Configuration Management for Continuous Delivery Pipelines

Continuous Integration

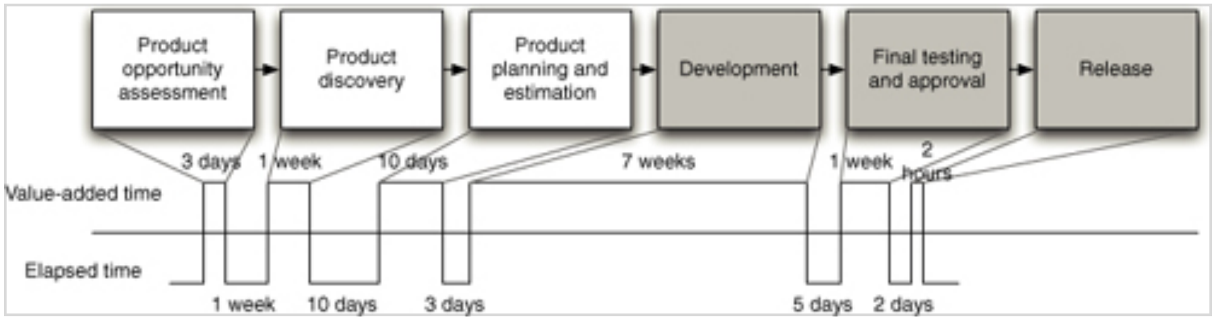
CI ensures that the code that we create, as a team, works by providing us with rapid feedback on any problems that we may introduce with the changes we commit. It is primarily focused on asserting that the code compiles successfully and passes a body of unit and acceptance tests

Much of the **waste** in releasing software comes from the progress of software through testing and operations. For example, it is common to see

- Build and operations teams waiting for documentation or fixes
- Testers waiting for "good" builds of the software
- Development teams receiving bug reports weeks after the team has moved on to new functionality
- Discovering, towards the end of the development process, that the application's architecture will not support the system's nonfunctional requirements

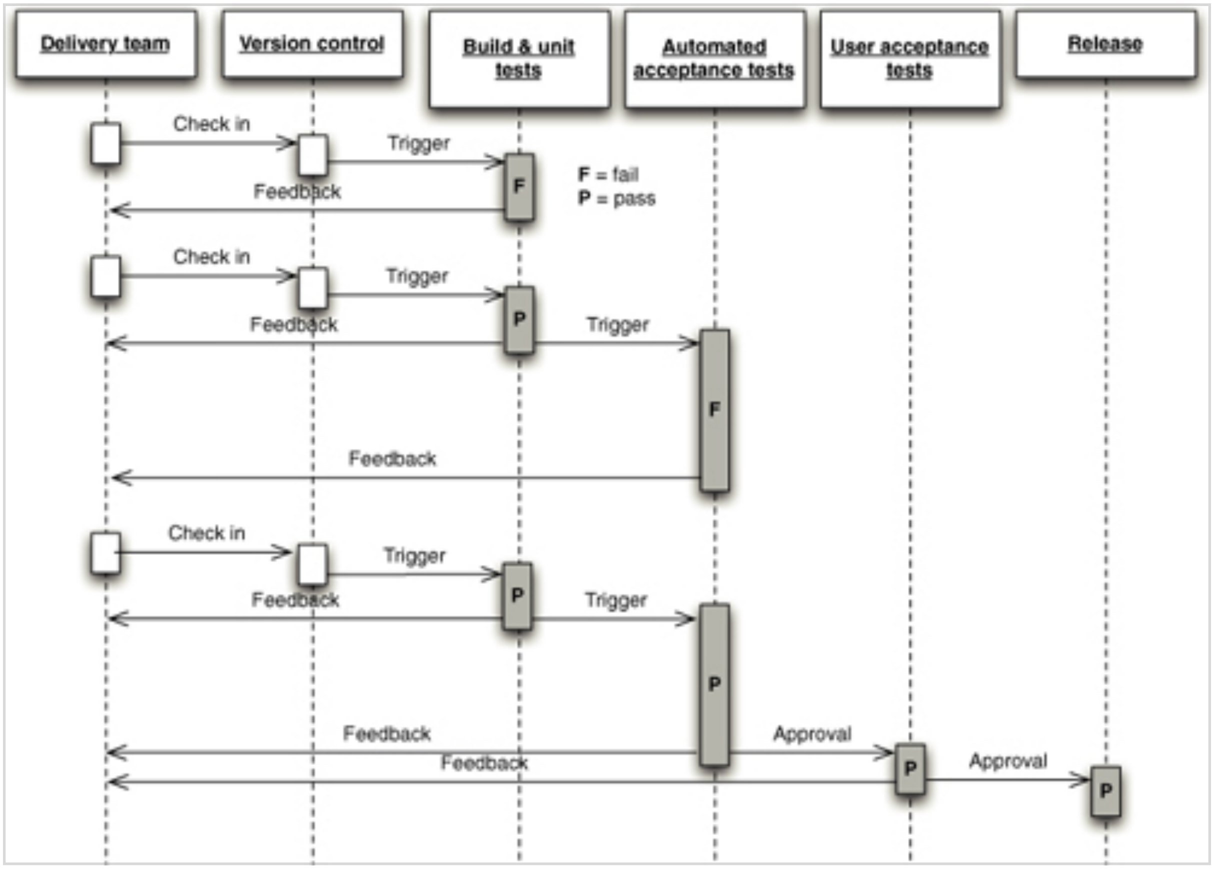
What we end up with is (in lean parlance) a *pull system*. Testing teams deploy builds into testing environments themselves, at the push of a button. Operations can deploy builds into staging and production environments at the push of a button. Developers can see which builds have been through which stages in the release process, and what problems were found.

What is a Deployment Pipeline?

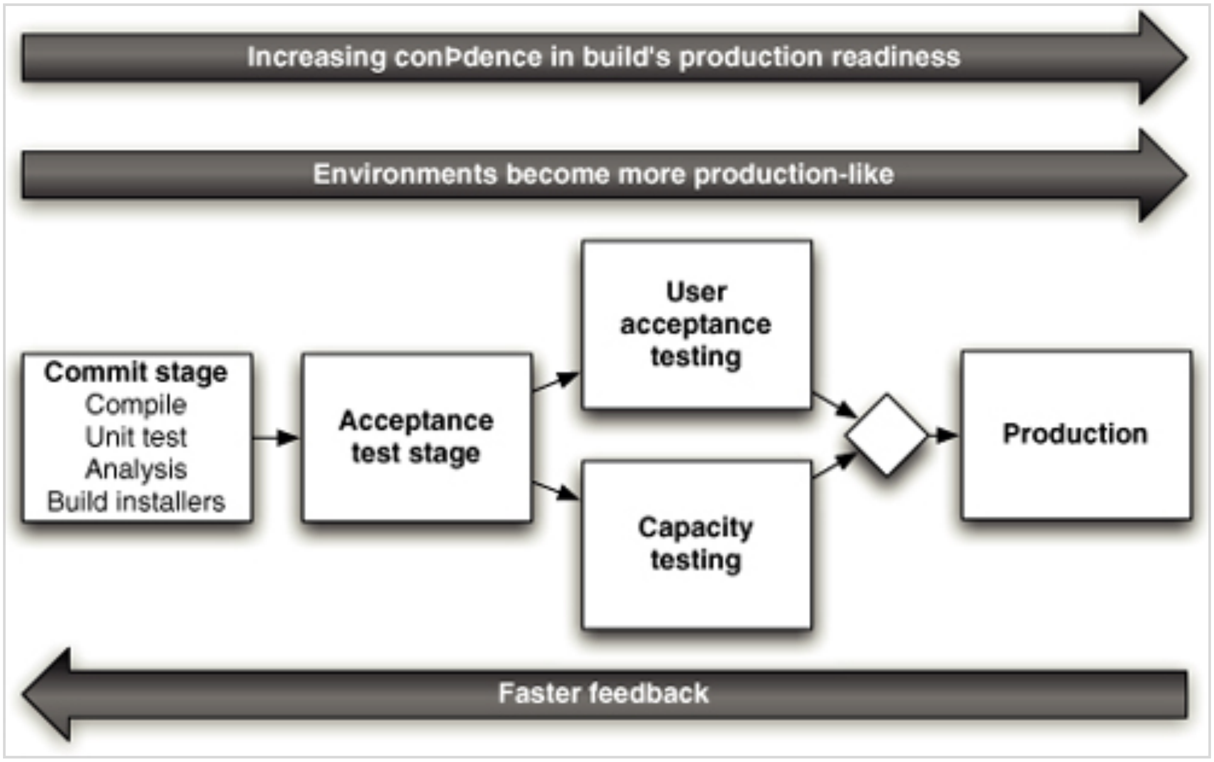


A simple value stream map for a product

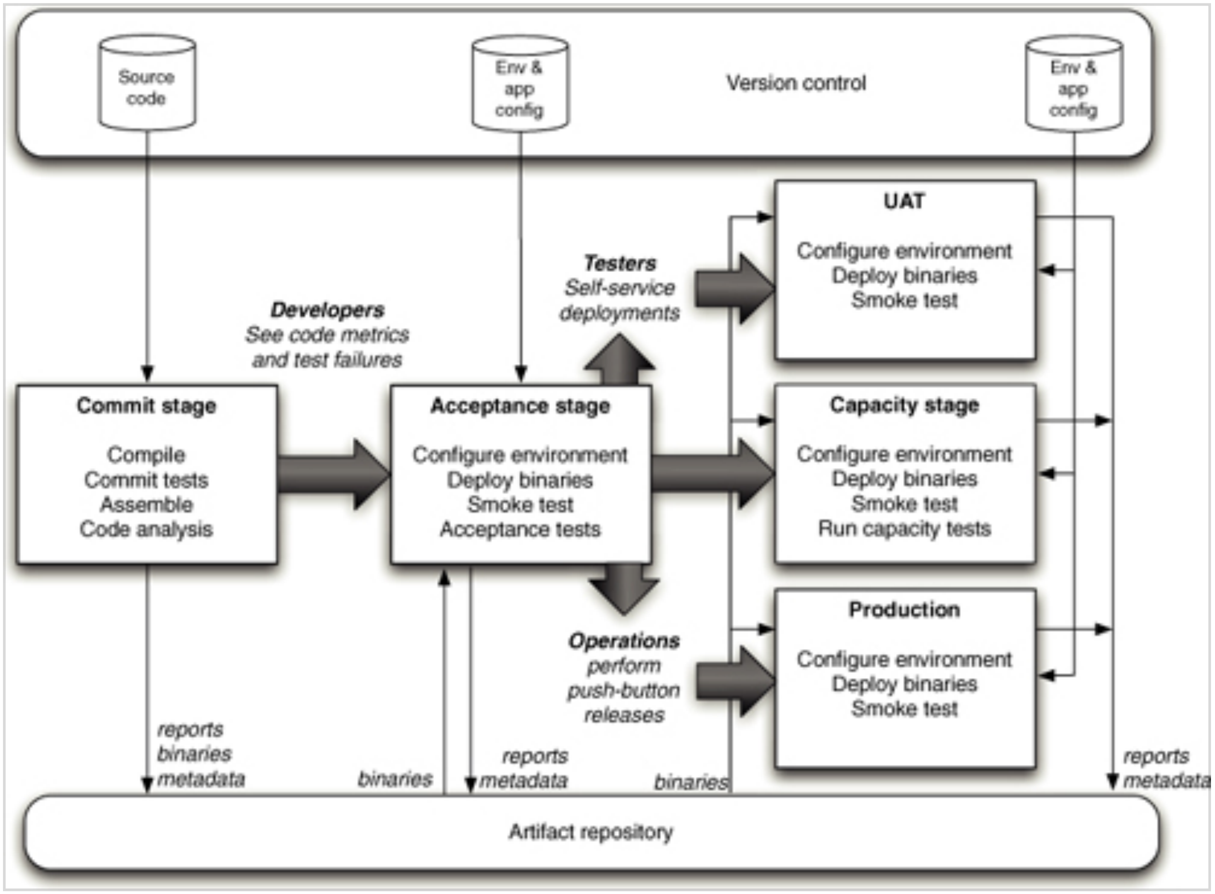
Thus the process modeled by the deployment pipeline, the process of getting software from check-in to release, forms a part of the process of getting a feature from the mind of a customer or user into their hands



Changes Moving through a deployment pipeline



Trade-offs in the deployment pipeline



Basic deployment pipeline

Deployment Pipeline Practices

Only Build your Binaries Once

- you should only build your binaries once, during the commit stage of the build - these binaries should be stored on a filesystem somewhere (not in version control, since they are derivatives of your baseline, not part of its definition) where it is easy to retrieve them for later stages in the pipeline - CI servers will handle this for you
- we will refer to the collections of executable code as binaries, although if you don't need to compile your code these "binaries" may be just collections of source files (e.g., minified Javascript bundles)
- Every time you compile the code, you run the risk of introducing some difference. The version of the compiler installed in the later stages may be different from the version that you used for your commit tests. You may pick up a different version of some third-party library that you didn't intend.
- Objective: keep the deployment pipeline efficient
- If we re-create binaries, we run the risk that some change will be introduced between the creation of the binaries and their release, such as a change in the toolchain between compilations, and that the binary we release will be different from the one we tested

Deploy the Same Way to Every Environment

- it is essential to use the same process to deploy to every environment—whether a developer or analyst's workstation, a testing environment, or production—in order to ensure that the build and deployment process is tested effectively
- Developers deploy all the time; testers and analysts, less often; and usually, you will deploy to production fairly infrequently. But this frequency of deployment is the inverse of the risk associated with each environment. The environment you deploy to least frequently (production) is the most important.
- **Premise:** Every environment is different in some way. If nothing else, it will have a unique IP address, but often there are other differences: operating system and middleware configuration settings, the location of databases and external services, and other configuration information that needs to be set at deployment time

Smoke-Test Your Deployments

- When you deploy your application, you should have an automated script that does a smoke test to make sure that it is up and running. This could be as simple as launching the application and checking to make sure that the main screen comes up with the expected content.

Deploy into a Copy of Production

- you need to do your testing and continuous integration on environments that are as similar as possible to your production environment —> Docker or other containers / VMs —> Puppet, Ansible and other config management tools
- You need to ensure that:
    - Your infrastructure, such as network topology and firewall configuration, is the same.
    - Your operating system configuration, including patches, is the same.
    - Your application stack is the same.
    - Your application's data is in a known, valid state.

If Any Part of the Pipeline Fails, Stop the Line

- The most important step in achieving the goals of this book—rapid, repeatable, reliable releases—is for your team to accept that every time they check code into version control, it will successfully build and pass every test. This applies to the entire deployment pipeline. If a deployment to an environment fails, the whole team owns that failure

Configuration Management

Configuration management is a term that is widely used, often as a synonym for version control - *Configuration management* refers to the process by which all artifacts relevant to your project, and the relationships between them, are stored, retrieved, uniquely identified, and modified.

If you have a good configuration management strategy, you should be able to answer “yes” to all of the following questions:

- Can I exactly reproduce any of my environments, including the version of the operating system, its patch level, the network configuration, the software stack, the applications deployed into it, and their configuration?
- Can I easily make an incremental change to any of these individual items and deploy the change to any, and all, of my environments?
- Can I easily see each change that occurred to a particular environment and trace it back to see exactly what the change was, who made it, and when they made it?
- Can I satisfy all of the compliance regulations that I am subject to?
- Is it easy for every member of the team to get the information they need, and to make the changes they need to make? Or does the strategy get in the way of efficient delivery, leading to increased cycle time and reduced feedback?

Config Mgmt for Cont Delivery Objectives:

1. getting everything into version control and managing dependencies
2. managing an application’s configuration
3. configuration management of whole environments—the software, hardware, and infrastructure that an application depends upon; focused on environment management, from operating systems to application servers, databases, and other commercial off-the-shelf (COTS) software

Version Control

a mechanism for keeping multiple versions of your files, so that when you modify a file you can still access the previous revisions

Two-fold Aim

1. it retains, and provides access to, every version of every file that has ever been stored in it
2. it allows teams that may be distributed across space and time to collaborate

Why would you want to do this?

To Determine:

- What constitutes a particular version of your software? How can you reproduce a particular state of the software's binaries and configuration that existed in the production environment?
- What was done when, by whom, and for what reason? Not only is this useful to know when things go wrong, but it also tells the story of your application.

Practices:

- **Keep Absolutely Everything in Version Control (typically except binary files)**
  - every single artifact related to the creation of your software should be under version control - source code, tests, database scripts, build and deployment scripts, documentation, libraries and configuration files for your application, your compiler and collection of tools, and so on
  - you need *everything* required to re-create your application's binaries and the environments in which they run
  - objective is to have everything that can possibly change at any point in the life of the project stored in a controlled manner
  - allows you to recover an exact snapshot of the state of the entire system, from development environment to production environment, at any point in the project's history
- **Check in regularly**
  - once you check your changes into version control, they become public, instantly available to everybody else on the team
  - you have just given birth to a build that could potentially end up in acceptance testing or even production
  - checking in is a form of publication, it is important to be sure that your work, whatever it may be, is ready for the level of publicity that a check-in implies - with caveats based on how the CI System is configured and whether you're checking into a feature branch etc.
- **Use meaningful commit messages**
  - most important reason to write descriptive commit messages is so that, when the build breaks, you know who broke the build and why

Managing Dependencies

The most common external dependencies within your application are the third-party libraries it uses and the relationships between components or modules under development by other teams within your organization

Managing External Libraries

external libraries are normally installed globally on your system by a package management system such as Ruby Gems, Python PIP, Go Get, npm install

recommend that you keep copies of your external libraries somewhere locally (e.g., node_modules)

your build system should always specify the exact version of the external libraries that you use (e.g., requirements.txt, package.json)

Managing Components

good practice to split all but the smallest applications into components

- limits scope of changes, reduces regression bugs, encourages reuse, & more efficient for large projects
- components that are widely used across projects can have their own CI setup and pipeline

Managing Environments

- Every application depends on hardware, software, infrastructure, and external systems in order to work i.e., your application's environment
- managing the environment that your application runs in is that the configuration of that environment is as important as the configuration of the application

Being able to reproduce your environments is essential for several reasons:

- It removes the problem of having random pieces of infrastructure around whose configuration is only understood by somebody who has left the organization and cannot be reached.
- Fixing one of your environments can take many hours. It is always better to be able to rebuild it in a predictable amount of time so as to get back to a known good state- can also roll back the environment
- It is essential to be able to create copies of production environments for testing purposes - testing environments should be exact replicas of the production ones, so configuration problems can be found early

Environment Data:

- operating systems in your environment, including their versions, patch levels, and configuration settings
- The additional software packages that need to be installed on each environment to support your application, including their versions and configuration
- The networking topology required for your application to work

- Any external services that your application depends upon, including their versions and configuration
- Any data or other state that is present in them (for example, production databases)

####### Footnote: Notes derived from Continuous Delivery" by Jez Humble and David Farley 2010 (see: https://amzn.to/2nOWHRO) #######