Agile Advocate: Turbo-Charge Delivery with Multiple Release Streams

How to get that carefully crafted functionality into end users’ hands

Bill Drew

Tue, 02/20/2018 - 11:19

The Agile Advocate is a series of thoughts or musings on the agile movement and DevOps in particular. It was motivated by the seminal work of social philosopher Eric Hoffer, “The True Believer” (1951). It's not meant to be a manifesto, but simply a series of thoughts and reflections coming from over three decades of developing and delivering software and systems of a wide variety — to an even wider variety of users and customers. My aim is not to evangelize or convert, but to provoke and stimulate discussion.
In previous articles, we’ve talked about a number of items related to developing high-quality software and what’s required to establish the foundation for successfully delivering to your end user.

We’ve talked about the need to define our [continuous integration](#) requirements and what it provides or its output, the developer’s sandbox supporting experimentation, test-driven development and maintaining the integrity of the CI, the feedback loop enabling fine-grained course corrections toward the “best” product, and establishing an effective branching strategy as the first step toward building a migration path to production.

In this article, we’re going to talk about how we bring these all together. How we get that carefully crafted functionality into our end users’ hands in order to affect that outcome we’ve deemed so important. And not just deliver, but deliver confidently and reliably at a cadence that matches our end users' needs.

**Define Your Continuous Integration**

It’s critical you define what your CI will provide and how it will provide it. In doing so, you can clearly understand what your CI does “under the covers.” It’s not only important to understand what your CI does in terms of functionality, but also in terms of what it produces (the output).

Perhaps most importantly, you must understand the CI’s ability to deliver viable release candidates in a manner, which supports the “product owner to development team” [feedback loop](#). It is this feedback loop that allows development teams to apply fine-grained course corrections from POs and testers in order to vector the application toward the “best” outcome for the end user.

**Developer’s Sandbox**

Maintaining the credibility of the CI process and its ability to provide release candidates for inspection and evaluation by POs and testers is critical to successful delivery. In short, the CI process (or build) must not be “broken.” If a build “breaks’ or fails, fixing the build must become the first priority of the developer who broke it.

But at the same time, agile best practices call for a fine-grained commitment cadence for all developers. Many small changes committed on at least a daily basis are preferred over a few large changes over an extended period of time. Generally,
this allows for changes to be more easily integrated into the code base and reduces the level of effort required to back out or roll back changes.

These best practices may seem to be at odds with each other. On one hand, we need to maintain a stable build while on the other hand, we insist on constant changes to the code base — which could tend to destabilize the build. So, how do we satisfy these seemingly incompatible requirements?

That’s where the developer's sandbox comes into play. It’s a minimal build environment set up on a developer’s workstation, which allows for the execution of a small portion or specific phase(s) of the same build executed by the full CI process. Being able to execute it locally or outside the CI process provides “fast fail” feedback to the developer and encourages test-driven development and experimentation.

This does not require code to be committed to the trunk or master branch, which could break the build, jeopardize the CI integrity and interrupt the production of release candidates. Figure 1 below shows the workflow across a development team each using their own sandbox but staying in sync with changes to the code base.

Fig 1: Developer Sandbox
Release Candidates and the Feedback Loop

Maintaining the integrity and operational readiness of the build is critical because its outputs (the release candidates) are what fuels the feedback loop between POs, the testers and each of the development teams. As code is committed, the build for the trunk is triggered. If the build is successful, this results in a release candidate, which is deployed to the integration-testing environment during the integration-testing phase of the build.

The CI process should also produce an updated release notes document that records the new and updated functionality along with the bug fixes for that release candidate. POs and other interested parties can review this document to determine if there is anything they would like to review and provide feedback on the release candidate.

However, if developers are adhering to the practice of regular commits to the code base, then this integration environment will be updated frequently and may prove far too dynamic for POs and testers to properly review and validate functionality.

We’ll need to establish an environment where the CI product can be deployed for validation and testing. I refer to this environment as the “Release Candidate,” or “RC,” environment as it will host the most recent release candidate generated by the CI process as shown in Figure 2 below.

![Fig 2: Feedback Loop](image)

The decision to deploy to the RC environment is made whenever one or more POs determine, by reviewing the updated release notes, that the latest release
candidate has functionality they are interested in reviewing and validating.

The deployment of the release candidate is done with the same process and tools that will be used later to migrate the release candidate through the various environments along the path to production. Optimally, this process will make heavy use of infrastructure as code to dynamically provision and decommission the environment as needed. Automation via IaC provides for not only a repeatable, predictable, reliable process, but can also result in potentially significant infrastructure costs savings as resources are paid for only as they are needed and then decommissioned.

**Initial Release Stream**

All development targeted for the next release should be committed to the trunk. An entry or job is created in the CI server for the trunk for building and delivering release candidates that represent the latest version of the product.

For each release candidate, POs and testers are evaluating and providing feedback to the development teams. At some point along this process, POs in coordination with product management will decide the code in the trunk has reached a level of functionality that constitutes the next release.

At this point, a release branch is cut and a new job or jobs are created in the CI server that build from this release branch, as shown in Figure 3 below. The product of this build, which we’ll call Release 1.0.x., will be built and tested in the same continual integration and testing manner as we did with the trunk branch. It’s expected some additional functionality may be required and some bugs addressed. Many of these changes, especially bug fixes, will be merged back into the trunk.

Once POs verify and testers validate the release candidate functionality, the candidate is promoted to the test environment for end-to-end testing and additional testing by users. From there, the application is promoted to staging, which traditionally emulates production to a large extent and provides the opportunity for testing for performance, load, scalability, reliability and of course, security scans. From staging, the application can be deployed into production and delivered to the end users.
Multiple Simultaneous Release Streams

Note during the migration of Release 1.0.x to production, development proceeded along on the trunk for the next release. And just as with the previous release, once the code in the trunk gets to a point it constitutes the functionality required for the next release, which we’ll call Release 1.1.x, another release branch will be cut as shown in Figure 4.
In fact, the release branch for 1.1.x could have been done before the release 1.0.x had been deployed to production. By isolating each release in its own branch, we can have multiple, simultaneous releases moving toward production at their own pace without colliding or confining each other.

And as one is moving into production, another can be following right behind it, enabling an enhanced delivery rate without compromising on quality. And of course, this isn’t limited to just two or three streams, but can sustain as many work streams your development, user and testing resources can support.

In this article, we’ve touched on some techniques, best practices and approaches we’ve discussed in prior articles regarding software delivery. We looked at the importance of defining the requirements for your CI process, establishing a developer’s sandbox to promote experimentation and maintain the integrity of your CI process, the feedback loop to enable fine-grained course corrections and a branching strategy derived from industry best practices and past experience.
We've pulled these together here and combined them with multiple release or work streams to create a pipeline for developing, deploying and ultimately delivering high quality software. And by integrating the users into this pipeline, we haven’t simply accelerated our output, but also greatly increased our chances of delivering a positive outcome for our users.