Software Testing for Continuous Delivery

Seminar 7: Continuous Integration

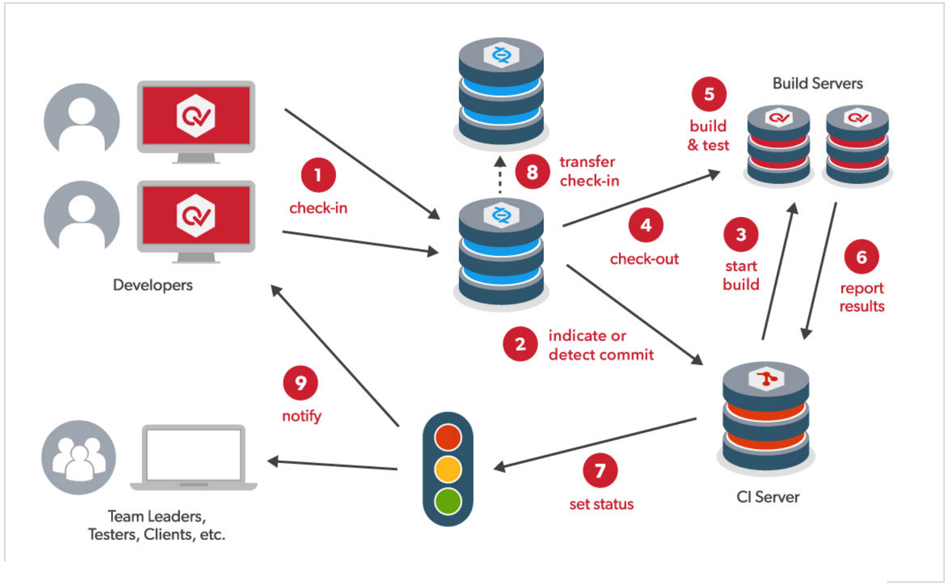
Dr. Byron J. Williams

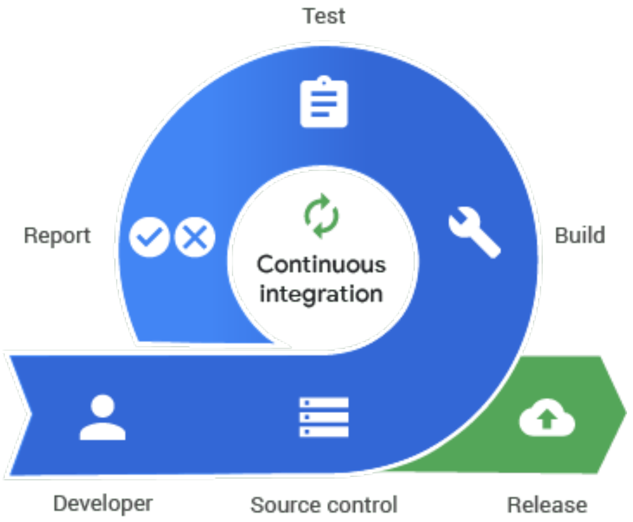
September 16, 2019

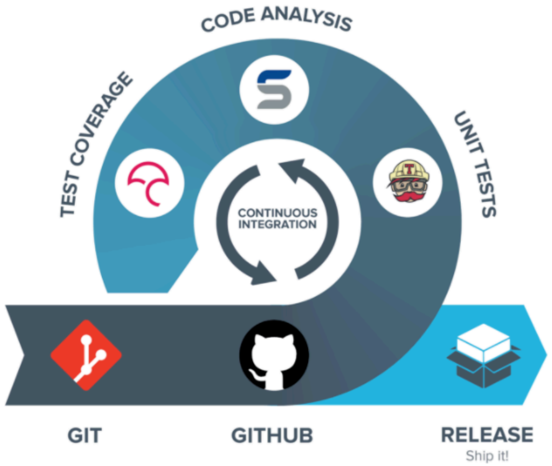
UF Herbert Wertheim College of Engineering
Department of Computer & Information Science & Engineering
UNIVERSITY *of* FLORIDA

Content adapted from “Continuous Delivery” by Jez Humble and David Farley 2010
Images courtesy Google Image Search

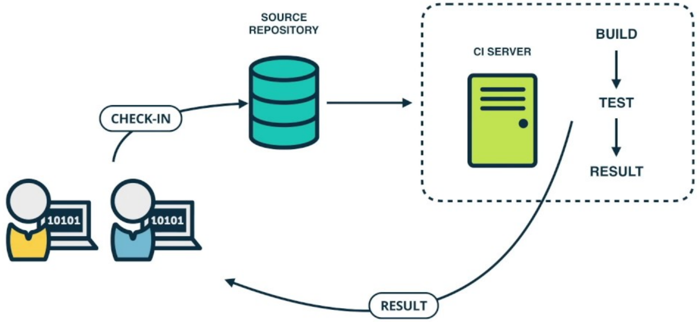
FICS Research







Continuous Integration (CI)



Software Development Value Proposition

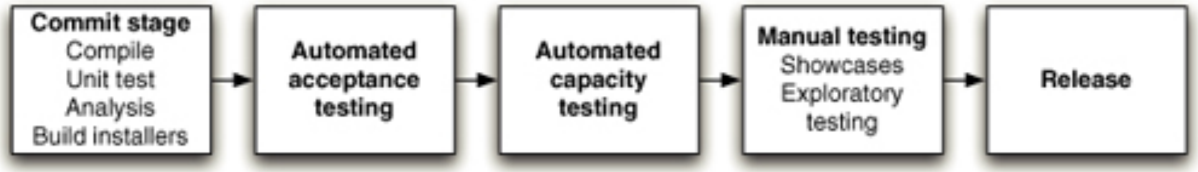
- The most important problem that we face as software professionals is this: If somebody thinks of a good idea, how do we deliver it to users as quickly as possible?

Problem

- For long periods of time during the development process the application is not in a working state
  - software developed by large teams spends a significant proportion of its development time in an unusable state
- Why?
  - Nobody is interested in trying to run the whole application until it is finished.
  - Developers check in changes and might even run automated unit tests, but nobody is trying to actually start the application and use it in a production-like environment.

The Deployment Pipeline

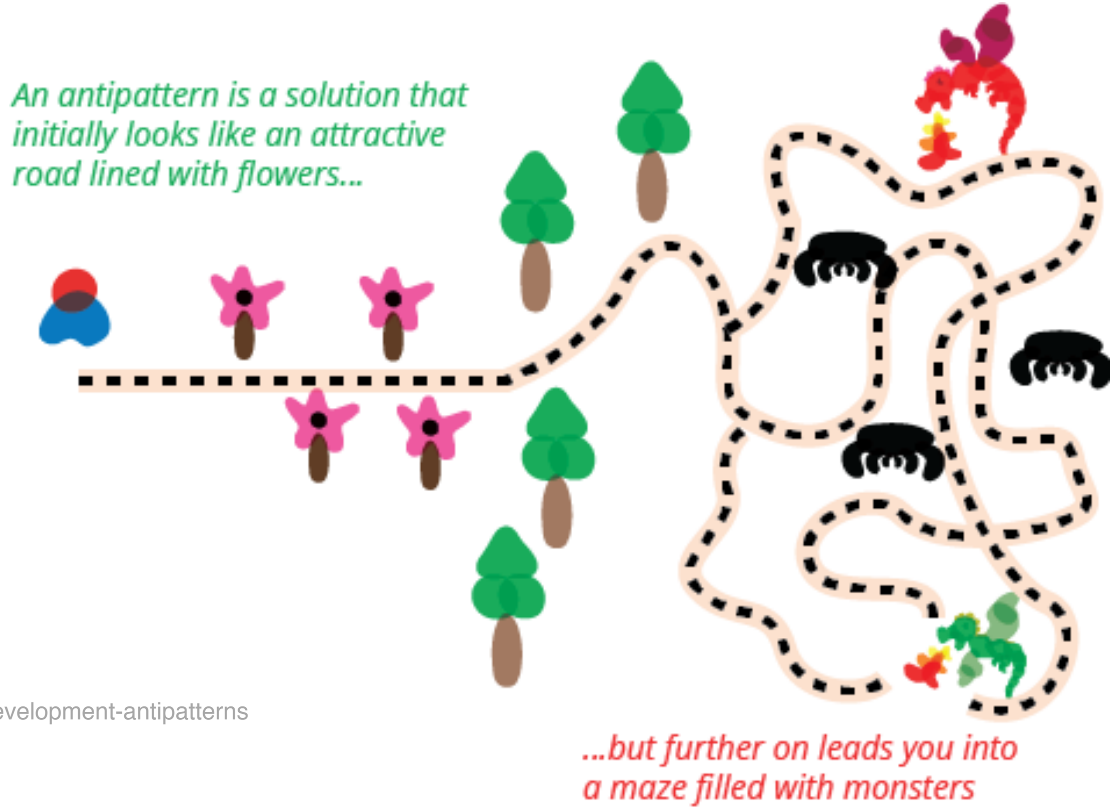
- What happens once requirements are identified, solutions designed, developed, and tested?
- How are these activities joined together and coordinated to make the process as efficient and reliable as we can make it?
- How do we enable developers, testers, build and operations personnel to work together effectively?



A **deployment pipeline** is, in essence, an automated implementation of your application's build, deploy, test, and release process

Antipatterns

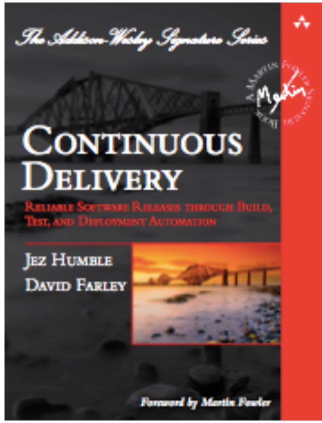
- A **design pattern** is a general repeatable solution to a commonly occurring problem in software design
- Antipatterns, like their design pattern counterparts, define an industry vocabulary for the **common defective processes** and implementations within organizations.
- An antipattern is just like a pattern, except that instead of a solution it gives something that looks superficially like a solution but isn't one. — *Andrew Koenig*


An antipattern is a solution that initially looks like an attractive road lined with flowers...
...but further on leads you into a maze filled with monsters

https://sourcemaking.com/antipatterns/software-development-antipatterns
https://martinfowler.com/bliki/AntiPattern.html

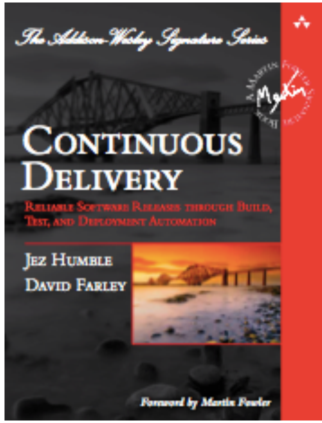
Antipattern - Deploying Software Manually

- The production of extensive, detailed documentation that describes the steps to be taken and the ways in which the steps may go wrong
- Reliance on manual testing to confirm that the application is running correctly
- Frequent calls to the development team to explain why a deployment is going wrong on a release day
- Frequent corrections to the release process during the course of a release
- Environments in a cluster that differ in their configuration, for example application servers with different connection pool settings, filesystems with different layouts, etc.
- Releases that take more than a few minutes to perform
- Unpredictable releases, that often have to be rolled back or run into unforeseen problems (or, late night debugging after release is pushed with errors)


The Addison-Wesley Signature Series
Continuous Delivery
Reliable Software Releases through Build, Test, and Deployment Automation
Jez Humble
David Farley
Foreword by Martin Fowler

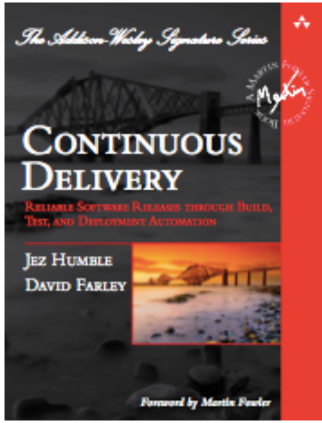
Antipattern - Deploying Software Manually

- Instead…
  - **over time**, deployments should tend towards being **fully automated**
  - When the deployment process is not automated, it is not repeatable or reliable
  - The set of automated deployment scripts serves as documentation
  - Automated deployments encourage collaboration, because everything is explicit in a script
  - Only an automated process is fully auditable
  - The automated deployment process must be used by everybody, and it should be the only way in which the software is ever deployed
  - Use the same script to deploy to every environment


The Addison-Wesley Signature Series
Continuous Delivery
Reliable Software Releases through Build, Test, and Deployment Automation
Jez Humble
David Farley
Foreword by Martin Fowler

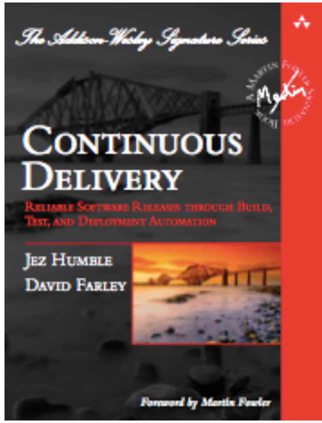
Antipattern - Deploying to a Production-like Environment Only after Development Is Complete

- The first time the software is deployed to a production-like environment (e.g., staging) is when most of the development work is "done"
- Incorrect assumptions about the production environment can be baked into the design of the system
- Instead…
  - The remedy is to **integrate** the testing, deployment, and release activities into the development process. Make them a normal and ongoing part of development so that by the time you are ready to release your system into production there is little to no risk, because you have **rehearsed it** on many different occasions in a **progressively more production-like** sequence of test environments


The Addison-Wesley Signature Series
Continuous Delivery
Reliable Software Releases through Build, Test, and Deployment Automation
Jez Humble
David Farley
Foreword by Martin Fowler

Antipattern - Manual Configuration Management of Production Environments

- Many organizations manage the configuration of their production environments through a team of operations people
  - deployed successfully to staging, production deployment fails
  - operations team take a long time to prepare an environment for a release
  - cannot step back to an earlier configuration of your system, which may include operating system, application server, web server, RDBMS, or other infrastructure settings
  - Servers in clusters have, unintentionally, different versions of operating systems, third-party infrastructure, libraries, or patch levels

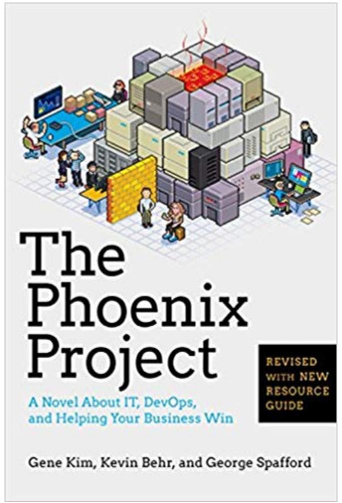

The Addison-Wesley Signature Series
Continuous Delivery
Reliable Software Releases through Build, Test, and Deployment Automation
Jez Humble
David Farley
Foreword by Martin Fowler

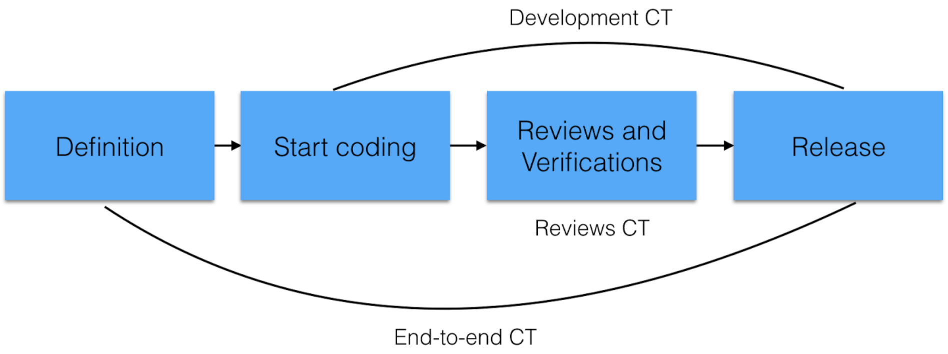
Antipattern - Manual Configuration Management of Production Environments

- **Instead**…
- All aspects of each of your testing, staging, and production environments, specifically the configuration of any third-party elements of your system, should be **applied from version control** through an automated process
- ***Configuration Management***: being able to repeatably re-create every piece of infrastructure used by your application (e.g., operating systems, patch levels, OS configuration, your application stack, its configuration, infrastructure configuration)
- You should know exactly what is in production. That means that every change made to production should be recorded and auditable.
- Indeed it should not be possible to make manual changes to testing, staging, and production environments

Cycle Time

- Software release can—and should—be a low-risk, frequent, cheap, rapid, and predictable process
- **Cycle time** - the time it takes from deciding to make a change, whether a bugfix, or a feature, to having it available to users


The Phoenix Project
A Novel About IT, DevOps, and Helping Your Business Win
REVISED WITH NEW RESOURCE GUIDE
Gene Kim, Kevin Behr, and George Spafford

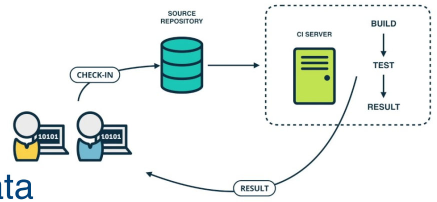


we want to find ways to deliver high-quality, valuable software in an efficient, fast, and reliable manner —> **frequent, automated releases**

https://www.klipfolio.com/blog/cycle-time-software-development

Feedback

Continuous Integration (CI)



- Every change should trigger the feedback processes
  - 4 application components: executable code, configuration, host environment, data
  - All 4 under version control - verify changes and test whole system
- Checks
  - creating the executable code - verifies that the syntax of your source code is valid
  - unit tests must pass - application’s code behaves as expected
  - fulfill certain quality criteria (e.g., test coverage) and other technology-specific metrics
  - functional acceptance tests must pass - conforms to its acceptance criteria— the business value intended
  - nonfunctional tests must pass - performs sufficiently well in terms of capacity, availability, security, etc.
  - must go through exploratory testing and a demonstration to the customer and a selection of users - manual testing environment

Herbert Wertheim College of Engineering
UNIVERSITY *of* FLORIDA

POWERING THE NEW ENGINEER TO TRANSFORM THE FUTURE

DEPARTMENT OF COMPUTER & INFORMATION SCIENCE & ENGINEERING
FLORIDA INSTITUTE FOR CYBERSECURITY RESEARCH (FICS)
Dr. Byron J. Williams

Continuous Integration

Continuous Integration (CI)

- A software engineering **process / practice** where isolated **code changes are immediately tested and reported** on as they are added to a larger codebase
- Developers incorporate their progress and code changes to the codebase **frequently**
- The goal is to **provide rapid feedback** to identify and **correct defects** as soon as they are introduced
- **Enables automation**
- Requires dedicated tools and automated tests to be written for the system *(unit —> end-to-end acceptance tests)*

RECAP Updated

Continuous Integration

- Agile methods work best when the current version of the software can be **run against all tests at any time**

A ***continuous integration server*** rebuilds the system, returns, and reverifies tests whenever *any* update is checked into the repository

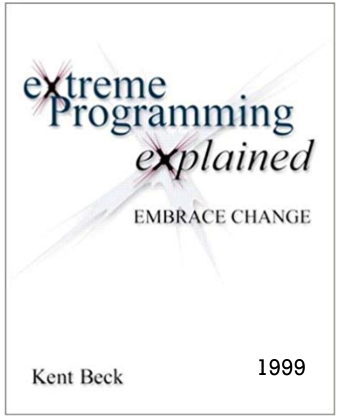
- Mistakes are caught earlier
- Other developers are aware of changes early
- The rebuild and reverify must happen as soon as possible
  - Thus, tests need to execute quickly

A ***continuous integration server*** doesn't just run tests, it decides if a modified system is still correct

RECAP

Continuous Integration - Requirements

- Continuous integration requires that every time somebody commits any change, the entire application is built and a comprehensive set of automated tests is run against it.
- if the build or test process fails, the development team stops whatever they are doing and fixes the problem immediately
- **GOAL**: software is in a working state all the time



Implementing CI - Requirements

1. Version Control

- Everything in your project must be checked in to a single version control repository: code, tests, database scripts, build and deployment scripts, and anything else needed to create, install, run, and test your application

2. An Automated Build

- You must be able to start your build from the **command line**

3. Agreement of the Team

- Continuous integration is a **practice**, not just a tool - it requires a degree of commitment and discipline from your development team

Using your CI Server (for the first time)

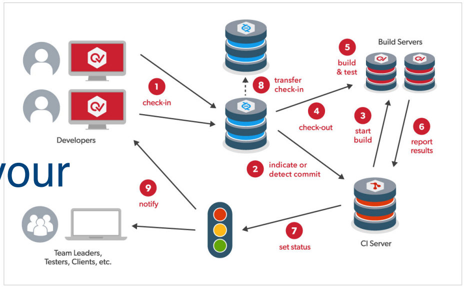
- You are likely to discover that the box you're running your CI tool on is missing a stack of software and settings
- Make a note of everything that you did to get things working, and put it on your project's wiki
- Check any software or settings that your system depends on into version control and **automate** the process of **provisioning** a new box

Jenkins

Travis CI

Using your CI Server (first build + general steps)

1. Check to see if a build is already running. If so, wait for it to finish. If it fails, you'll need to work with the rest of the team to make it green before you check in.
2. Update the code in your development environment from this version in the version control repository to get any updates
3. Run the build script and tests on your development machine to make sure that everything still works correctly on your computer
4. If your local build passes, check your code into version control.
5. Wait for your CI tool to run the build with your changes
6. If it fails, stop what you're doing and fix the problem immediately on your development machine—go to step 3
7. If the build passes, rejoice and move on to your next task



CI Essential Practices

- **Don’t check on a broken build**
  - check-in that breaks the build - to have the best chance of fixing it, need a clear run at the problem - not others checking in further changes, triggering new builds, and compounding the failure with more problems / conflict
- **Always Run All Commit Tests Locally before Committing**
  - want check-ins to be lightweight enough to check in regularly (e.g., every 20 mins) but also formal enough to briefly pause to think before committing
- **Wait for Commit Tests to Pass before Moving On**
  - build breakages are a normal and expected part of the process - until check-in has compiled and passed its commit tests, the developers should not start any new task

CI Essential Practices

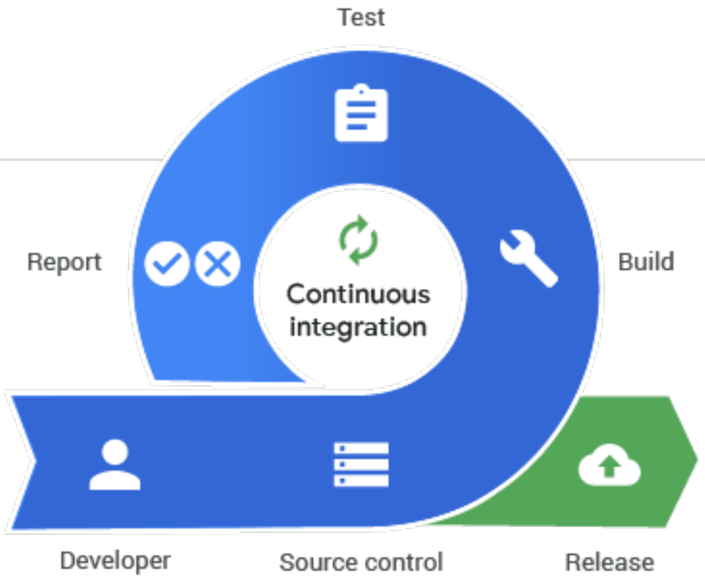
- **Never Go Home on a Broken Build**
  - revert your changes and return to your check-in attempt next day / week - after, your memory of the changes you made will no longer be fresh - give yourself time
- **Take Responsibility for All Breakages That Result from Your Changes**
  - If you commit a change and all the tests you wrote pass, but others break, the build is still broken. Usually this means that you have introduced a regression bug into the application
- **Always Be Prepared to Revert to the Previous Revision**
  - we expect that everyone will break the build from time to time - either can't find where the problem lies, or just after the check-in fails we realize that we missed something important about the nature of the change that we have just made

CI Essential Practices

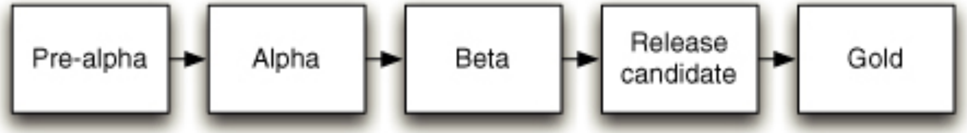
- **Time-Box Fixing before Reverting**
  - When the build breaks on check-in, try to fix it for ~10 minutes - no fix, revert to the previous version
- **Don't Comment Out Failing Tests** (last resort)
  - it is essential to put in the work to find out what is going on and either fix the code (if a regression has been found), modify the test (if one of the assumptions has changed), or delete it (if the functionality under test no longer exists)
- **Test-Driven Development (BDD, ATDD)**
  - having a comprehensive test suite is essential to continuous integration - the fast feedback, which is the core outcome of continuous integration, is only possible with excellent unit test coverage - TDD (in some form) is essential to CI/CD success

The Release Candidate

- Every change must be evaluated for its fitness
- If the resulting product is found to be free of defects, and it meets the acceptance criteria set out by the customer, then it can be released
- Every change that a developer makes to a codebase is intended to add value in some manner
- Every change committed to version control is supposed to enhance the system that we are working on
- The software is in fact always in a releasable state.

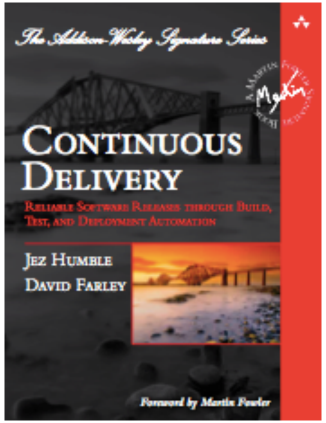


“traditional” view of release candidates



Principles of Software Delivery

- Create a Repeatable, Reliable Process for Releasing Software
- Automate Everything (almost)
- Keep Everything in Version Control
- If It Hurts, Do It More Frequently, and Bring the Pain Forward
- Build Quality In
- Done Means Released
- Everybody Is Responsible for the Delivery Process
- Focus on Continuous Improvement


The Addison-Wesley Signature Series
Continuous Delivery
Reliable Software Releases through Build, Test, and Deployment Automation
Jez Humble
David Farley
Foreword by Martin Fowler

Git & GitHub

- **Github Flow**
  - Process to be followed: https://guides.github.com/introduction/flow/
  - Code owner will create project and allow forks
  - Team will decide on names of functions that will be developed as feature branches of the overall project
- **Git understanding is important for CI workflow**
  - GitKraken - Gui Tool https://www.gitkraken.com/
  - GitFlow Atlassian Tutorial
    - https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow
  - Pull Requests - https://www.atlassian.com/git/tutorials/making-a-pull-request