Photo Credit: Emily Reimer , Unsplash

CI/CD 101

Niall Byrne

About the speaker, and this talk.

- I’m Niall (like the Nile river, or something else phonetic like that.)
- A 10 year veteran of our field, with a background in SysOps and Application/Tooling Development.
- To understand CI/CD, it might help to start with the problem CI/CD is there to solve, and get a big picture view **BEFORE** sitting down to write pipeline code.
- Notes, slides and resource links are all available at the end!

Untangling Definitions

- CI/CD is one of a group of difficult to unravel definitions in our field.
- DevOps is a related term, and can be just as confusing.
- It can help to separate meaning into Connotation and Denotation

Photo Credit: Hilde Demeester, Unsplash

Connotation of CI/CD

- **People**
  - Like the “DevOps Guy” who manages your pipeline
  - If your company is big enough, a “Platform Team" who handles this kind of stuff
- **Tools**
  - Firebase CLI, Vercel, Github Actions, Jenkins, and many, many more…
  - Pipelines themselves fall into this group, and are a technical implementation of CI/CD.
- **Practices, Methodologies, Habits**
  - Often we know that Development Teams follow common practices, and patterns but don’t always know “Why?”
  - I’d like to suggest there’s a lot of value for individual developers in building understanding and perspective in this area. (Much more than a how/to on a specific tool.)
- **Relationship to DevOps**
  - DevOps and CI/CD are very interwoven. I find it hard to believe you could do DevOps without an implementation of CI/CD of some kind.
- **Frustration**
  - Anyone who’s ever had to troubleshoot a pipeline…

Denotation of CI/CD

Simpler to start with terse definitions of:

- Continuous Integration
- Continuous Delivery
- Continuous Deployment

Photo Credit: Romain Vignes, Unsplash

Denotation of CI

Sources of Information:

- **Martin Fowler**'s Article on Continuous Integration
- **The Agile Software Manifesto**
- **Kent Beck**'s concepts of **Extreme Programming** and **TDD**
- Surprisingly, there is a lot of overlap with something the folks at Heroku called "**The 12 Factor App**".

Coarse Dictionary Definition:

- Merging Code Continually

Photo Credit: Susan Q Yin, Unsplash

Integration Hell 1/2

In traditional software development this was the process of pulling together everyone's changes and attempting to produce a release.

- **Ugly symptoms of Integration Hell:**
  - Merge Conflicts
  - Missing Dependencies
  - Missing Data or Configuration Files
- **Nasty secondary complications:**
  - Duplication of code
  - Duplication of specifications, documentation or configuration

Photo Credit: Peter Leong, Unsplash

Integration Hell 2/2

- The **Synchronization** problem
  - Developers working in isolation eventually need to exchange code and acquired knowledge.
- **Communications**
  - A central theme in both DevOps literature and a central theme in Extreme Programming, and Agile as well.
- The **Team** Mindset
  - Solo developers working on a project don’t encounter Integration Hell.
  - But… they should still be thinking about this. Especially if your project is open source.

Photo Credit: Angela Loria, Unsplash

OK Help… How do we solve Integration Hell?

Our Compass:

- **Martin Fowler's 11 Practices**
  - Maintain a Single Source Repository
  - Automate the Build
  - Make your Build Self Testing
  - Everyone Commits to the Mainline Every Day
  - Every Commit Should Build the Mainline on an Integration Machine
  - Fix Broken Builds Immediately
  - 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

Some Additional Support:

- **Kent Beck**
  - Extreme Programming
  - Test Driven Development
  - "Feedback Loops"
- **Heroku's "12 Factor App" Document**
  - https://12factor.net/

1. Codebase
2. Dependencies
3. Config
4. Backing Services
5. Build, Release, Run
6. Processes
7. Port Binding
8. Concurrency
9. Disposability
10. Dev/Prod Parity
11. Logs
12. Admin Processes

Practice #1:

Maintain A Single Source Repository

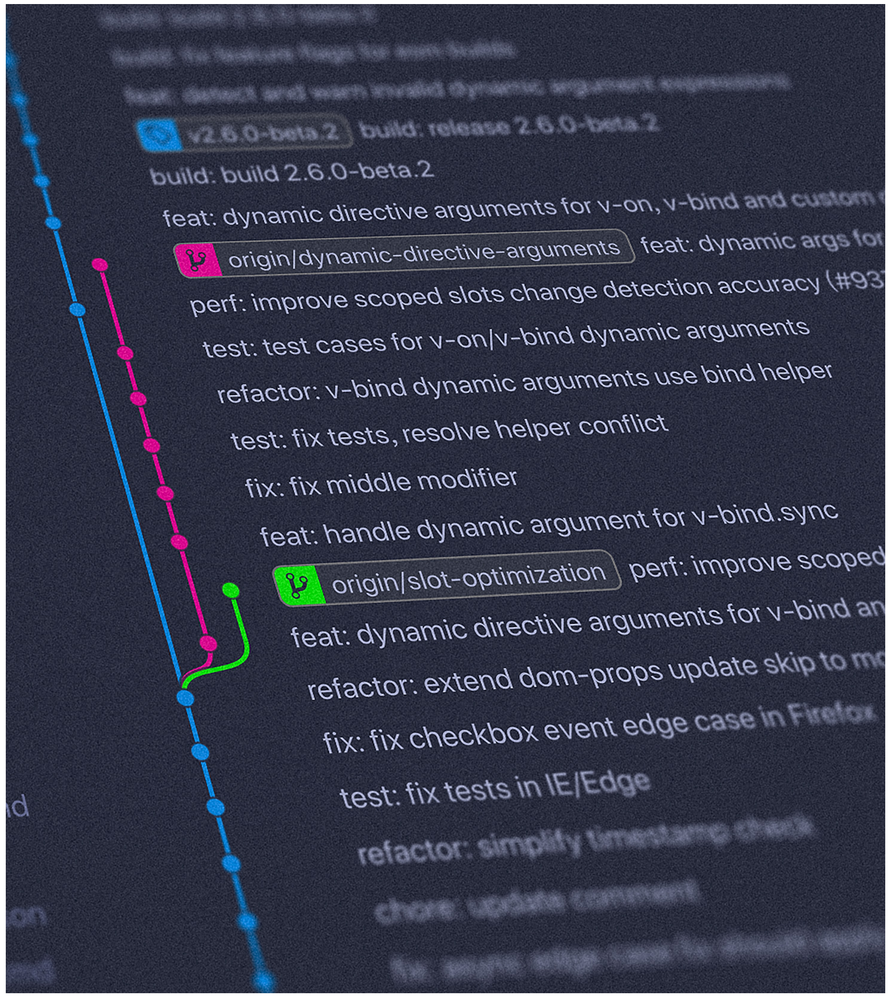
We want to avoid Integration Hell. What goes in the repository? What doesn't?

- **The codebase MUST contain everything needed to build the application:**
  - All Libraries:
    - If it's javascript, include your package.json
  - OS Level Dependencies:
    - Anything you install with an OS package manager. Think **brew**, **apt**, **apk**, **yum**. Every library and executable your app needs.
- **You codebase MUST NOT include:**
  - Credentials: api keys, passwords
  - Locations of backing services: database server hostname, redis server hostname, hostname of 3rd party api
  - Use environment variables.

12 Factor App Support:

- Factor 1: Codebase
- Factor 2: Dependencies
- Factor 3: Config

Photo Credit: Yancy Min, Unsplash



Practice #2:

Automate Your Build

OK, we followed practice #1. Everything is in the code base, so what now?

- **Figure out the exact sequence of commands you need to run to build your app.**
  - Don’t be afraid to get familiar with basic BASH scripting. (It will still be around in 10 years.)

```
#!/bin/bash

sudo apt-get install -y git
git clone https://github.com/myproject/webapp
cd webapp
npm install
npm run build
```

  - Keep the environment variables you created from Practice #1 out of the script.
- **Goal: a validated build that can be produced by following the same steps:**
  - If it works on someone else’s laptop, and gives you a working app it’s a success.
- **Work with your framework and CI tooling to solve the environment variable stuff.**
  - Read the documentation of your Framework or Library, and of the CI tool you’re using before writing pipelines.
- **Document as you go.**
  - Build instructions and environment variables are often out-of-date areas of a code base. Things change fast.

12 Factor App Support:

- Factor 2: Dependencies
- Factor 3: Config
- Factor 5: Build, Release, Run

Photo Credit: Lenny Kuhne, Unsplash

Practice #3:

Make Your Build Self Testing

We want to avoid Integration Hell. Can we use automation to check quality?

- **Automated Testing PREVENTS Integration Hell**
  - If my changes aren't breaking my Team Mate's tests, then there's no last minute surprises. We're integrated already.
- **Automated Testing ENABLES Automated Deploys (which we'll cover soon.)**
  - It sounds reckless to automate the deployment of anything without testing it's functionality
  - Testing here sounds like sensible risk management
- **Agile Software Development, and Feedback Loops:**
  - Iterative software delivery is the norm for most modern development teams, you ship features one at a time- enhancing the existing functionality of software.
  - Extreme Programming advocates continuous feedback that keeps everyone focused so that development continues in the right direction without any delays. This is also Iterative. The Product team measures the success of each shipped feature and then plans accordingly.
  - Tests are a Feed Back mechanism. Did my changes break something? Did someone else's changes break my test?

Extreme Programming Support:

- Feedback Loops
- Test Driven Development

Photo Credit: Yancy Min, Unsplash

Practice #4:

Everyone Commits to Mainline Everyday

The best purported remedy to integration hell, is frequent integration.

Do that really awful hard thing, more often and faster so that it's small little pieces of code that get integrated instead of huge change sets that are impossible to review or manage.

- Two popular implementations of this practice:
  - **Trunk Based Development:**
    - Break apart your work into small targeted commits that are slowly streamed directly to the master or main branch. Replace the huge change sets of Integration Hell, with a stead stream of small commits.
    - Requires you to really think through how you're going to tackle a problem, and in what order.
    - Synchronization becomes consuming a stream of your Team Mate's code.
  - **Short Lived Feature Branches:**
    - The shorter the better… long lived feature branches are the path back to Integration Hell.
    - Still requires planning, and controlled precise code commits. Only change what's relevant to your intended outcome. Don't get side tracked, keep refactors in separate commits. (Hard to resist!)
    - Think about the big picture, you need to the keep the PR small. The less you change each time the better.
    - Change sets will be larger than Trunk Based development, but cannot recreate Integration Hell.
    - You must have empathy for the guy you are asking to review your code.

Extreme Programming Support:

"If you integrate throughout the project in small amounts you will not find yourself trying to integrate the system for weeks at the project's end while the deadline slips by."

Photo Credit: Markus Spiske, Unsplash

Practice #5:

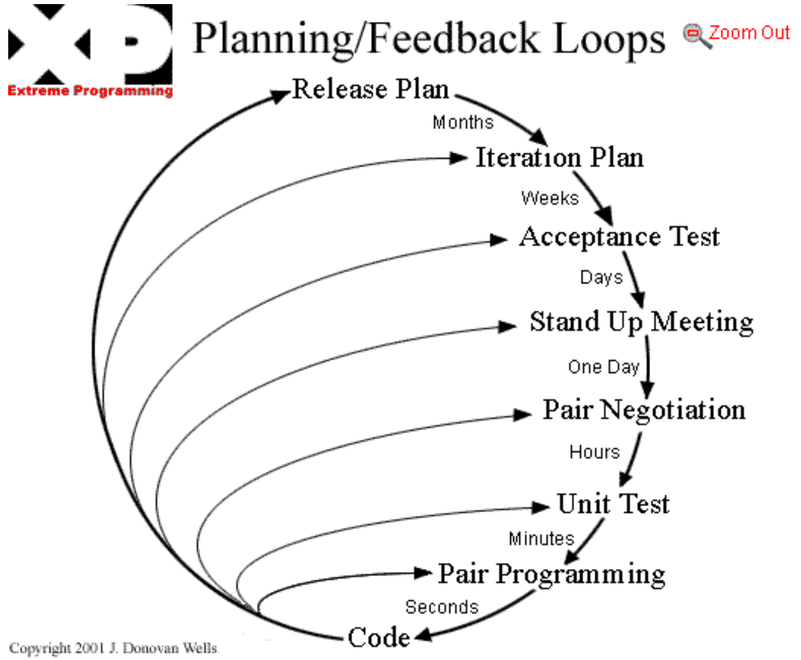
Every Commit Should Build Mainline on an Integration Machine

Finally… we git to some actual pipelines. Well almost …

- **Monitoring For Your Code:**
  - Fowler's article has this concept of using the build as a way to **Monitor** your code base.
  - If every commit to the code base results in a build, you can continually validate:
    - Your work (via the tests your write, and the tests others have written)
    - Whether the build itself is still working
    - Your pipeline configuration is also validate by this process
    - This is all part of iterative feedback, you push to the codebase, the build gives you feedback on the results of your work. You can course correct, or continue as appropriate.
- **Timed or Cron Builds:**
  - Fowler also mentions scheduled, nightly or cron builds, these do NOT serve the same purpose as Continuous Integration, but are immensely useful as codebase **Monitoring:**
  - Scheduled repeating builds are incredibly valuable for codebases that aren't actively being developed, but have third party libraries I want to scan for vulnerabilities (think "npm audit").

Extreme Programming Support:

- Feedback Loops
- Test Driven Development



Practice #6:

Fix Broken Builds Immediately

We want to avoid Integration Hell. How do we keep ourselves on the right path?

- **The Pragmatic Programmer: Fix Broken Windows**
  - One broken window leads to many.
- **“Nobody has a higher priority task than fixing the build.” — Kent Beck**
  - For Trunk based development, this is especially true. For feature based development, it may only be you that’s impacted by the failed build.
  - Regardless, the build automation is your central feedback mechanism that tells you whether or not you and your team is on the right track. Without the build, you’re flying blind.
  - Fowler is very explicit about saying, if you break the build- it’s not a big deal, that’s supposed to happen… but you need to get it working. You need to course correct.
  - Test Driven Development: Red, Green, Refactor - same general principle, you want to get to Green as fast as possible.

Extreme Programming Support:

- Feedback Loops
- Test Driven Development

Photo Credit: Jilbert Ebrahimi, Unsplash

Practice #7:
Keep The Builds Fast

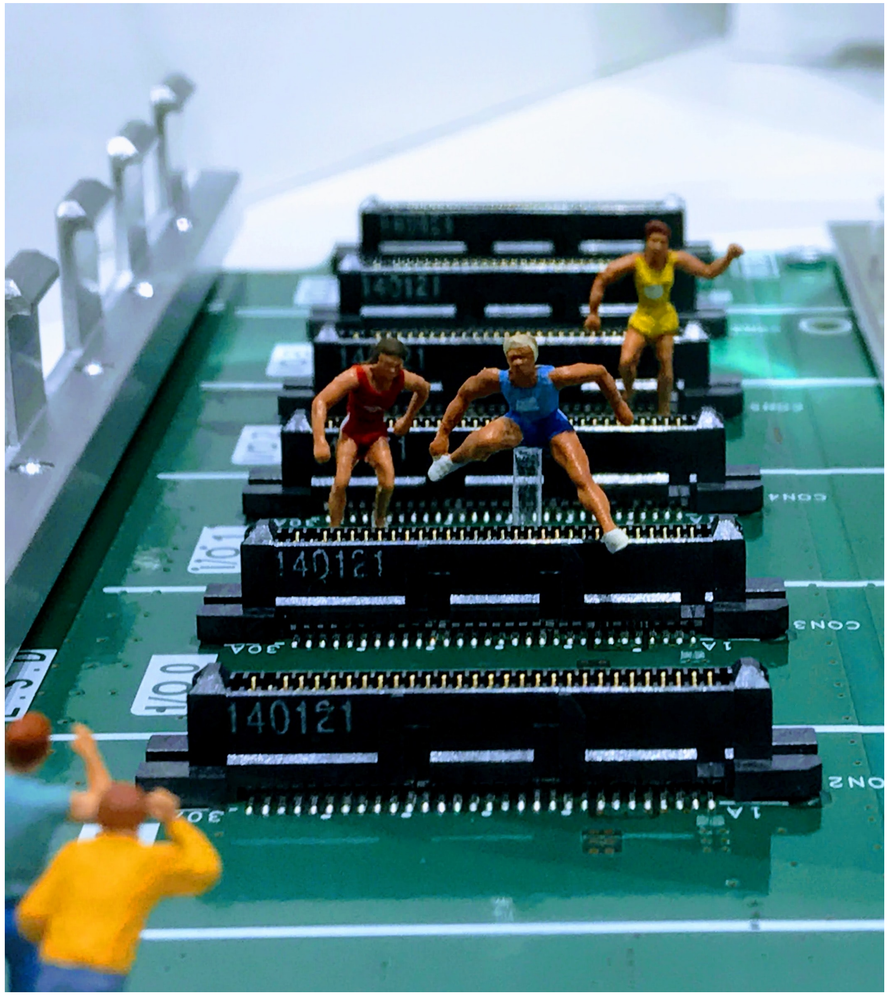
We want to avoid Integration Hell. How can we maintain velocity while keeping on the right path?

- **In order for Feedback loops to be effective, they need to be fast:**
  - Slow builds kill momentum, and can cause people to start ignoring the build.
  - Extreme Programming caps a build length at 10 minutes, but I think most projects easily exceed this now.
  - The faster we can identify a problem, the faster we can respond.
- **Know what you're optimizing, and know how:**
  - Measure. If you can visualize build times before diving into optimizations that would probably be smart.
  - Writing good, fast tests is an art form.
  - Equip yourself. The more knowledge you have of BASH, software testing, the test suite your team is using, and your CI tool the more effective you'll be.
- **Not every step in your build needs to be sequential.**
  - Make sure you know how your CI tool handles concurrency.
  - Look for opportunities to parallelize before writing pipeline code. Measure before optimizing.

Extreme Programming Support:

- Feedback Loops
- Test Driven Development

Photo Credit: John Cameron, Unsplash



Simple Concurrency Example:

Refactored Build, For Concurrency Across Three Machines

```
#!/bin/bash
set -e
sudo apt-get install -y git
git clone https://github.com/myproject/webapp
cd webapp
npm install
npm test
npm audit --production
npm run build
```

This would work well for Github Actions.

It may not be appropriate for Jenkins: be familiar with **your** CI tool.

Concurrency can also be costly. It may be impractical to make EVERYTHING concurrent.

How does your CI tool handle concurrency?

- Processes?
- Additional Machines?

How is it priced?

Practice #8:

Test in a Clone of the Production Environment

We want to avoid Integration Hell. How do we ensure our testing actually represents the user experience?

- **Minimize the differences between Production, Development and Testing**
  - Don’t add unknown variables into your system, your tests will start to become meaningless. Consider the risk management aspect of any production deploy.
  - Don’t forget about backing services. Use the same versions of your database, caching server, any consumed service in all the environments.
  - A dedicated environment for QA testing may be necessary for more involved testing.
- **The Cross Browser Testing Analogy**
  - When we use a different browser on our code, we introduce a new frontend environment.
  - To mitigate the risk of their being differences we proactively review our code across difference browsers during development.
  - Don’t introduce untested environments for your app, be as deliberate as you can be about ensuring your environments are the same- or you’re opening the door for unexpected problems.

12 Factor App Support:

- Factor 2: Dependencies
- Factor 3: Config
- Factor 4: Backing Services
- Factor 10: Dev/Prod Parity

Photo Credit: Ammar Sabaa, Unsplash

Photo Credit: CHUTTERSNAP, Unsplash

10 Minute Break

Practice #9:

Make it Easy for Anyone to Get the Latest Executable

We want to avoid Integration Hell. How do we know we avoided it?

- **There is a great quote from Martin Fowler's article:**
  - "We've found that it's very hard to specify what you want in advance and be correct; people find it much easier to see something that's not quite right and say how it needs to be changed."
- **Feedback Loops that invite participation from Non-Developers:**
  - Agile's iterative approach means functionality is incrementally added or modified as you go. But Kent Beck reminds us, we need something to try trigger to the Feedback conversation.
  - This might be a demo environment, where you can deploy the latest build, or your own feature branch for review.
  - This might also be a prebuilt executable that is generated after every build, including your own feature branches.
  - This DEFINITELY needs to be easy accessible, and not something we're reserved about letting other folks take for a spin or interact with.

Extreme Programming Support:

- Feedback Loops
- Test Driven Development

Photo Credit: Marvin Meyer, Unsplash

Practice #10:

Everyone can see what's happening

We want to avoid Integration Hell. How do we avoid the “Synchronization Problem”?

- **"Continuous Integration is all about communication” -** Martin Fowler
  - Think about any modern CI tool (GitHub Workflows, Jenkins, Circle CI…) they have made a large effort to make it easy to see what’s happening, and where.
  - This “openness” triggers the Feedback loop:
    - Broken builds get fixed sooner.
    - Duplication is avoided, and spotted sooner.
    - Outside parties can see progress being made.
- **Good dash-boarding facilitates openness and progress tracking:**
  - Build Automation: “A monitoring tool for the code repository”
  - Jira, Trello, Kanban: “A monitoring tool for project progress.”
- **Communication is a powerful antidote to Integration Hell:**
  - The open flow of information cures the “Synchronization Problem”.

The Agile Manifesto, Core Value #1:

- individuals and **interactions** over processes and tools

Photo Credit: Josh Calabrese- Unsplash

Automating Deployments

We're now ready to ship it.

A quick recap, before we ship.

We finally got here. Wait, how did we get here?

1. We have all our code in single source repository, with everything needed for the build
2. We have automated the build.
3. We made our build self-testing. Our tests help us feel confident about what we've built.
4. We are merging code frequently, and not allowing large changes to take us down the dark path to Integration Hell.
5. Every time we commit, we build the codebase, and are assured everything is still working.
6. If something breaks, we fix it right away. We're using the Feedback Loop to help keep us on track.
7. We're optimizing our builds for speed, so that we can cycle through the Feedback Loop quickly and address any problems long before they turn into Integration Hell.
8. We're testing in a clone of the production environment, so again, we have confidence that our build is ready for production.
9. We've made the executable, or website available for easy access to everyone involved.
10. We're embraced open communication and transparency of information to prevent the "Synchronization Problem" from ever clouding our vision.

Photo Credit: Markus Spiske, Unsplash

Denotation of CD

Coarse Dictionary Definition:

- Deploying or Delivering Software Continually
- **Super Simple Starting Point:**

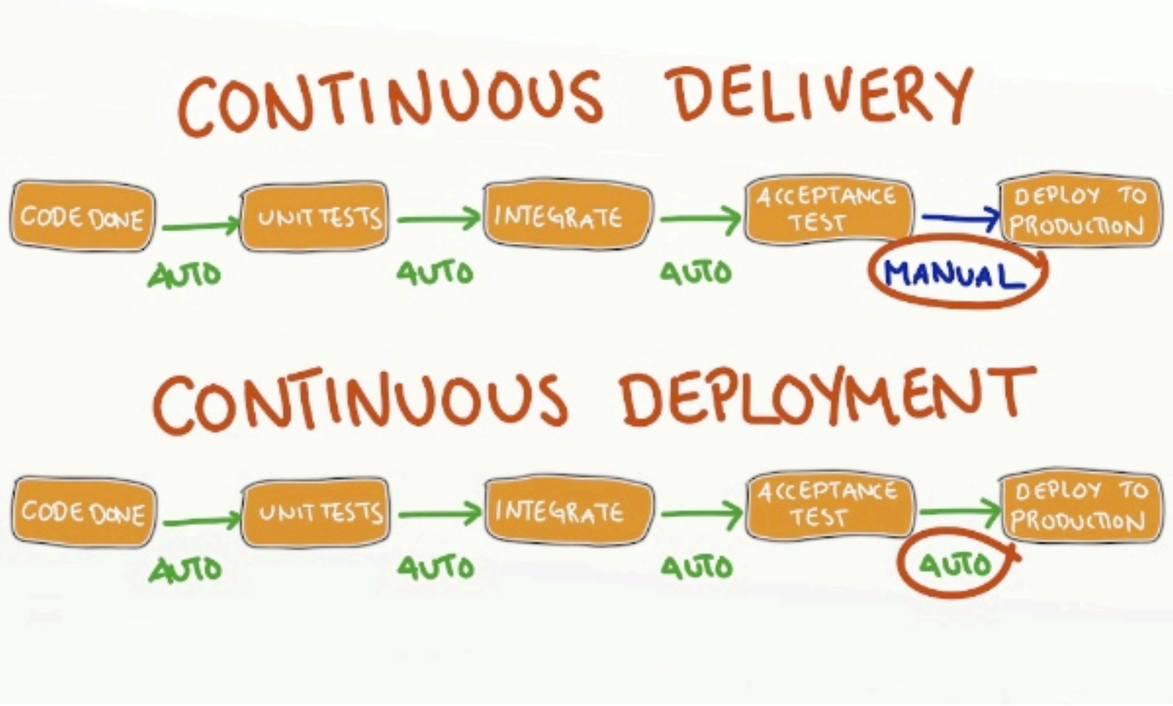


Image Credit: Sanjay Zalavadia (SD Times)

Photo Credit: Phil Hearing, Unsplash

Practice #11:
Automate Deployment

Ship it.

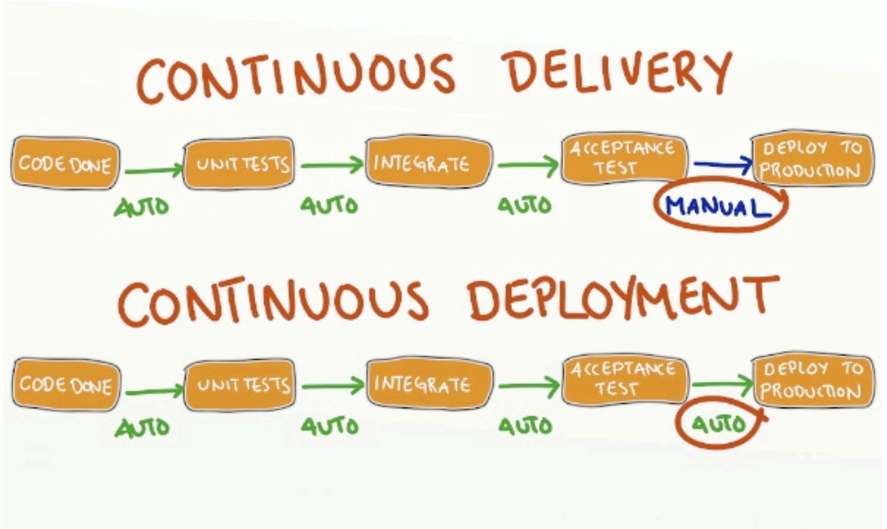
Continuous Delivery and Continuous Deployment are BOTH implementations of this practice.

- **Deployments are traditionally scary, risky endeavours.**
  - Like anything else painful or scary, do it a lot so it becomes routine.
  - Technically this is usually a solved problem. Don't reinvent the wheel!
  - Figuring out the process is often more difficult than actually getting the bits onto a server.
- **Continuous Integration REQUIRES multiple environments.**
  - You deploy the code to every developer's machine.
  - You deploy the code to the CI environment (GitHub Actions, Gitlab, etc.)
  - You deploy the code to a Demo or Stage environment.
  - You would have **already** automated the deploy. Sorry, I know anti-climactic.

Photo Credit: eelias, Unsplash

Release Engineering:
Risk Management for CD

Image Credit: Sanjay Zalavadia (SD Times)



This is the realm of Release Engineering (Relang). This is a data driven discipline, where metrics drive decision making, and optimization.

There are a ton of resources out there for getting background information on defining a release process.

- From a high level though you need to determine:
    - how releases are marked (maybe its release branches or tags?)
    - should we do continuous integration or continuous deployment
    - which technical tools suit our needs?
    - how are you going to measure success or rollback?

Continuous Delivery vs. Continuous Deployment:

- **Continuous Delivery- humans included:**
    - “Go/No Go” Meetings
    - Downtime is required. We need to inform users ahead of time and schedule a maintenance window.
- **Continuous Deployment- humans not needed here:**
    - Popular Strategies involve adding an additional layer of risk management to your Deployment Automation, and creating early, reversible feedback:
        - Canary Deploys: deploy to a subset of users, see how it goes. Infrastructure based, requires Ops personnel to manage.
        - Feature Toggles: hide the new features, and enable them post release for a subset of users. Software based, can be managed by the development team.

Bonus #1:

Pipelines as Code 1/2

- **YAML instead of scripts:**
  - PAC emerged as a way to create customized pipeline definitions, that goes beyond just BASH scripting, to controlling sequence, concurrency, notifications, and more.
  - A definition file in ‘YAML’ or ‘Groovy’ is checked into your codebase alongside your code, and your pipeline definition now lives in source control.
  - This config file controls your code base’s integration with a third party CI/CD as a service vendor. Think Circle CI, GitHub, GitLab, Jenkins… well basically all of them at this point.
- **Key Concepts:**
  - **Multiple Implementations.** Each vendor has slightly different offerings and syntax, although conceptually they do the same things. There is a bit of a learning curve.
  - Shell scripting (BASH) is still very prominent. Having competence in BASH will go a long way in helping you work with these config files.
  - **Environment Variables.** Remember credentials belong in the environment. You should not be checking sensitive material into your config file- rather consult your pipeline vendor’s documentation for how to manage secrets correctly.

Photo Credit: Danil Sorokin, Unsplash

Bonus #1:

Pipelines as Code 2/2

- **Off the shelf parts for your pipeline:**
  - A recent addition to PAC, is the appearance of reusable “modules”. These are refactored pieces of “pipe” that fill a common pipeline need.
  - BASH is notoriously hard to test. When possible, it’s nice to be able to use an off the shelf generic “Module” to do the task you need done, that is already tested.
  - Common Implementations: Jenkins: Groovy, GitHub Workflows: Actions, Circle CI: Orb, GitLab: Via Containers, Bitbucket: Pipes.
  - Your “PAC” config file slowly transforms into a series of calls to third party modules. BASH moves into a supporting role, rather than the primary implementation.
  - Look for a module before you sit down to write the code yourself. But, wear your critical hat when evaluating the quality of community modules. Don’t be afraid to BASH it out yourself.

Photo Credit: The Blowup Unsplash

Bonus #2:

Continuous Security

- **Times have changed:**
  - We are equipped with more automated tools today that relate to software security, than when Continuous Integration was first conceived.
  - It's now commonplace to have dire existential consequences to your organization if it were to be compromised.
    - Personally identifiable information (PII) being leaked.
    - Financial Implications.
- **Continuous Security - The Intersection of Security and CI**
  - Scanning dependencies on every commit against continually updated databases of vulnerabilities. (Think: npm audit)
  - Scanning for leaked credentials on every commit. (Gitleaks, CredScan)
  - Other relevant static analysis tools. (Bandit for Python.)
  - Your builds now fail when there is a security threat present, requiring the team to take action.

Photo Credit: Milan Malkomes, Unsplash

Thanks for listening.

- Niall Byrne: niall@niallbyrne.ca

Sources

- The Agile Manifesto
- Martin Fowler's Article on CI
- Kent Beck's TDD by Example
- The Twelve Factor App
- The Pragmatic Programmer
- The Google SRE Book

Photo Credit: Alexas Fotos- Unsplash