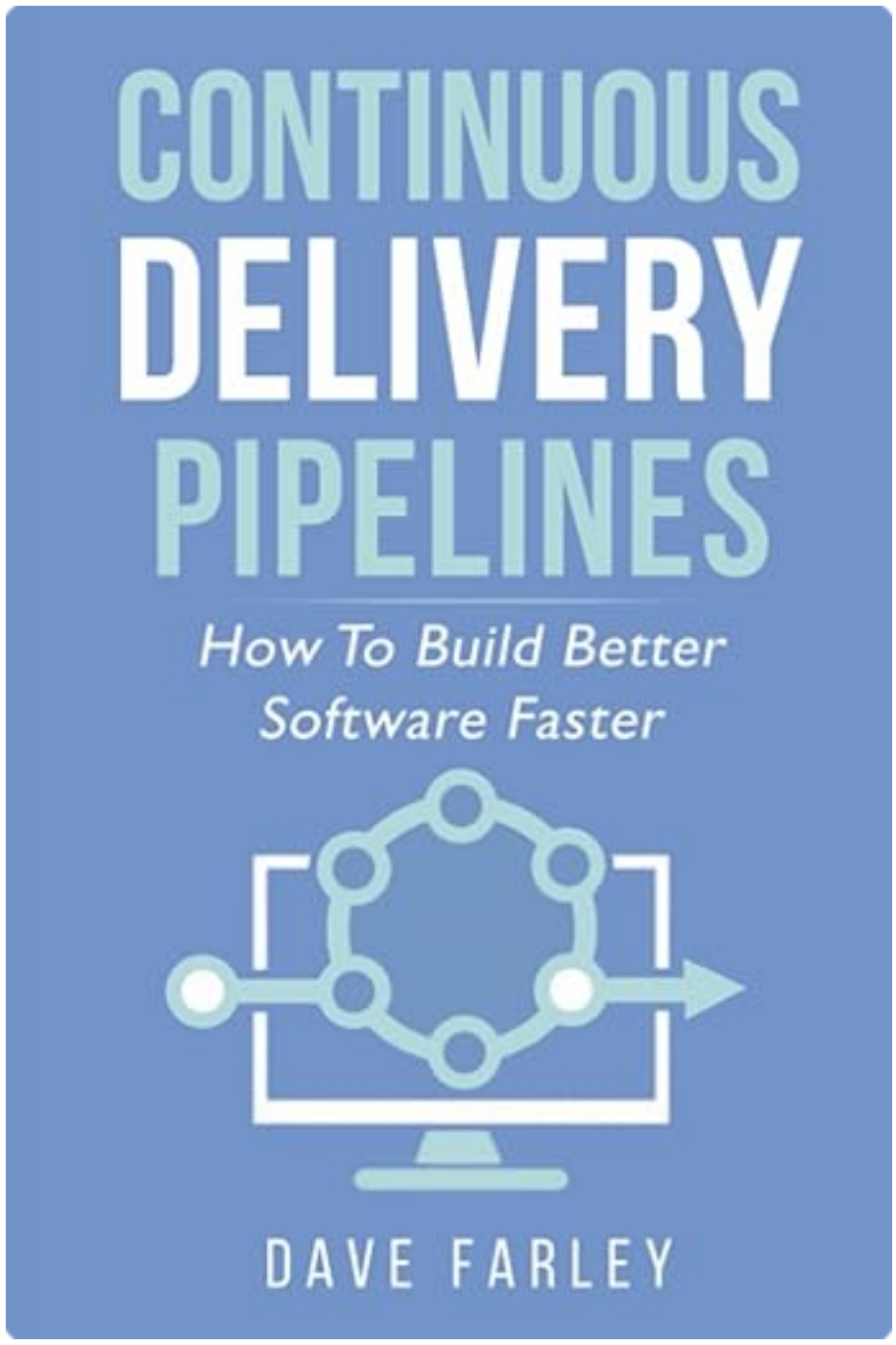
Continuous Delivery Pipelines



Dave Farley Style

CI/CD vs CD

So what is the difference between CI/CD and CD? I suspect in the beginning there wasn't a difference but over time and through semantic diffusion CI/CD in our industry has just become a buzz phrase for software building tools. I find it ironic that CI/CD explicitly calls out Continuous Integration but so far in my limited exposure, I have not found a team that claims to be doing CI/CD actually practicing Continuous Integration at all. I believe the dominance of Git in the industry has brought about a profound change. This is because Git made branching much easier than other VCS (Version Control Software) tools. I'm not bashing Git but I will bash the use of GitFlow for teams that are made up of trusted software professionals working inside software companies every chance I get. In my opinion GitFlow is a great branching strategy for what it was intended to support and that's open source projects where the contributors are not known and cannot be trusted. TBD (Trunk Based Development) is far better for software companies that want to build high quality software and deliver value to their customers faster.

So What Do You Mean, Dave Farley Style?

Mr. Farley and his coauthor Jez Humble first described Continuous Delivery in the 2011 award winning book Continuous Delivery. I highly recommend this book. Things move fast in software development so you will find some information about HOW to implement will be a little dated but the more important part of WHY is just as relevant now as it was when the book was first published. You will learn that CD is the natural extension of CI and therefore CD requires CI.

The book cover I have on the first page, Continuous Delivery Pipelines is a newer book that is all about concisely describing how to build a Continuous Delivery Pipeline. It doesn't show you how to implement it with examples using particular tools like GitHub Actions, GitLab, Circle CI etc. Instead you will learn the anatomy of a CD Pipeline. I also highly recommend this book.

In this repository I'm doing my best to implement a CD Pipeline that is true to the guidelines laid out in this book using GitHub Actions Workflows. It's a minimal implementation or what Mr. Farley would call "a walking skeleton example."

After I read Accelerate which lead me to reading these two books on CD I went looking for a simple example that implements a CD Pipeline as described by Mr. Farley and I could not find one. I did find this website Minimum Viable CD which is endorsed by Dave Farley and it is very helpful but at some point I want to see the code. Not able to find it, I'm creating it.

From the Continuous Delivery Pipelines book.

> *Initially we just need the four essential components of the Deployment Pipeline:*
>
> 1. *Commit Stage*
> 2. *Artifact Repository*
> 3. *Acceptance Sage, and*
> 4. *Ability to Deploy into Production*
>
> *and we start with a simple use-case.*

Commit Stage

In the Commit Stage of a Continuous Delivery Pipeline fast feedback is paramount.

This stage involves:

- Compiling the code
- Running unit tests
- Conducting static code analysis

- Storing the compiled code in the Artifact Repository

- Merging the code change into Trunk (Main Branch)

Continuous Integration (CI) is integral to this process, ensuring that code changes are regularly and automatically integrated into a shared repository. The unit tests, designed to run in under 5 minutes, play a crucial role by providing approximately 80% confidence that code changes are correct and align with expectations. If the unit tests and static code analysis pass, the compiled code is marked as a release candidate. To meticulously document the state of the codebase, the source files, tests, and other files used to create the binaries are tagged in the code repository. This is achieved by applying a GitHub tag in the form “rc-<build_number>". where build_number is the number of times the CI workflow has been executed. This tag is incremental, resulting in a sequence such as rc-50, rc-51, rc-52, and so on. The compiled code is stored in the Artifact Repository to maintain consistency. This repository houses the release candidates and serves as a reliable source for deployment to test environments. The synergy of CI and CD streamlines development workflows, allowing for early error detection and efficient deployment practices.

The Trunk in this repository is named Main. You will not see long lived branches with names like Master, Production, Staging, QA, and Development. You will only see other branches as short lived branches that once pass the checks in the commit stage will immediately be merged into the trunk (main branch) and then deleted.

Versions And Tagging

Semantic versioning is very common and we are using it in this project. However, when it comes to semantic versioning used in standalone applications the rules of semantic versioning don’t exactly apply. Semantic versioning is a scheme to avoid dependency hell. In the case of our simple application nothing depends on it. We are just using the version to indicate major changes verses smaller chances. We are also using the PATCH number to indicate the build_number. For example 1.1.356 indicates this is the 356th build of the software. This allows us to use the MAJOR version and MINOR version to describe features but unlike strict semantic versioning the PATCH is never reset to zero in our scheme. This is how we can quickly and easily correlate source code to the compiled code. For example, if a bug were found in release 1.1.356 I know immediately that if I look at the code tagged with rc-356 I’m looking at the source code that created that release. If I have a RC fail in the Acceptance Stage e.g. rc-357 then I know to see the source code for that RC will be tagged rc-357. For changing the MAJOR.MINOR we update that in our package.json. The

PATCH as explained is really the build_number and that is automatically updated by the CD Pipeline. When a RC has passed the Commit Stage and we run the app it will have a version like 1.1.356-RC. The -RC will not be dropped until the RC passes the Acceptance Stage then it will become 1.1.356 in this example. Keep in mind not all versions/releases will be deployed. It could be that 1.1.356 is never deployed.

Commit Stage Implementation

See the NodeJS Continuous Integration Workflow in the WoodyB/vending-machine-project.

Artifact Repository

GitHub has a built in Artifact Repository so it wasn't necessary to use a third party or roll our own. An Artifact Repository is different than a Source Code Repository. In simple terms it is an ephemeral storage area for items produced by processes like building the software and testing it. Items like log files, compiled code, and data that needs to persist from one GitHub Workflow to the next.

Acceptance Stage

Acceptance Stage Implementation

See the NodeJS Continuous Delivery Workflow in the WoodyB/vending-machine-project. This is where we test the RC in a production like environment. The tests running in the Acceptance Stage will be testing behavior from a customer's point of view.

Ability To Deploy Into Production

Deploy Into Production Implementation

See the Create Release Workflow in the WoodyB/vending-machine-project. Read more about deploying to production in the Developer Flow below.

Start With A Simple Use-Case

How about Hello World? At the time of writing the first version of this document that is all our application does. Well that and print the version. Bryan Finster said this in one of his 5-Minute DevOps articles. *The delivery pipeline is feature 0. Before I write a feature, I need a way to test and deliver that feature. So the first test is to test that I can ship "Hello world."* That's the commitment I made for this project.

Developer Flow

1. Sync up to main for latest changes

2. Create a local short lived branch e.g. `git checkout -b WoodyB/Fix-Bug-123`

3. Using TDD begin implementation of a bug fix or new feature developing in a way that will allow us to merge to main at least once a day if the fix or feature takes longer than a

day. Preferably the frequency will be more like every hour or half hour. There will be no PR Review/Approval. Instead, pair programming or mob programming will be used. Code Reviews can still be done after a successful merge to main if desired.

4. With all Lint Checks and Commit Stage tests passing locally on the dev machine push the branch to origin e.g. `git push origin WoodyB/Fix-Bug-123`

5. On GitHub create a PR for the branch to merge into main and submit it.

6. This will trigger the NodeJS Continuous Integration Workflow.

7. If all the checks pass (linter, unit tests and no merge conflicts) the changes will automatically be merged into main. This is what is known as Gated Continuous Integration. In plain Continuous Integration the merge would happen regardless of if the checks passed or failed. When a failure occurs in plain Continuous Integration then the Pipeline is blocked until the issue is fixed. The main tradeoffs are there's a small chance of merge conflicts in Gated CI vs small chance of blocking the CD pipeline for the whole team in plain CI. One of the jobs of the CI Stage is to compile/build the code. This only happens once in a CD Pipeline. This "binary" will be stored in the artifact repository where it will be used by later stages. In our project the "binaries" are JavaScript that has been transpiled from TypeScript. These will be stored as rc-<build_number> e.g. `rc-356` in the Artifact Repository. If we want to see the sources for `rc-365` then we simply switch our view in GitHub to tag rc-356. We can also pull that code down onto our local dev machine if we need to using the tag.

8. Go back to step 1 and repeat.

9. Since the Commit Stage should be able to finish in 5 mins or less you should monitor the results. If there's a problem address it immediately this includes flakey tests. Flakey tests should be fixed or removed with extreme prejudice. They are cancer to a CD Pipeline.

10. Every so often the NodeJS Continuous Delivery Workflow will wake up and look for the latest RC (Release Candidate.) When the NodeJS Continuous Integration Workflow finds the changes are acceptable (passes all checks) it merges those changes to the trunk (main branch.) This means that several merges could have happened since the last time the Continuous Delivery Workflow work up and ran. This is by design. In a fast paced CI

environment if every RC were fully tested our CD Pipeline would get backed up pretty fast. The latest RC will have all the changes. RC-123 will contain the changes from RC-122 and all other RCs before it. If there's a failure in the Acceptance Stage you may not know who's change broke the Acceptance tests so the responsibility falls to all those that committed those changes. It's extremely important to take failures in the Acceptance stage seriously. This means the developer(s) should check on the results of the Acceptance Stage results but since it may run for a while not stop work and wait for it to finish. The goal is to keep Acceptance Stage able to complete in one hour or less. Once a RC passes the Acceptance Stage it is viable to be deployed as a Release. The artifact will be renamed from rc-<build_number> to rc-<build_number>-accepted. e.g. `rc-356-accepted` in the Artifact Repository.

11. If the CD Pipeline is down it should be all hands on deck to fix it. If you've ever watched the reality TV show Gold Rush this is like when the Wash Plant is down. "If the wash plant is down you ain't making gold you're just moving dirt around." If our CD Pipeline is down then we aren't making software because our one way to production is blocked.

12. Any RC that has been accepted by the NodeJS Continuous Delivery Workflow can be deployed. In our simple case this means we upload our compressed binaries in a zip file to box.com where it can be downloaded. There's a workflow in place to perform this action. It is the Create Release Workflow and is manually triggered.