Continuous Delivery
Jez Humble &
Dave Farley

Building Microservices Ch 7

Build

"What happens when a developer has a change ready to checkin?"

Continuous Integration (CI)
(Jez Humble Q's)

1. Do you check in to mainline once per day?
2. Do you have a suite of tests to validate your changes?
3. When the build is broken, is it the #1 priority of the team to fix it?

Branching Models

- Feature branches, merge to trunk (trunk-based) w/ Feature Flags (↑)
  "Integrate early, and integrate often"
- Small, readable patches, and automatic testing of changes make everyone more productive

Build Pipelines & Continuous Delivery (CD)

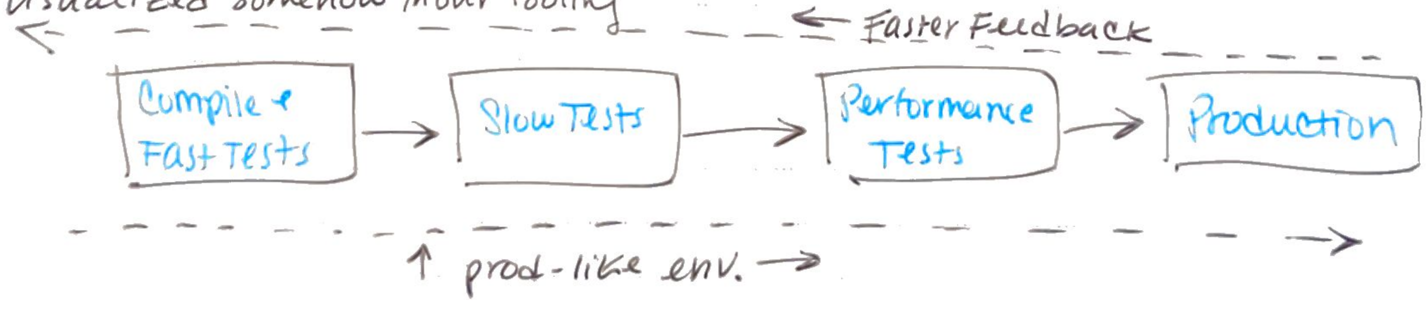
* different stages in a build make up the build pipeline, try to fail fast & early
* Create deployable artifact & run it through each stage of the build to ensure quality before deployment.

"CD is the approach whereby we get constant feedback on the prod. readiness of each & every check-in, and furthermore treat each and every check-in as a release candidate"

Continuous Deployment is where code that passes all stages of continuous Delivery gets automatically deployed.

* You can do cont. delivery w/o doing continuous deployment. You CAN'T do cont. deployment w/o cont. delivery

* Some stages may be manual (ex. UAT) and these steps should still be visualized somehow in our tooling



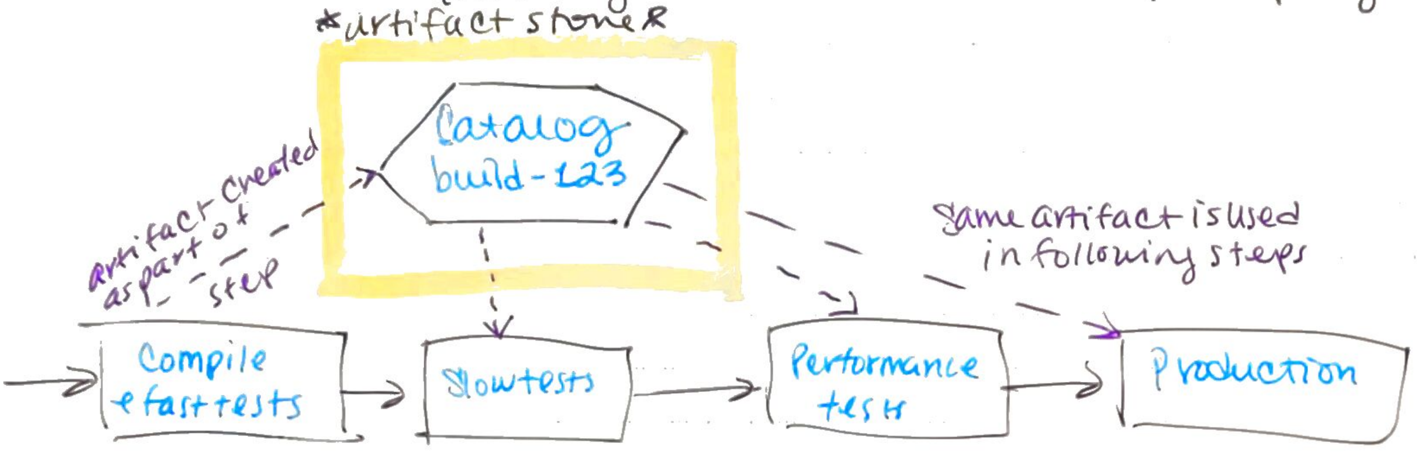
Have to find a good BALANCE

Artifact Creation

- Assume it's a single deployable blob for now

① build an artifact once & only once

② the artifact you verify should be the artifact you deploy



* any aspects of configuration that vary from env. to env. need to kept outside of the artifact itself.

Mapping Source Code & Builds to μservices

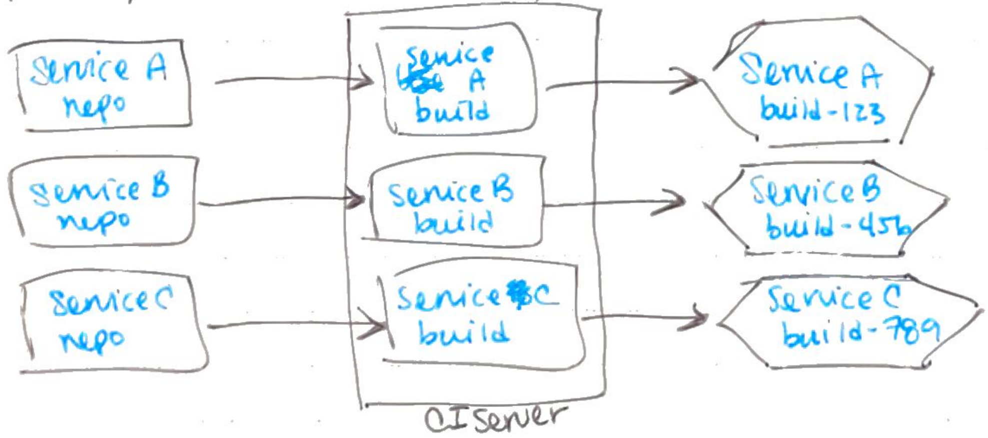
① One giant repo, One giant build (↓↓)

* any commit will trigger a build + verification for all μservices, regardless of what got changed

* good for lockstep releases, possibly early on in a project for ↓ time

* can waste time & make it difficult to tell what should be ~~to~~ deployed so some companies just deploy everything

② 1 repo - 1 μservice (aka multirepo) (↑↑)

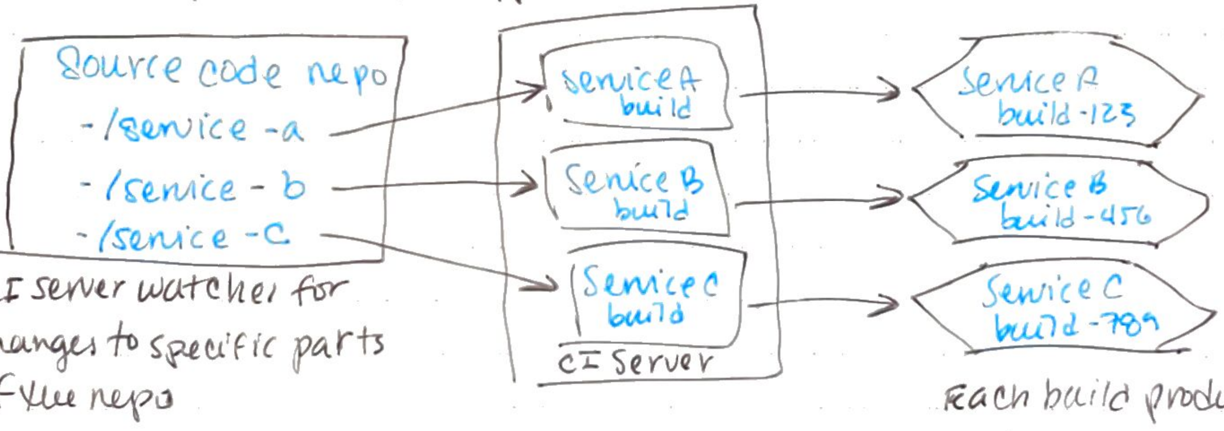


* can easily change ownership / repository

* devs may be working across repos & changes can't be atomic

② multirepo cont'd
- * still have to be aware of code reuse & deployability issues
  - * if you're changing code in multiple repos, lack atomicity of commits means you have to think about staging commits & how to undo / rollback
    - ↳ if this is happening often, service boundaries may not be right
    - "Cross-cutting changes should be the exception, not the norm"
- * the pain of working across ∞ repos can help enforce μservice boundaries
  - → pushing / pulling to ∞ repos can be easier w/ a good IDE or a simple wrapper script to make life a bit easier

③ monorepo
- * code for ∞ repos / other projects is in the same repo
- * changes can be made across ∞ projects in an atomic fashion
- * ↑ code visibility, ↑ code reuse, ↑ changes on ∞ projects
- * still need to consider order of deployment to avoid lockstep deploys
  - ↳ map code to deploys, possible 1 folder → 1 build?



- * ↑ complex w/ more involved ~~f~~ folder structures
  - Bazel tool, build tool Lerna tool
- * ↑ finer grained code reuse across projects which can cause ↑ complex build mapping

- Strong Ownership, code is owned by a specific group where they must do the change desired by people outside of the group
- Weak Ownership, there are defined owners + people outside of the group can make changes that must be approved / reviewed by an owner
- Collective ownership, (< 20 devs) dev can change any μservice
  - * you may be able to specify owner of dirs or filepaths CODEOWNERS file
  - * ensures owners are pulled in on code reviews for their filepaths

Monorepo cont'd

- ★ You can also have per team monorepos
- ★ either very large or very small teams can make this work
  - ↳ pain happens w/ companies in the middle size