Continuous Integration and Deployment

Applications to Microservice Systems, with a link back to Git

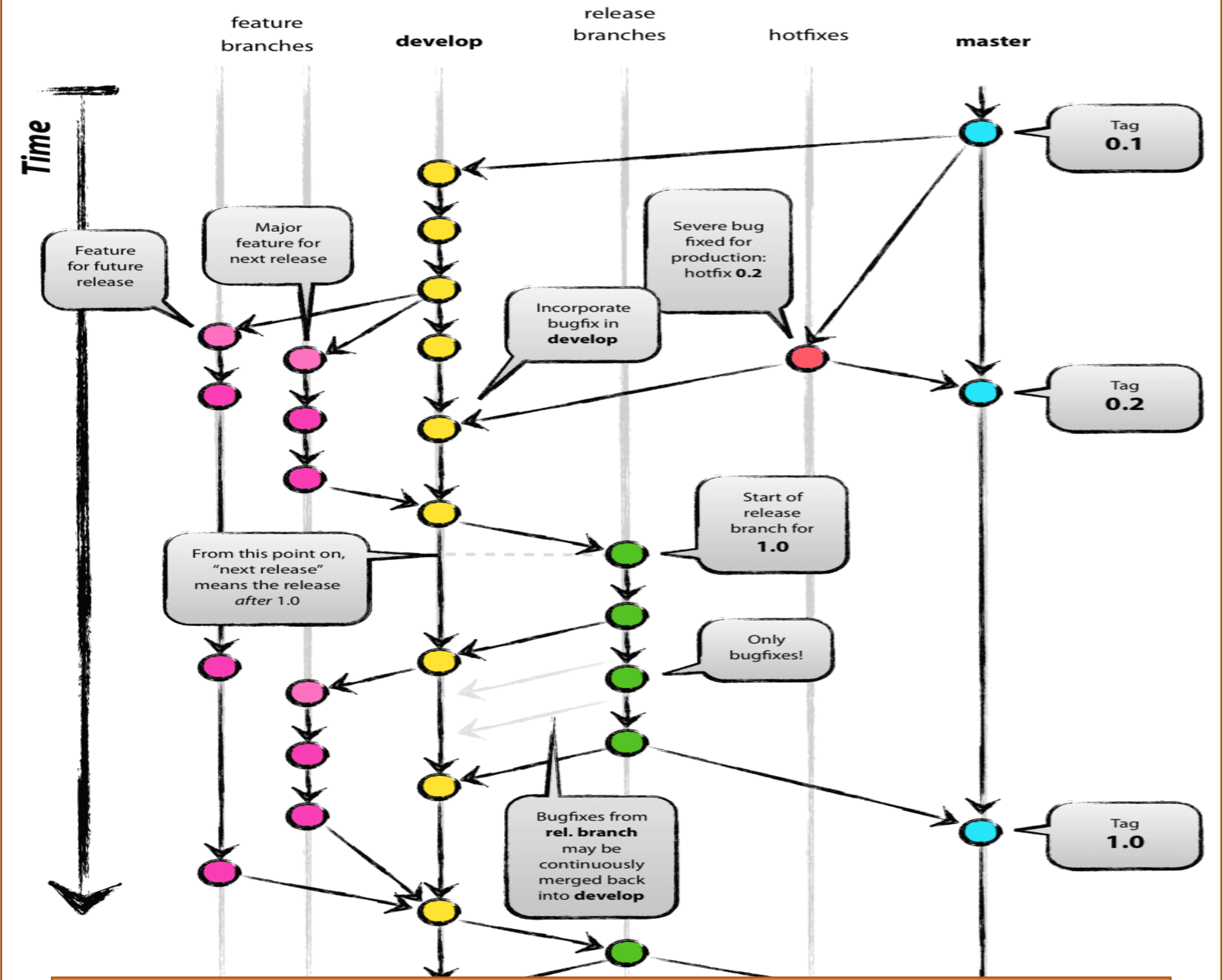
Bootstrapping Your Projects

- I encourage you to follow some form of Test Driven Development
- Create mocks for all of the services that you need.
- Create tests for each service and for the system as a whole
- Get your project’s “skeleton” right with build, test, deploy, etc procedures before you worry to much about specific service implementations.
- ”Fake it until you make it”

The Microservice Mantra Reminder

Compose your system from reasonably small parts, or services. A developer can quickly understand a service's code and how to modify or replace it.

What comes next is the subject of today's lecture.


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Severe bug fixed for production: hotfix 0.2
Major feature for next release
Feature for future release
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

See http://nvie.com/posts/a-successful-git-branching-model/

Git and Microservices

- Each microservice should have its own branch
  - Or even its own repository
- Each microservice could follow the previously illustrated branching strategy
- Tie continuous integration to the dev branch.
  - Don’t break the build
- Tie continuous deployment to the release and master branches
- Develop a CI/CD approach for your entire system, not just individual services.
  - But more about this below

Building, Testing, and Releasing Software and Software as a Service Are Different

Software	Software as a Service
Release software artifacts (zips, tars, dmgs, rpms, etc).	Continuously update capabilities that are composed of multiple internal microservices.
Monolithic build system	Different build systems for each microservice.
Many layers of testing of entire monolith	Limited testing of each service; full system testing more important
One release	Each service gets released, or goes into operation, independently.

Continuous Integration and Deployment Pipeline

Build

- Each service locally

Integrate

- All services on target environment

Deploy

- Updated services to production environment

You should be able to completely automate the deployment of your entire system for testing and deployment: “Hands Free”

Continuous Integration

Winning half the battle

The Golden CI/CD Rule

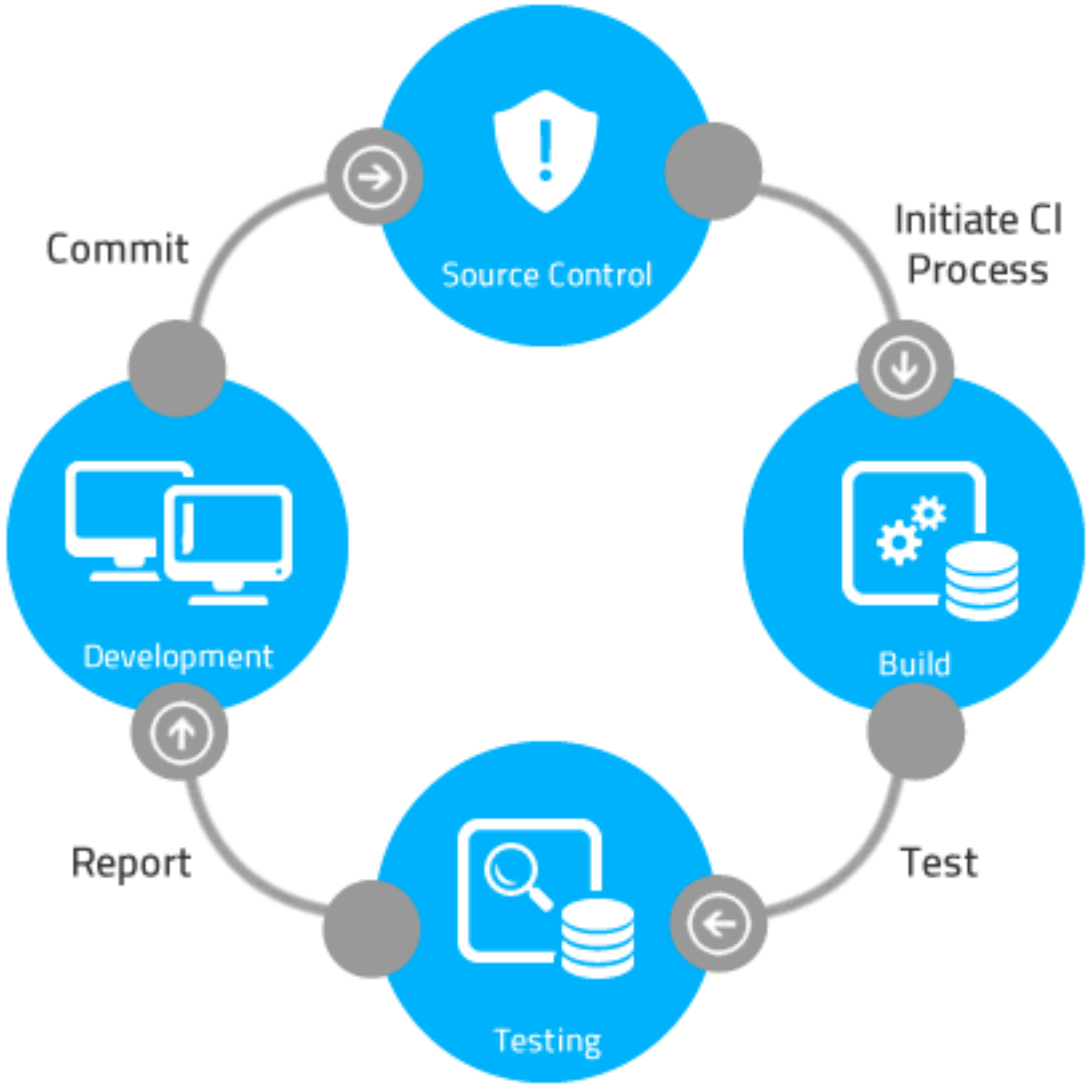
Never login directly to integration or deployment server hosts.
Always use a tool to deploy remotely.

Prerequisites for Continuous Integration

- You have build systems that can build and test your services.
  - Gradle, Maven, Ant, etc are some popular choices
  - These help you organize your projects, too
- **You can automate the deployment of any dependencies**
  - Libraries, run time environments, compilers, databases, web servers, etc.
- You have different branches for your code in GitHub to separate features, integration code, and release code.
- CI applies to your test/integration branch
  - We called this “develop” earlier

Infrastructure as Code

Continuous Integration Cycle



CI is distributed building and testing for multi-developer teams.

At the end of the CI phase, you have a running, fully functioning system that can undergo integration and systems testing.

Continuous Integration

- Continuous Integration (CI) is a development practice that requires developers to integrate code into a shared repository **several times a day**.
- Each check-in is then verified by an automated build that runs unit and some systems tests, allowing teams to detect problems early.
    - Your build system had better be quick
- By integrating regularly, you can detect errors quickly and fix

https://www.thoughtworks.com/continuous-integration

Continuous Integration Practices

- Maintain a single source repository
- Have a defined build process
- Automate the build
- Make your build self-testing
- Every commit to the “integration” branch should build on an integration machine
- Keep the build fast
- Test in a clone of the production environment
- Make it easy for anyone to get the latest executable
- Everyone can see what’s happening
- Automate deployment

Note this is not just for microservices.

https://www.thoughtworks.com/continuous-integration

CI Sequence

1. Developers check out code into their private workspaces.
2. When done, commit the changes to the repository.
3. The CI server monitors the repository and checks out changes when they occur.
4. The CI server builds the system and runs unit and integration tests.
5. The CI server releases deployable artefacts for testing.
6. The CI server assigns a build label to the version of the code it just built.
7. The CI server informs the team of the successful build.
8. If the build or tests fail, the CI server alerts the team.
9. The team fix the issue at the earliest opportunity.
10. Continue to continually integrate and test throughout the project.

https://www.thoughtworks.com/continuous-integration

Extending CI Concepts to Microservices

- Integration and system testing are import for the CI phase
    - Unit tests are not enough with decentralized development
- You have at least two choices on how to deploy microservices for testing...

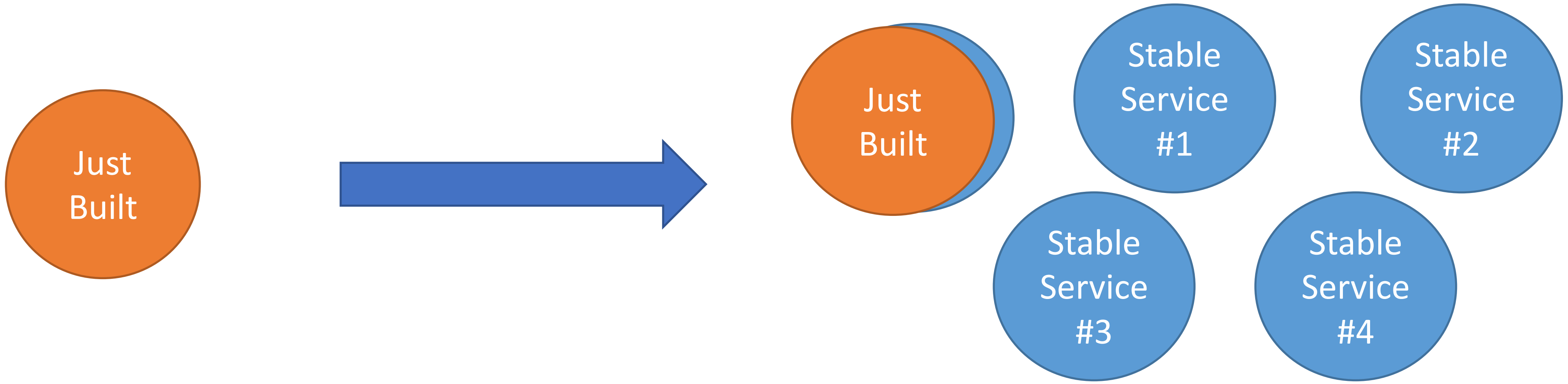
Option #1: Build All Services

- Build and test the whole environment
  - Your current check-in plus all other stable branch services.
- Other branches: could be ”dev” or could be “release”
  - Depends on how you handle deployments (see below)
- Advantage: you know exactly what you are building against
- Disadvantage: may not be feasible with many services, very slow builds
- What kinds of tests?
  - Unit, integration, and end-to-end system tests

This is what you can do for class assignments, but at some point it doesn’t scale up to multiple developers

Option #2: Build and Test One Service at a Time

- Build only your service.
- Deploy your service into a stable production-like setting for testing.
  - Consists of pre-built services based on stable (master or release) versions.
- But beware: this could lead to side effects if other services are actively being changed by other developers.
- Solution: deploy continuously

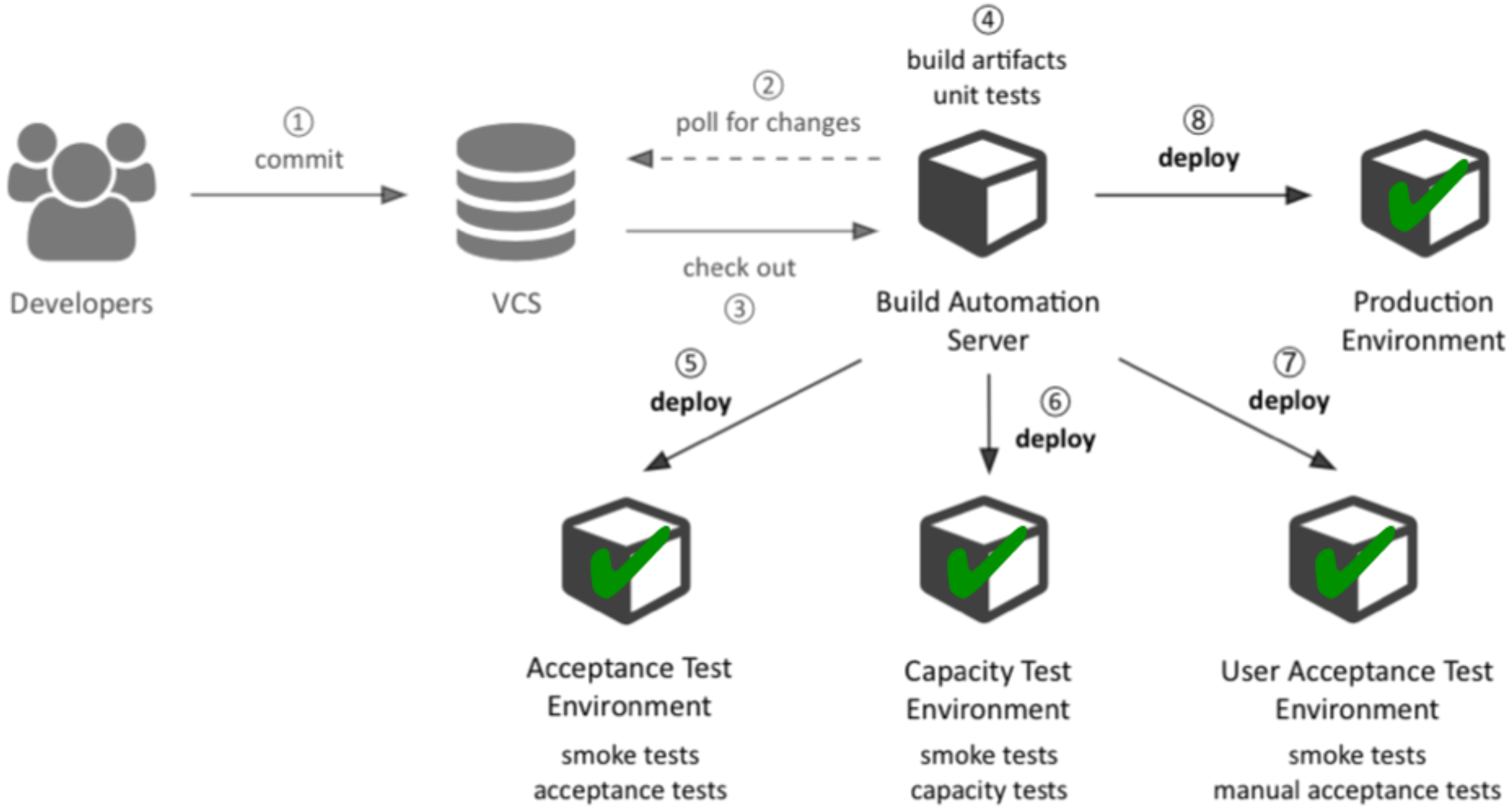


Continuous Deployment

The other half of the battle

Continuous Deployment (CD)

- CD extends CI concepts to frequently push working code into production.
  - ”Hands free” deployment
- The hard part of CD: transitioning and rolling back data behind stateful services
  - Airavata’s registry component, for example
  - More about this in a moment



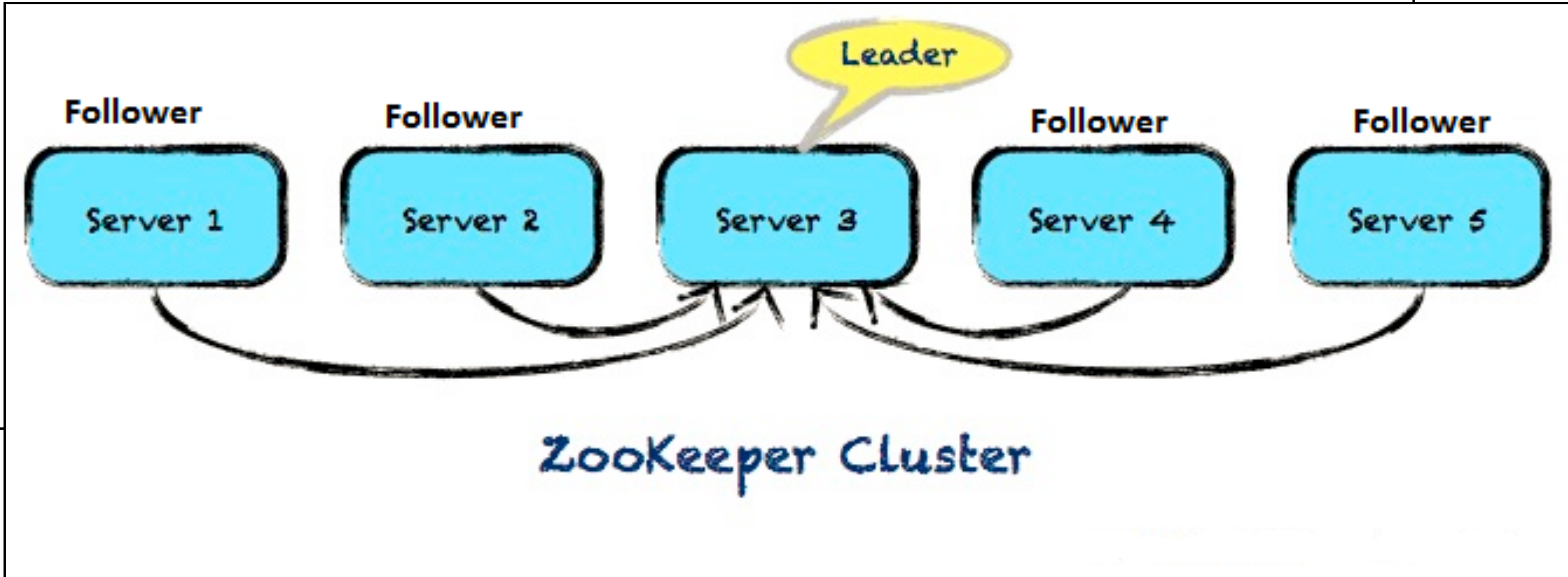
① commit
② poll for changes
③ check out
④ build artifacts unit tests
⑤ deploy
⑥ deploy
⑦ deploy
⑧ deploy
Developers
VCS
Build Automation Server
Production Environment
Acceptance Test Environment
smoke tests
acceptance tests
Capacity Test Environment
smoke tests
capacity tests
User Acceptance Test Environment
smoke tests
manual acceptance tests

Continuous Deployment and Microservices

- Microservices are not just small, they are redundant.
- Each service may be replicated numerous times for at least two different reasons....
  - Real systems are a mixture of these

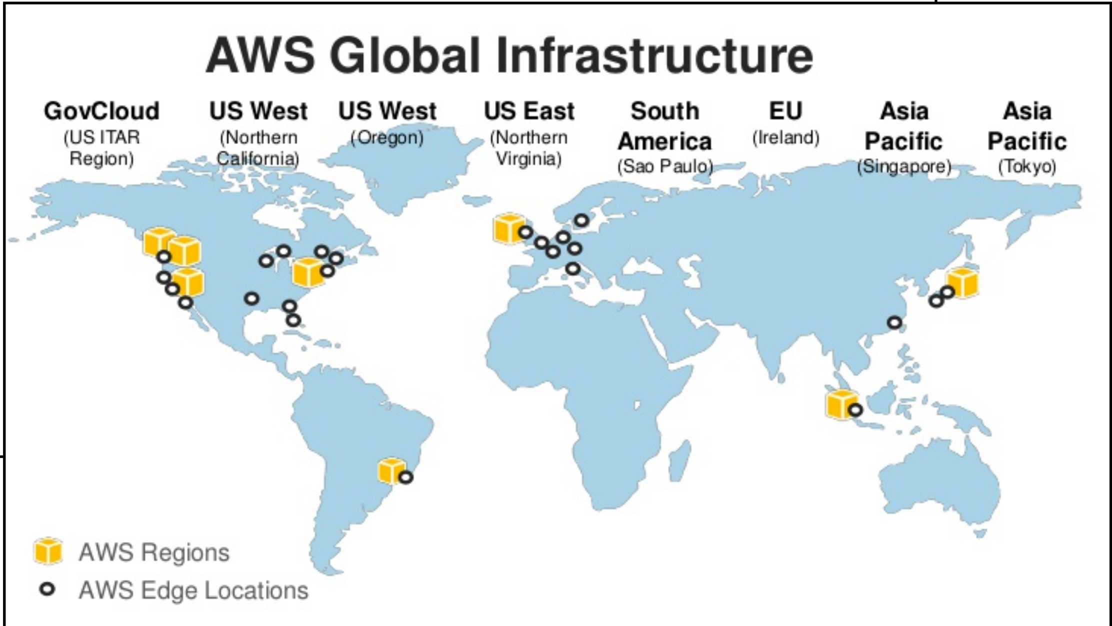
Reason 1: Fault Tolerance

- A backup service can replace the master instance if the master fails.
  - Stateful services like those wrapping databases.
  - “Leader election” situations: Apache Zookeeper, Zab and RAFT



Reason #2: Load Balancing

- Work is distributed among otherwise equal clones of a service for performance
  - Works best for stateless, lightly coupled services.
  - This could includes geographic distribution
  - Messaging systems (future lecture) are used for routing


AWS Global Infrastructure
GovCloud (US ITAR Region)
US West (Northern California)
US West (Oregon)
US East (Northern Virginia)
South America (Sao Paulo)
EU (Ireland)
Asia Pacific (Singapore)
Asia Pacific (Tokyo)
AWS Regions
AWS Edge Locations

Continuous Deployment Strategies

Rolling Updates

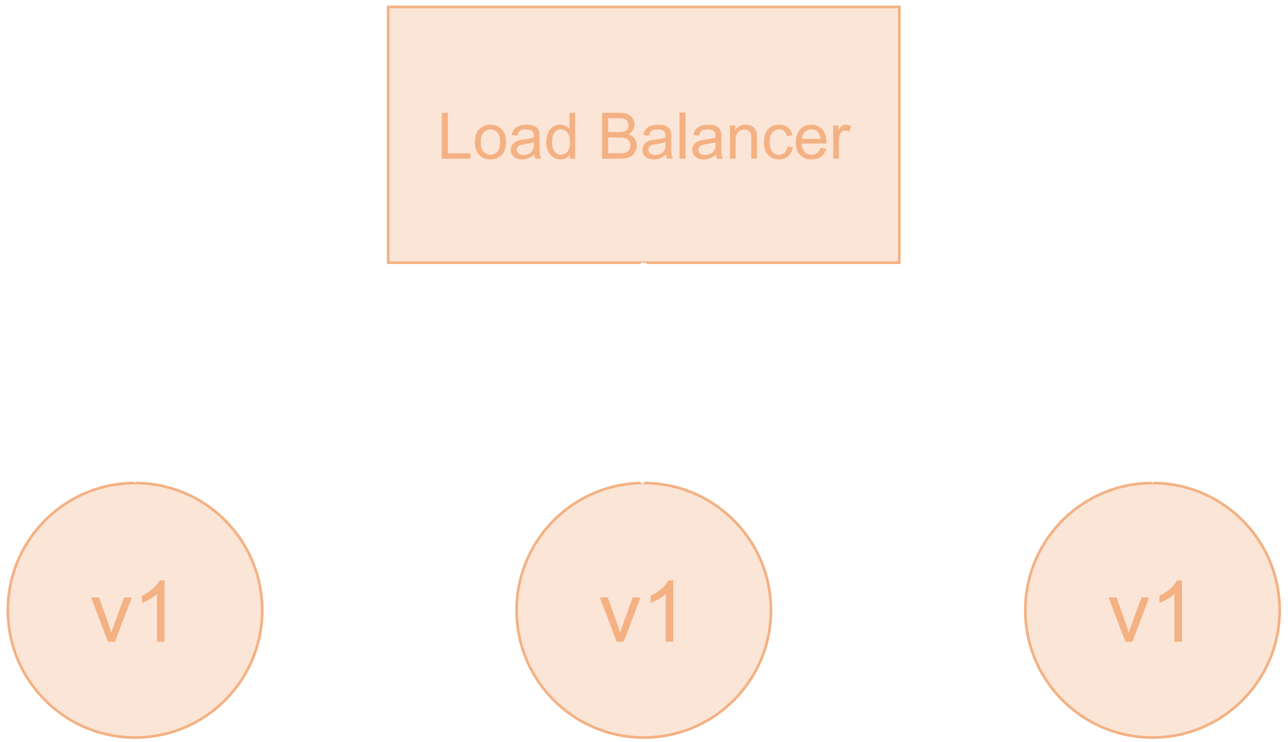
- Incrementally update every instance of every service.
- After each update, verify that the updated service is OK.
- If OK, update the next instance.
- If not, rollback the service instance to the last working version
    - Cancel the update

All-at-Once Updates

- Keep two versions of the system
    - Version Blue: the current production system
    - Version Green: the updated system, deployed with CD.
- Route all calls to Green.
- If there are any problems, switch back to Blue

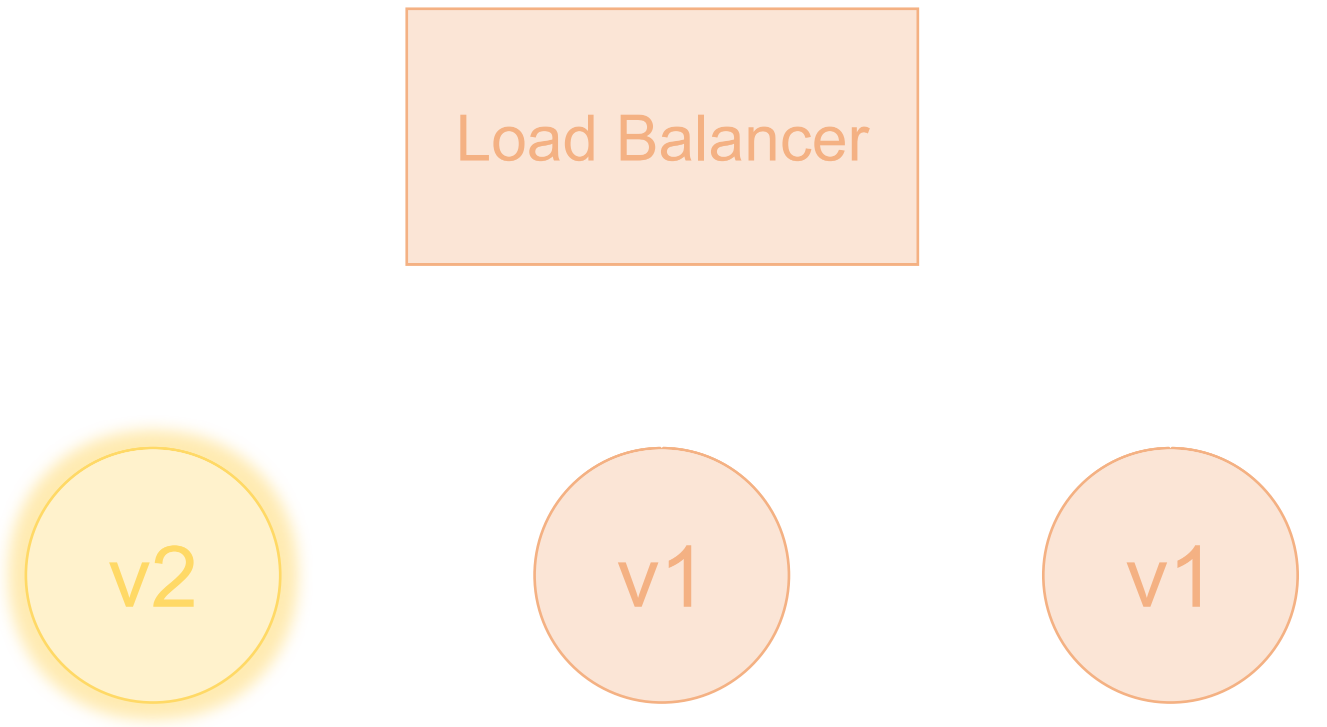
Note again this becomes tricky when services are stateful or record data.

Rolling update – Deploy without downtime



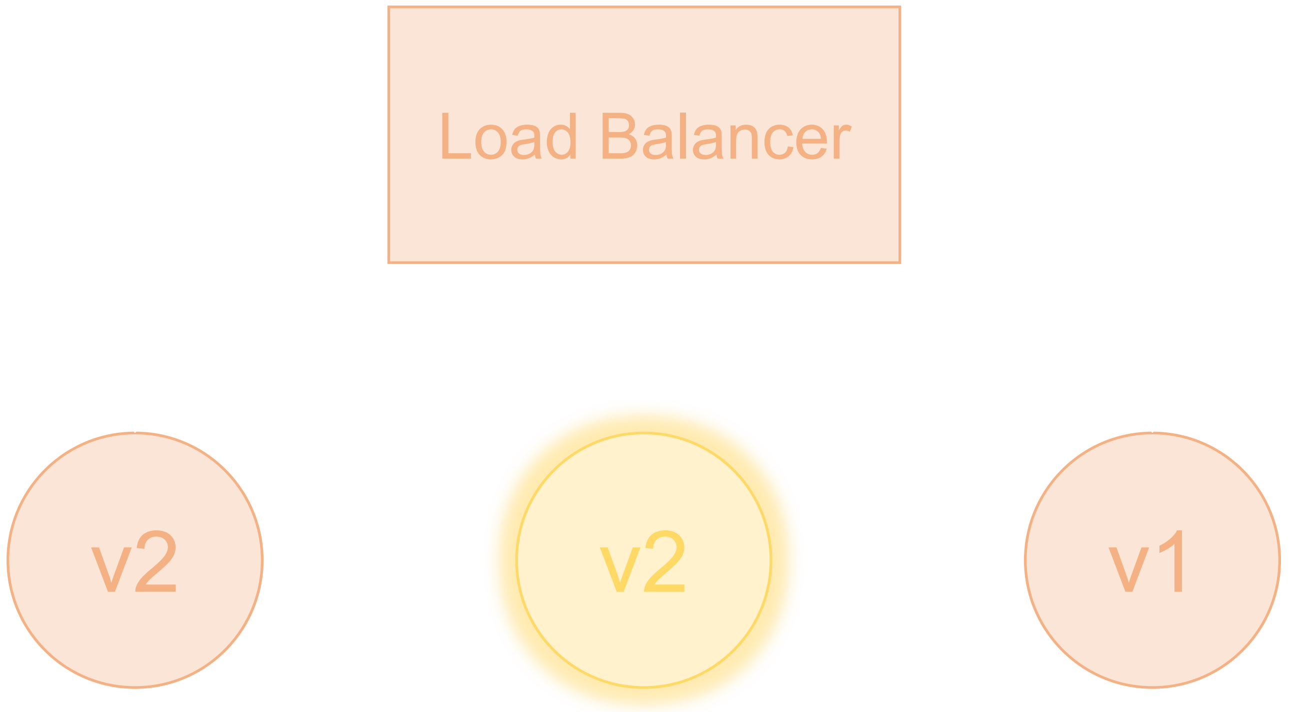
Slides based on Amazon Code Deploy

Rolling update – Deploy without downtime



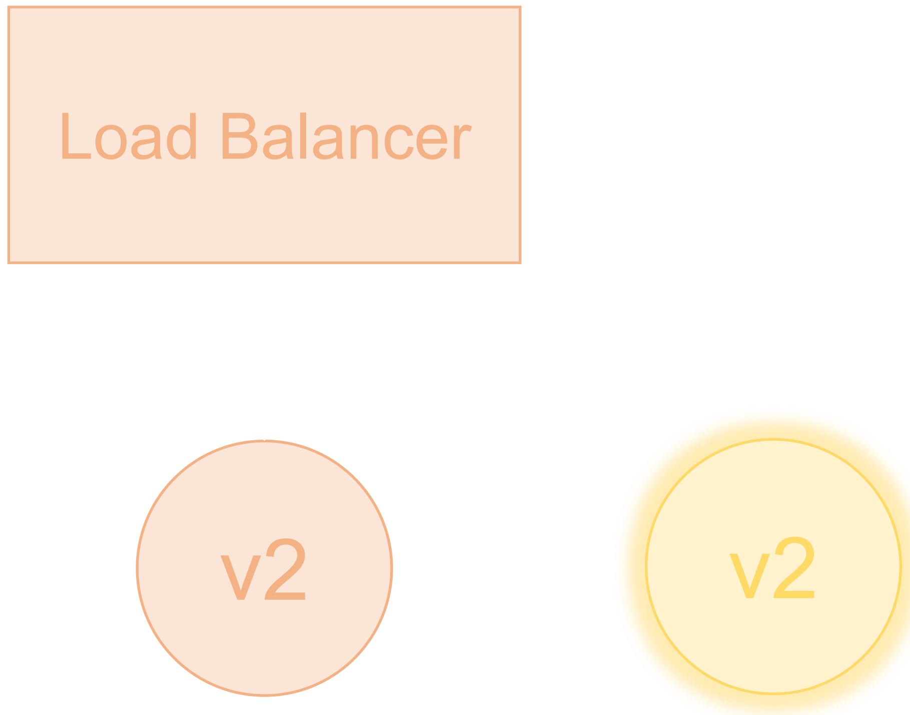
Slides based on Amazon Code Deploy

Rolling update – Deploy without downtime



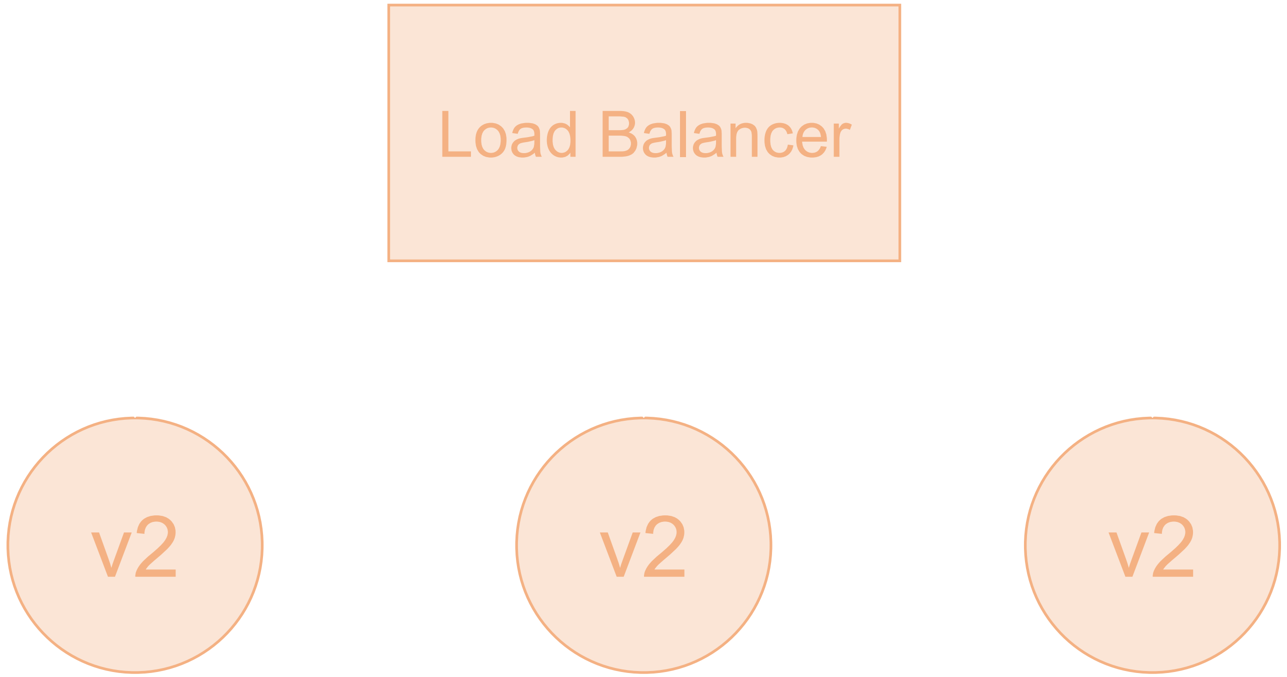
Slides based on Amazon Code Deploy

Rolling update – Deploy without downtime



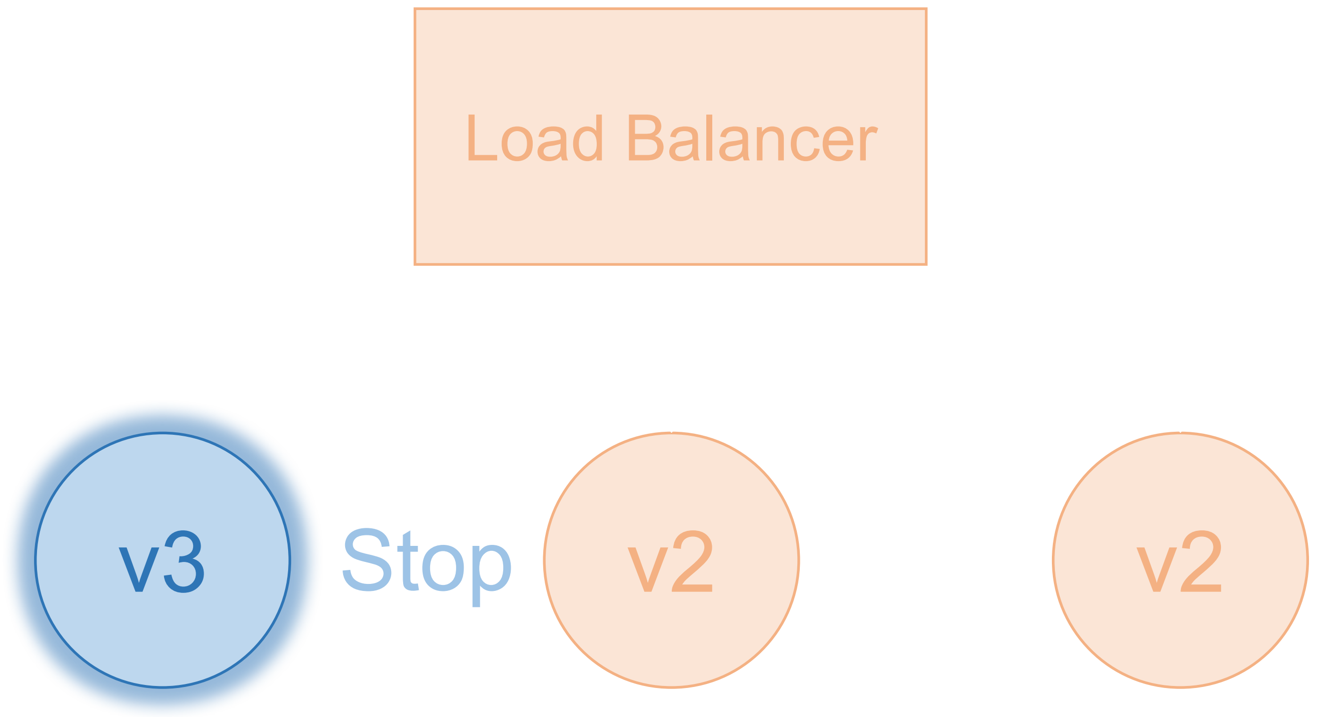
Slides based on Amazon Code Deploy

Health Tracking – Catch deployment problems



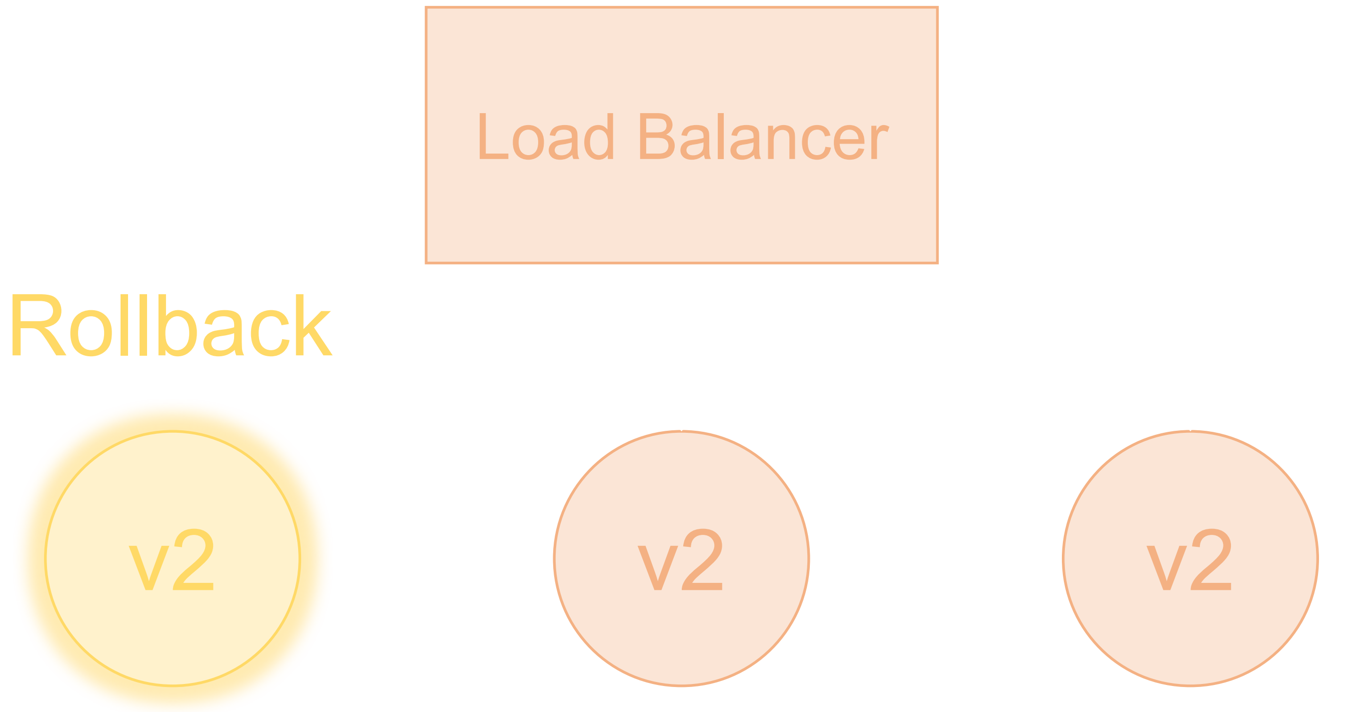
Slides based on Amazon Code Deploy

Health tracking – Catch deployment problems



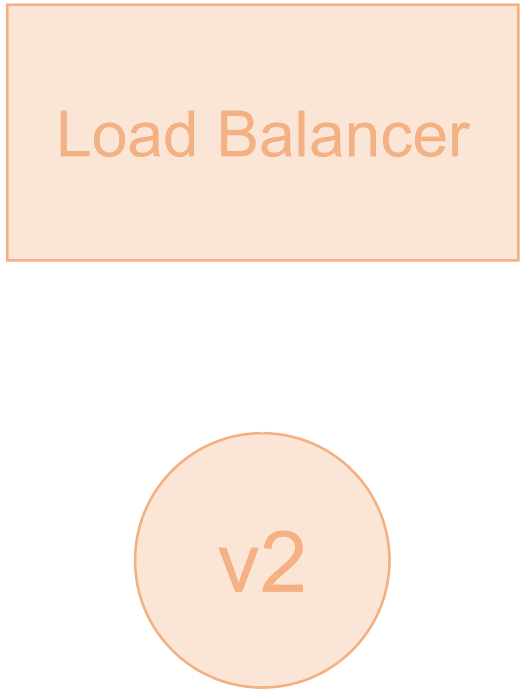
Slides based on Amazon Code Deploy

Health tracking – Catch deployment problems



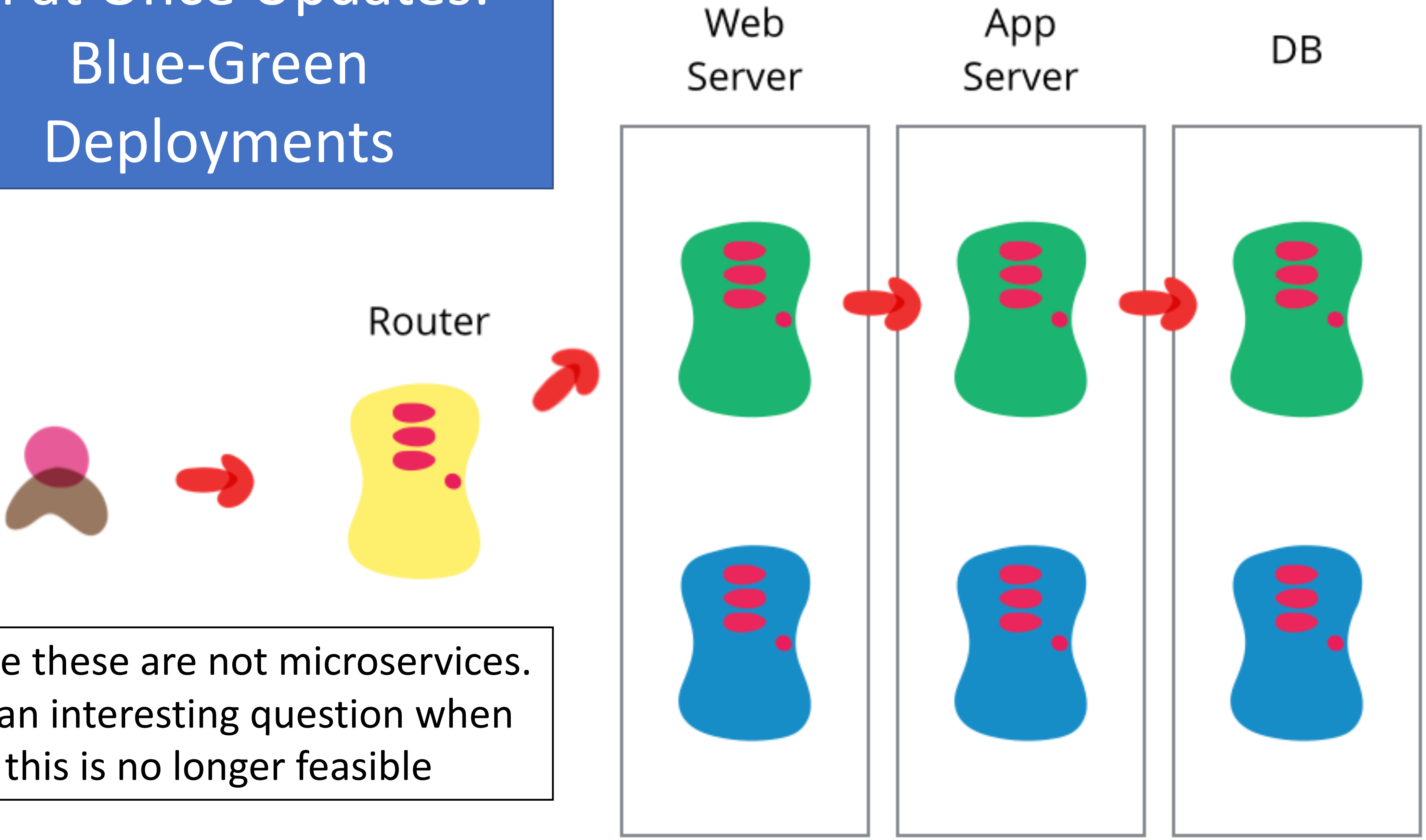
Slides based on Amazon Code Deploy

Health tracking – Catch deployment problems



These examples were based on Amazon Code Deploy, but you can follow a similar approach with Apache Mesos.

All at Once Updates: Blue-Green Deployments



Notice these are not microservices. It is an interesting question when this is no longer feasible

https://martinfowler.com/bliki/BlueGreenDeployment.html

RAFT and CI/CD: A Preview

- The RAFT protocol can potentially be used to manage updates to services.
    - HashiCorp’s Consul is an implementation of RAFT
- This may be an interesting thing to try...
- We’ll have in-depth lectures on RAFT later this semester

Final Thoughts on Update Strategies

- Well-done microservices may result in more stable code
  - You are more likely to add new services rather than fixing old ones
  - This ameliorates ”race condition” update problems when multiple developers are changing services.
- You can apply rolling or all-at-once update strategies to your entire system.
  - At some point, this becomes infeasible.
- You can also apply this strategy to each service or subsets of services.
- For example, you can apply a rolling update strategy for each instance of a microservice
- Or you can update all instances at once, keeping around all the old instances in case you need to roll back
- Having a well thought-out messaging strategy helps with this.

CI/CD and Docker

- Docker allows you to pack in all of your dependencies along with your code into a container.
  - One container per microservice
  - Pack lots of containers into VMs
  - Spread containers across geographically separated data centers
- You still need to test your service before you ship the container
  - Unit tests and mock integration tests.
- The role of CI/CD systems is to orchestrate the deployment and testing of all the microservices (in their containers) and their supporting services onto distributed VMs

Using Apache Jenkins for Continuous Integration and Deployment

A Swiss Army knife for your CI/CD battles

Apache Jenkins, in PMC’s own words

“Jenkins is a self-contained, open source automation server which can be used to automate all sorts of tasks such as building, testing, and deploying software.”

The Jenkins Pipeline

- Jenkins Pipeline is a suite of plugins which supports implementing and integrating continuous delivery pipelines into Jenkins.
- Pipeline provides an extensible set of tools for modeling simple-to-complex delivery pipelines **as code**
- You can define your pipeline using the UI or through a **Jenkinsfile**.

Jenkinsfile

- Text file that contains the definition of a Jenkins Pipeline
- Treat the continuous delivery pipeline as a first class part of your system, side by side with your code.
- This is a nice development, in my opinion.
  - Compare with similar ”CI/CD as code” approaches used by Travis-CI and Amazon CloudDeploy
  - You can check your entire pipeline into Git and keep track of it
  - Code review/iteration on the Pipeline
  - Audit trail for the Pipeline
  - Single source of truth for the Pipeline, which can be viewed and edited by multiple members of the project.

This is a major new development for Jenkins 2.x and best practice generally for any CI/CD system.

Some Problems to Solve

- How do you trigger a pipeline execution?
- How can you use Jenkins to deploy all of your runtime dependencies and supporting services as well as your code?
- How do you pass secrets like passwords to Jenkins without checking them into Git?
- How do you remotely execute commands on your deployment destination VMs?