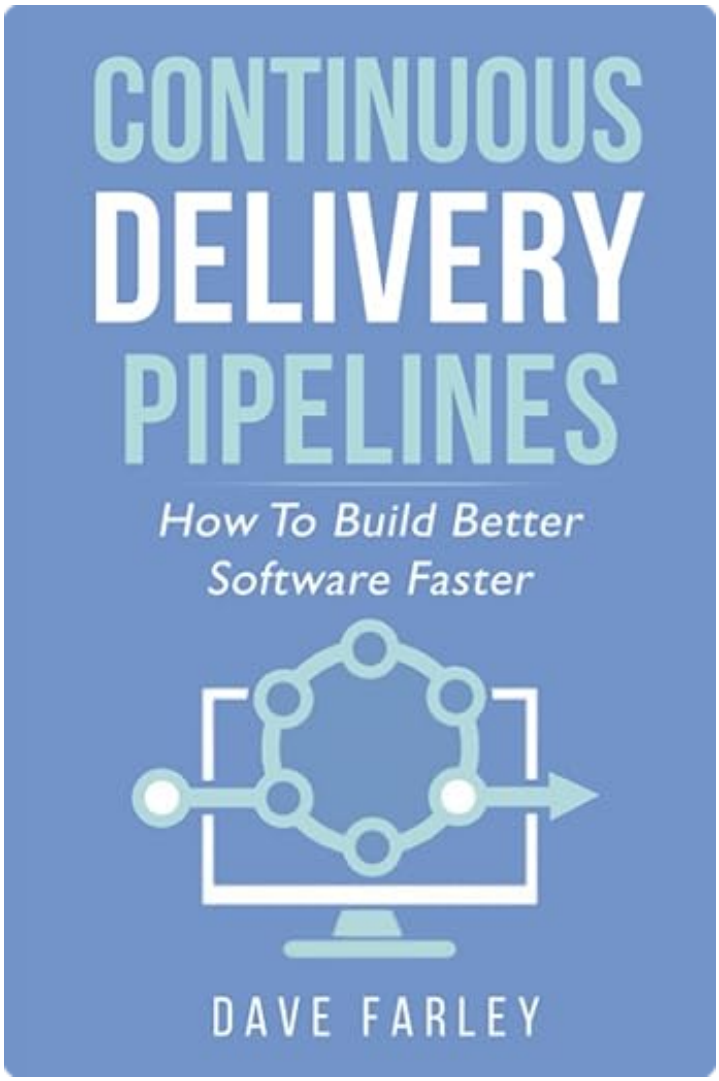
Continuous Delivery Pipelines



Dave Farley Style

Note: Hyperlinks are not active when viewed from GitHub's file viewer. Either navigate back to the main ReadMe for the project and click the link for this document in the Goals section or copy this link into your browser ***https://woodyb.github.io/vending-machine-project/design/CD-Deployment-Pipelines.pdf***

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 led 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 RC (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**. Besides the main branch 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 changes. 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.

Let's say a bug was found in release **1.1.356** just by looking at the release version I can see it was **build 356** so 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. **1.1.357-RC** then I know the source code for that RC will be tagged **rc-357**.

To change the **MAJOR.MINOR** of the version we update that by editing the package.json file as part of our PR. 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.

One of the jobs of the Commit Stage is to compile/build the code. This only happens once in a CD Pipeline. These "binaries" will be stored in the Artifact Repository where they 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>-binaries** e.g. **`rc-356-binaries`** in the Artifact Repository. As I have explained if we want to see the sources for **`rc-365-binaries`** 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. Since we are using shallow tags you will likely need to pull the tags down from the origin from time to time. The command is **`git fetch origin --tags –depth=1`**. Then do a **`git checkout <tag>`** e.g. **`git checkout rc-356.`**

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 can be stored in the Artifact Repository.

Acceptance Stage

Every so often the NodeJS Continuous Delivery Workflow will wake up and look for the latest RC (Release Candidate) e.g. **rc-356-binaries**.

Note: Several merges could have happened since the last time the NodeJS 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 could 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>-binaies** to **rc-<build_number>binaries–accepted**. e.g. **rc-356-binaries-accepted** in the Artifact Repository. If the RC fails in the Acceptance Stage it will be renamed to **rc-<build_number>binaries–rejected.**

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

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.

If the CD Pipeline is down it should be all hands on deck to fix it. If our CD Pipeline is down then we can't release software because our one way to production is blocked.

Deploy Into Production Implementation

See the Create Release Workflow in the WoodyB/vending-machine-project.

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 e.g. **`git checkout main`**

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`**

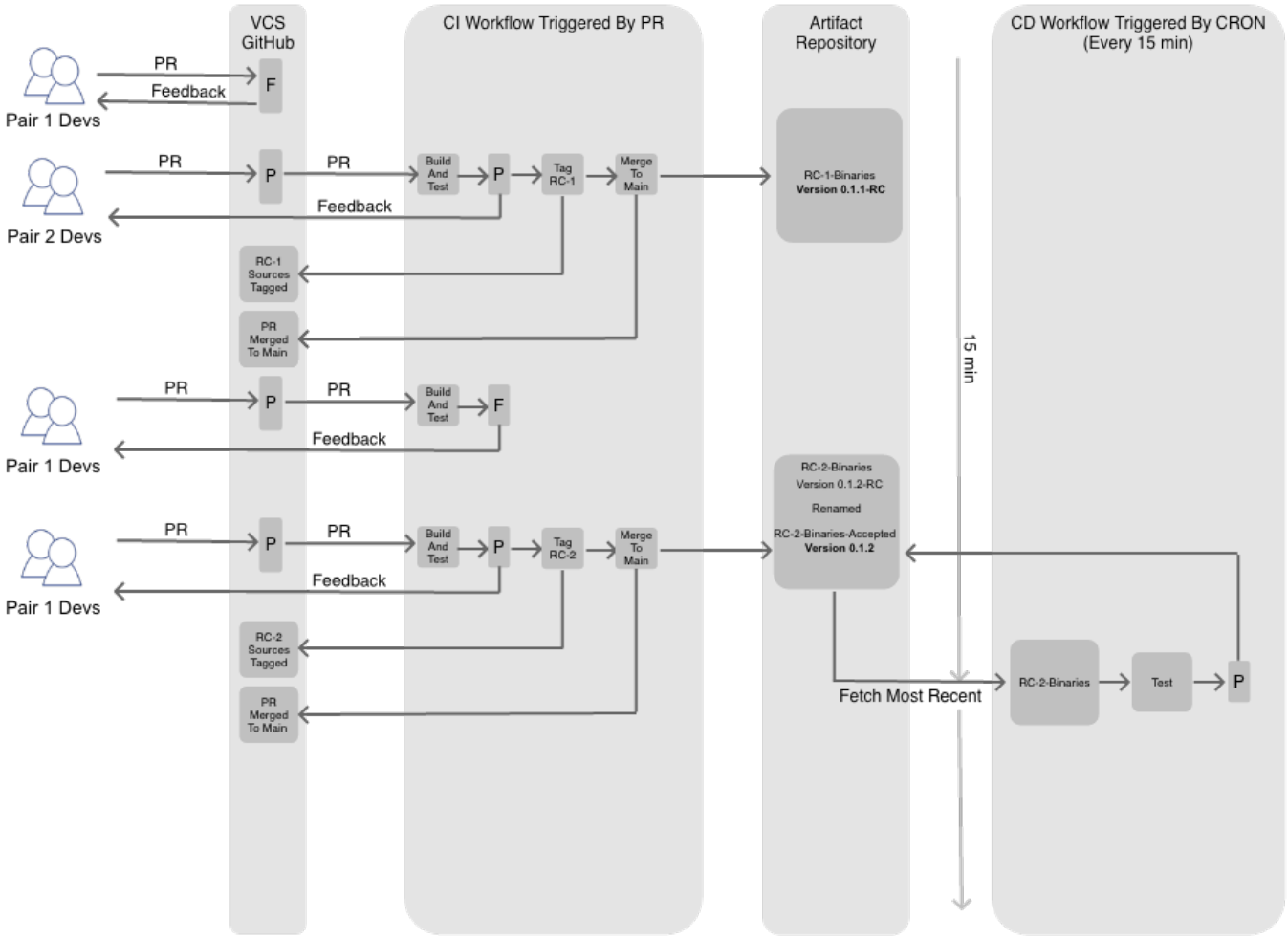
5. On GitHub create a PR for the branch to merge into main and submit it.

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.

8. Go back to step 1 and repeat.

Diagram Of The Flow



The Story Of Four Devs Working On The Vending Machine Project

That was a lot of words in the last section. If you are like me I need something visual to go with the words so I hope this little story with diagrams helps.

It's the beginning of the project and we are looking at the first 15 minutes. Pair 1 Devs have been working (pair programming) locally using TDD. They think they are at a spot where it makes sense to submit a PR. So after committing locally they push up their branch and submit a PR which they hope passes the

Commit Stage and is merged to the main trunk (main branch). However, something went wrong and the PR failed some branch check set in GitHub before even making it to the CI Workflow.

Meanwhile, the Pair 2 Devs submit their PR just a few minutes after and they have better luck. Their PR gets a pass on the first set of GitHub checks. The PR triggers the CI Workflow. It builds the software and tests it. The commit tests pass. The Pair 2 Devs were watching to make sure. Now they are free to start working on whatever is next but will keep in mind that their changes will eventually be picked up in the next stage of testing and they are responsible if there's a failure in that stage.

Looking at the diagram we can see there's a few things that happened once the Commit Stage tests passed. All the source files were tagged as a release candidate (**RC-1**), the PR was merged to trunk (main branch), the version was set to whatever the Devs set in the package.json file (**0.1.x**) for Major.Minor and the Patch was set by the CI Workflow to the **build_number** followed by **-RC** (**0.1.1-RC**) and the binaries were uploaded to the Artifact Repository.

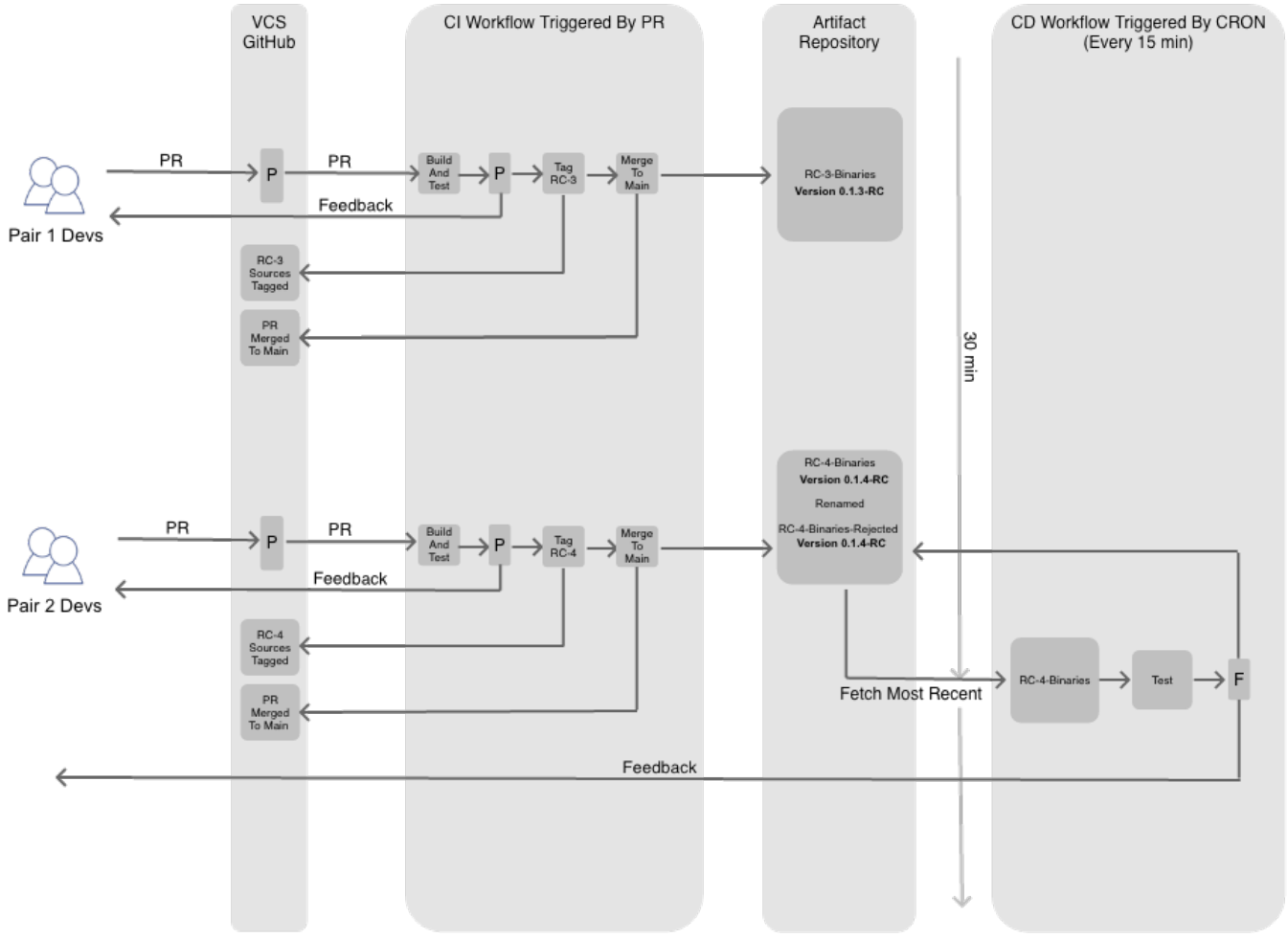
The Pair 1 Devs are having a hard day. They think they figured out their problem so they submit their PR again. It gets past the first set of GitHub checks, the software builds but the Commit Stage tests fail. Perhaps there's an incompatibility between their changes and the changes the Pair 2 Devs made. Better to find out now instead of a week later. Since their code change is small it shouldn't be too hard to figure out.

The Pair 1 Devs think they have it squared away now. They have synced up to main and merged it into their PR and ran all the Commit Stage tests locally. So they push up their short lived brach again then submit the PR. Bingo! Good work team! Now they have created a new Release Candidate (**RC-2**) that has their changes plus the changes the Pair 2 Devs got through earlier. Just in time for the second workflow which we are calling the CD Workflow. It's running

every 15 mins. Since this is Continuous Integration this workflow is only interested in the latest Release Candidate and it is **RC-2**.

The CD Workflow doesn't rebuild the software since we only build once in a CD Pipeline. Instead it retrieves the binary files from the Artifact Repository for **RC-2**. It runs the Acceptance Tests and as you can see in the diagram they pass. When the CD Workflow retrieved the artifact it was named **rc-2-binaries**. Now that the Release Candidate passed the Acceptance Stage it will be renamed to **rc-2-binaries-accepted**.

Note: *When the CD Workflow wakes up the first thing it does is determine the latest Release Candidate number. If the latest Release Candidate artifact name has been renamed to **rc-build_number-binaries-accepted** or **rc-build_number-binaries-rejected** then there's nothing for it to do but check back in another 15 minutes to see if there's something for it to test.*



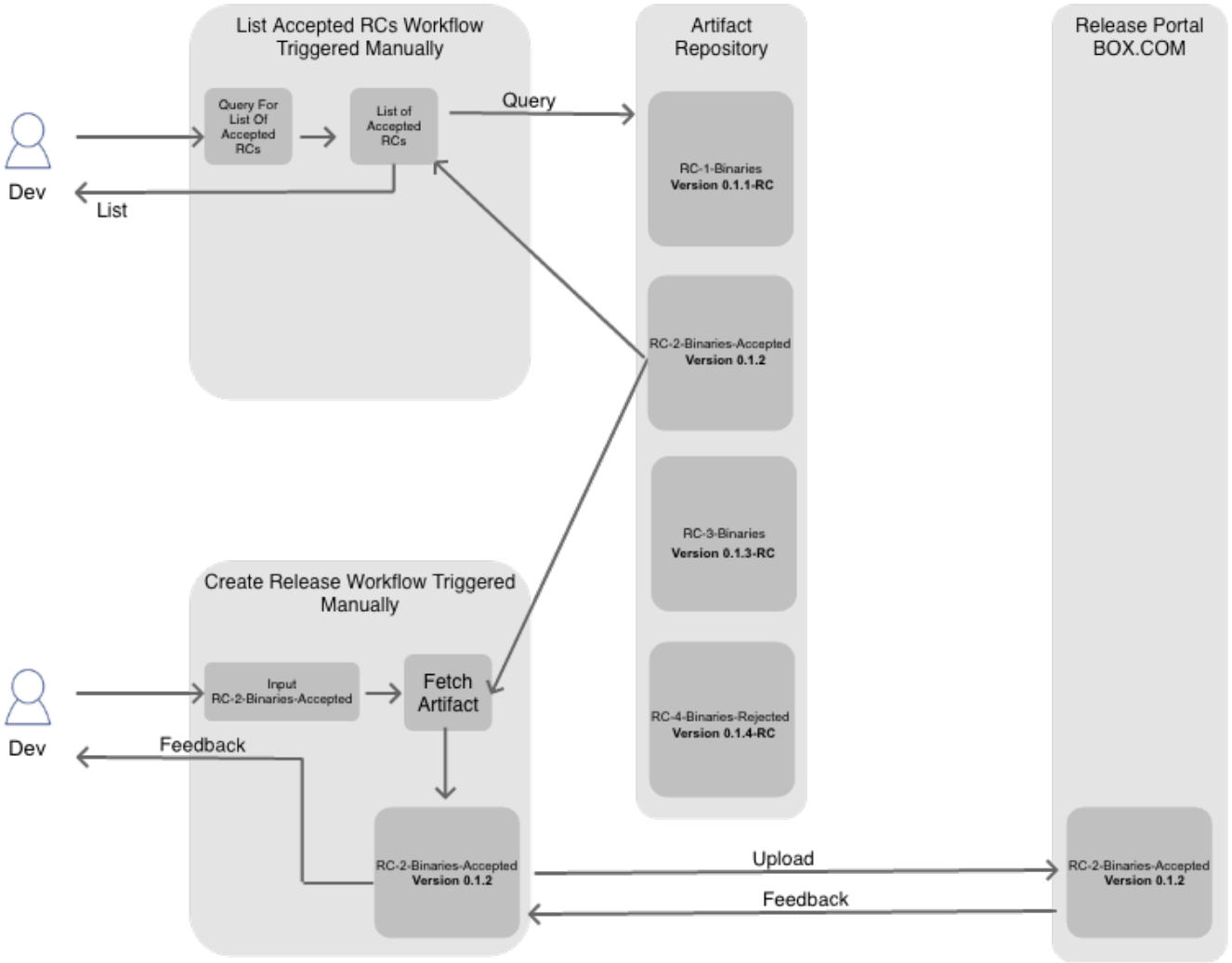
The story continues. Now we are going to look at the next 15 mins. Looks like the Pair 1 Devs have hit their stride. They've gotten another PR through the Commit State **RC-3**. Go team go!

The Pair 2 Devs are right behind them and they also get their PR through, **RC-4** which as before **RC-4** will also have the **RC-3** changes.

Now that another 15 mins has passed and you know what that means? That's right, time for our CD Workflow to wake up and test the latest Release Candidate in the Acceptance Stage. Oh, but hang on, it failed! Who's responsibility is it to investigate, Pair 1 Devs or Pair 2 Devs? The answer is both pairs since they both have changes in this cycle.

While they are investigating that issue let me show you how we can do a Release. In Continuous Delivery the idea is to always keep your software in a releasable state so that you can release it at least once per day. Since we had one Release Candidate make it all the way through the Acceptance Stage with a pass it is releasable.

First let's run our List Accepted RCs Workflow to see what's available (See diagram below). Usually, we will always take the latest Accepted Release Candidate but the system is flexible in case we don't want to do that for some reason. You can see when we run our List Accepted RCs Workflow that the only Release Candidate ready is **RC-2** which is named **rc-2-binaries-accepted**. So next we run the Create Release Workflow giving it the artifact name **rc-2-binaries-accepted**. It uploads the (zipped up) binaries to our Release Portal which is just a folder on box.com. Version 0.1.2 is ready for our users to install and enjoy.


List Accepted RCs Workflow Triggered Manually
Query For List Of Accepted RCs
List of Accepted RCs
Query
Dev
List
Artifact Repository
RC-1-Binaries
Version 0.1.1-RC
RC-2-Binaries-Accepted
Version 0.1.2
RC-3-Binaries
Version 0.1.3-RC
RC-4-Binaries-Rejected
Version 0.1.4-RC
Release Portal BOX.COM
Create Release Workflow Triggered Manually
Input RC-2-Binaries-Accepted
Fetch Artifact
Dev
Feedback
RC-2-Binaries-Accepted
Version 0.1.2
Upload
Feedback
RC-2-Binaries-Accepted
Version 0.1.2