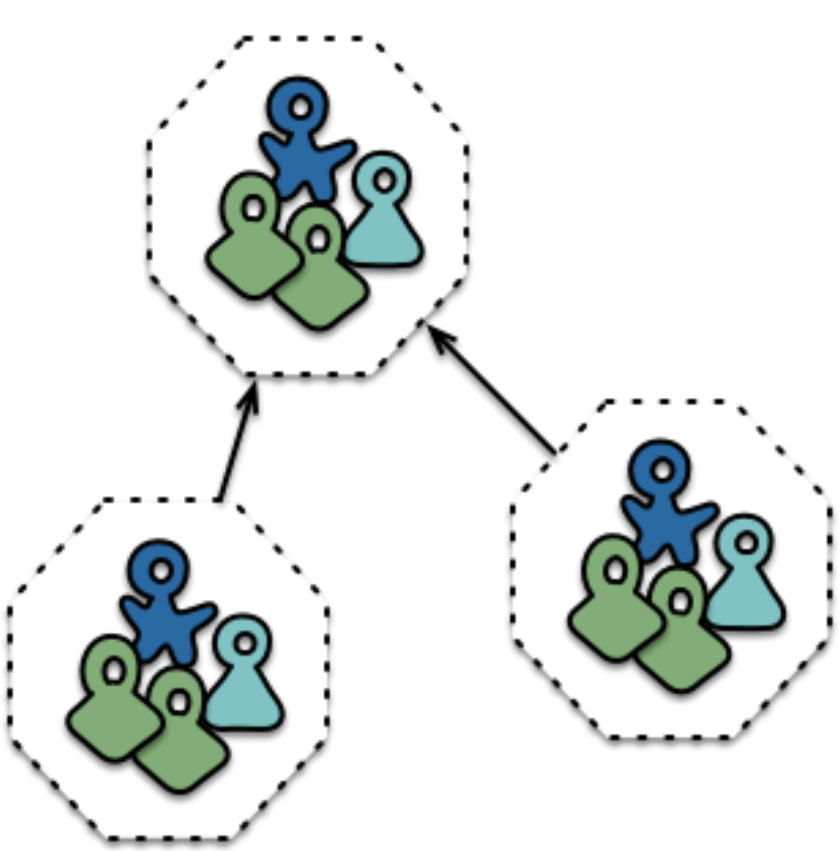
Conway's Law

"organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations"

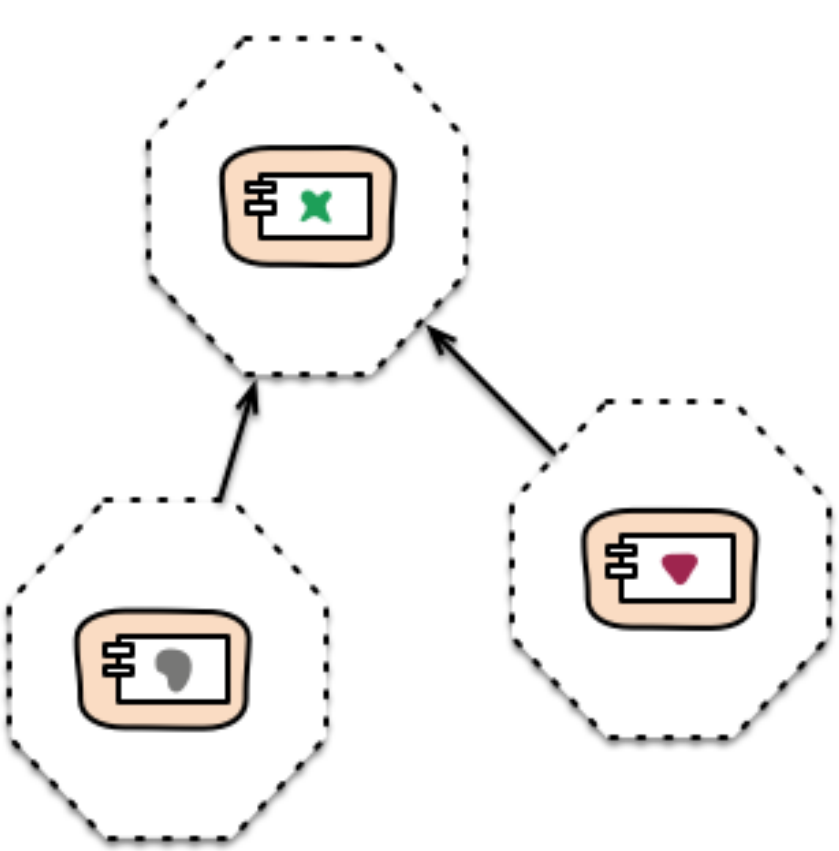
—Melvin Conway



Inverse Conway Maneuver

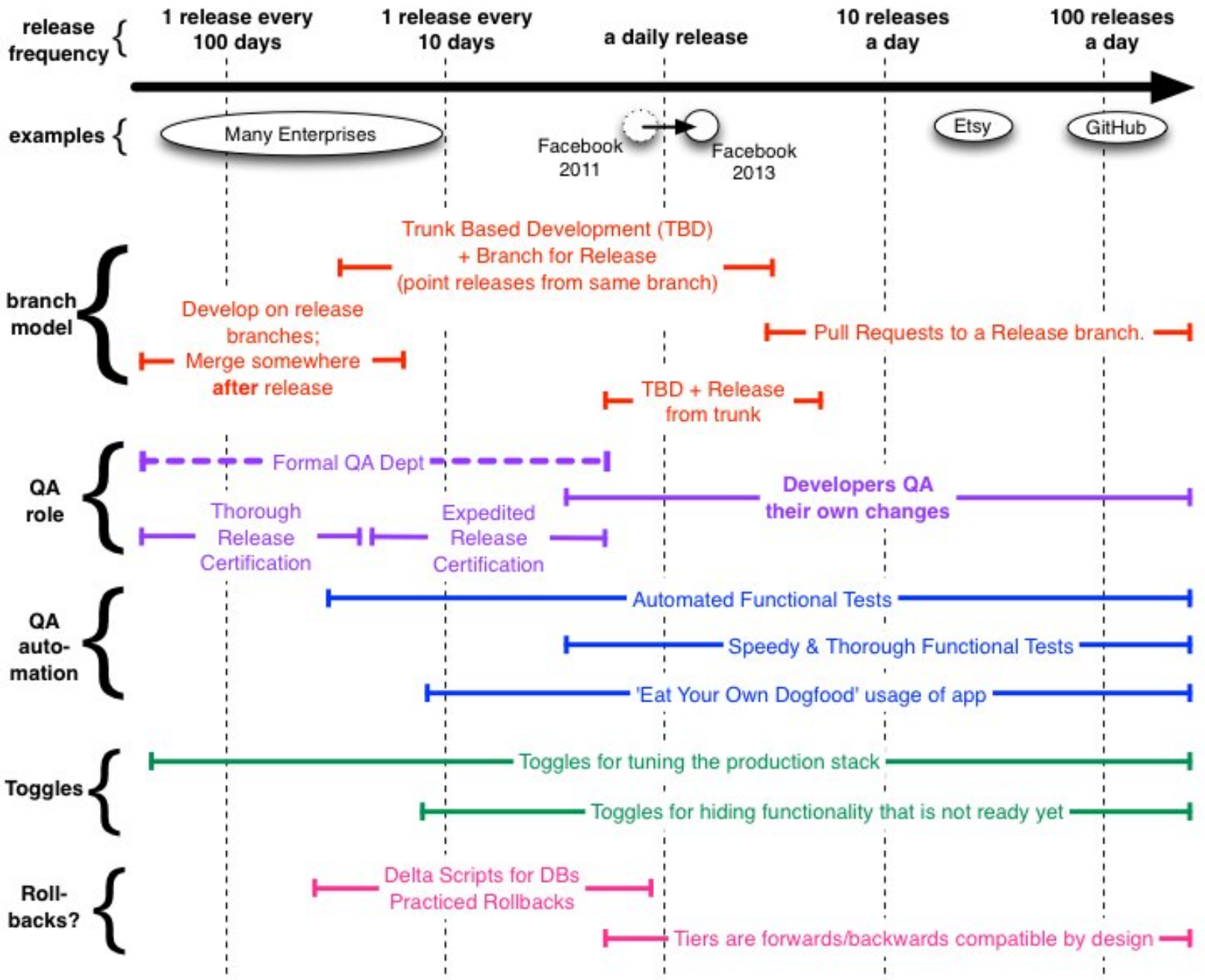


Cross-functional teams...



... organised around capabilities
Because Conway's Law

Continuous Delivery Maturity Model



http://paulhammant.com/2013/03/13/facebook-tbd-take-2/

Continuous Delivery

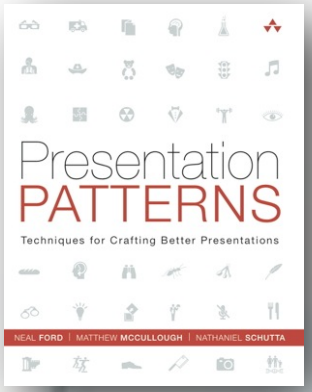
reduce friction

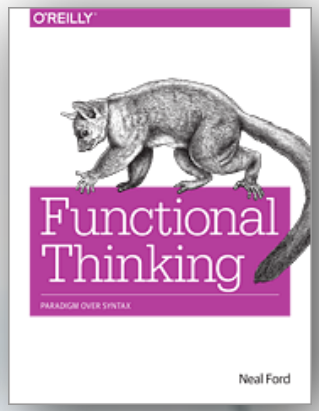
automate everything you can

incorporate everyone into Continuous Delivery practices

measure success via cycle time

continue to improve





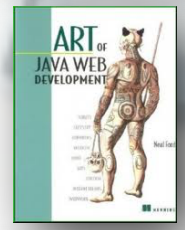
nealford.com

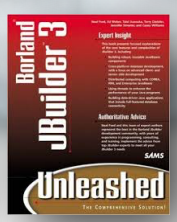
@neal4d

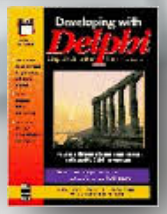
ThoughtWorks®

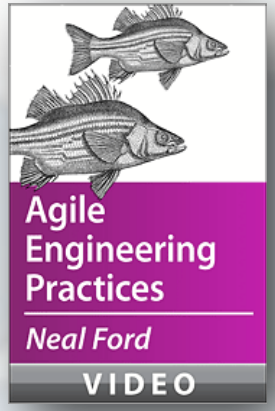
NEAL FORD

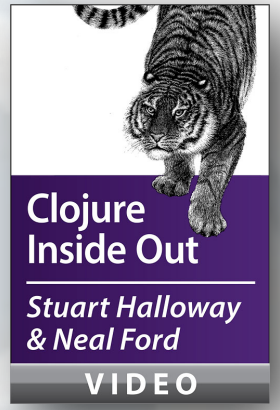
Director / Software Architect / Meme Wrangler

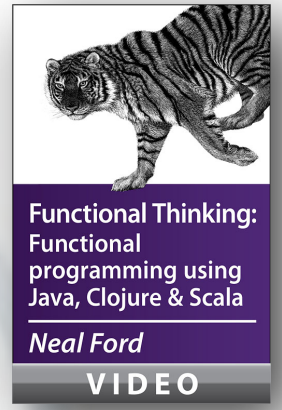












O'REILLY®

SOFTWARE ARCHITECTURE SERIES