Changing our Rhythm: Our Ongoing Journey towards Continuous Delivery

Sreerupa Sen
Senior Technical Staff Member, IBM
December 15, 2013

Abstract

In this experience report, I'll talk about how we transformed ourselves from a team that does one big-bang release a year, to one that delivers every quarter, and on our way to making more frequent releases and deployments. Our goal is continuous delivery, and though we haven't reached that goal yet, our journey has been an interesting one, both technically and culturally. We changed our way of operation, from how we plan to how we organize our teams to how we develop and test. Here I've tried to capture the essence of our transformation.

What is Continuous Delivery?

Continuous delivery is a practice to continuously build, test, and deliver software. If done well, this results in a highly efficient process that enables product developers to repeatedly push out new features and bug fixes to customers, while maintaining high quality and with minimal overhead. What does repeated mean? That could depend somewhat on the nature of the software you develop. Teams building web based software may be pushing changes many times a day. For folks like us who develop desktop versions as well, our best case may only be a couple of times a week!

What We Do, Who We Are…

My team builds a team collaboration tool that enables globally distributed teams to work together, following traditional or agile models of software development. It has desktop and web avatars, and now a hosted environment as well. So far we've released the hosted and non-hosted variants at the same time, but I can see that soon each will follow a different rhythm of external release more suited to its nature. We've over 100 people in the team distributed across the globe. We practice distributed agile.

Our development model has been agile from the start. Over the years we've honed our planning, team organization, automated build and deployment thereby streamlining our process for frequent deliveries.

Annual Releases: the Way We Worked

Up until about a year ago, we had annual releases with a sprint based development style. In parallel, we had quarterly test fixes to smooth out the rough edges of our product. Our sprints were 6 weeks long and each milestone build was published on the web for interested customers to try. It seemed fairly normal and agile enough – we did "self-host" on every sprint, did encourage our customers to try it out, and did try incorporating their feedback. So what wasn't working?

As it turns out, many things weren't working. Our releases were planned at the start of the annual cycle, and the long release cycles caused our customers a lot of pain. If a customer wanted a feature, they'd often have to wait more than a year for it, and so there was an endless negotiation of release content between development and field. Long release cycles also diluted our focus: we weren't working on our defect backlog in the most efficient way and our features often came together just in time and at the end of the release.

We realized that we would need to make more frequent releases to stay relevant to our customers.

Changing Our Rhythm

We needed to make several changes to move in the right direction,. We needed to vastly improve our automatic deployment and testing story. We needed to define and execute smaller features that we could push out more frequently. We needed crisply defined quality metrics that we'd measure our releases against. Finally, we needed smarter updates with minimum downtimes. Our goals fit naturally into the DevOps principles for continuous delivery: Plan and Measure, Develop and Test, Release and Deploy, Monitor and Optimize. The following sections describe what we've already done or plan to do.

Planning a release

When we switched gears, our first task was to break every feature in our backlog into 8-week releasable chunks. Our goal was to have 8 weeks of development for every feature, followed by 4 weeks of testing and defect-fixing, with more and more stringent rules around code delivery.

Though it seemed daunting at first, scoping made us look at what we did in a different way. With some practice, it was easier to plan for 8 week features than 18 week ones – the scope is small enough that one can estimate better and plan more predictably. Our common product backlog now has a ranked list of 8-week features in it which we share with our customers and reprioritize at the start of a release.

Features move from New to Ready when we have a clear understanding about its scope. Complex features may have an associated investigation task for an entire sprint or release before they can move to the top of the backlog. Features at the top of the backlog are ready to be implemented and move to the release plan at the start of a release. Thus we have a continuous pipeline of prioritized new features that we work on.

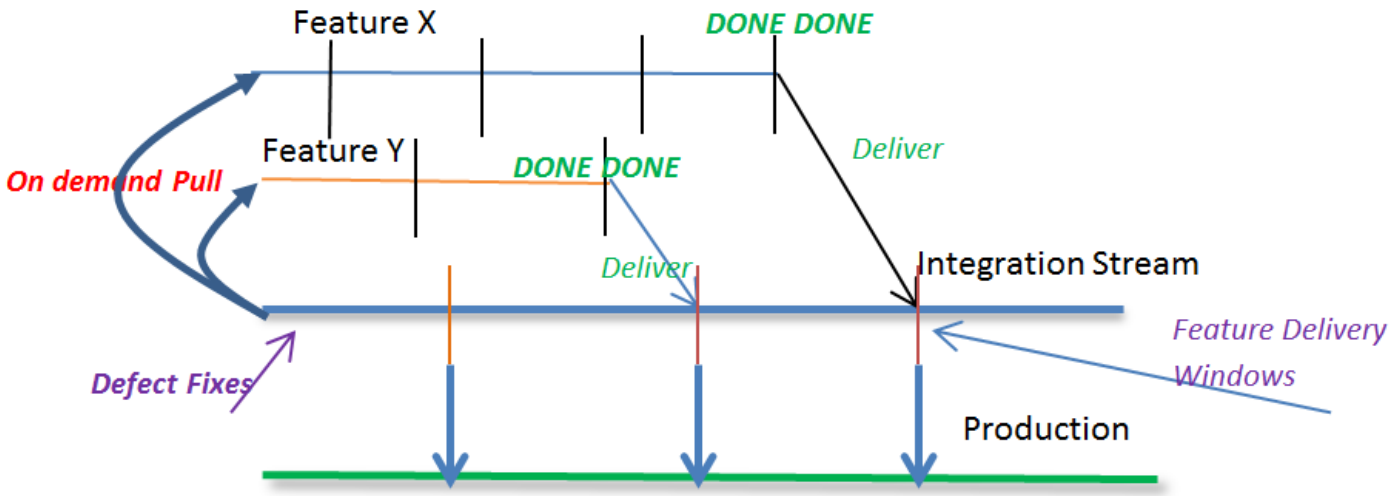
Team organization

One of our biggest challenges to quality and predictability has been developers doing too many things at the

same time. We needed to organize our teams very differently for continuous delivery to work. We needed a team focused exclusively on every feature that we built – from design to deployment, testing to documentation. Each such feature team is cross-cutting across the organization – developers, test automation engineers, user assistance, translators and so on. A feature team lead is the gatekeeper of the feature team.

Each feature team works with its own plan and on its own development branch which is called a stream in our tool's parlance. In our latest release, we're experimenting on when the best time it is for a feature to be merged back into the main steam. Thus far all features were delivered to the main development stream on a weekly basis or even a continuously, depending on the feature team's gating criteria. While this worked great when things were good, it made it very difficult to pull back a features cleanly if it couldn't be completed on time or for which the requirements changed. Now we are trying one way merges in which a feature team pulls in changes from the main stream, merges and runs tests, but doesn't deliver changes back until the feature is"DONE DONE". Our initial experience with this is promising though we have to try this out some more before we mandate it across the board.

Feature teams are complemented by Run teams. Run teams are responsible for the day to day operations of the product - working on the existing defect backlog and meeting the associated quality metrics, ensuring that builds are green, interacting with customers, responding to forum questions and triaging escalations. The Run Team works on the main stream, delivering defect fixes and small enhancements that customers have requested. Feature related defects are owned by the feature team.



This dual but separate focus on defects and new features alike seems to be working for us overall. Our defect backlogs are looking slimmer and new features are higher quality. Also our new features are not adding as many new defects as they used to.

This way of working has not been without challenges. The splitting of responsibility between the feature and run teams was a cultural change and was not an easy shift to make. There was a perception than Run Team was grunt work and that feature teams were cooler. Feature teams on the other hand felt isolated. We have

tried to address this by making these roles rotational, and also by having a free floating population of senior technical folks to provide technical guidance and mentoring to all teams. This approach seems to be working well.

Automated Build, Deployment, Testing

Our goal here is simple: creating automation that would allow our development team to request deployment of any build into an environment that resembles a production environment.

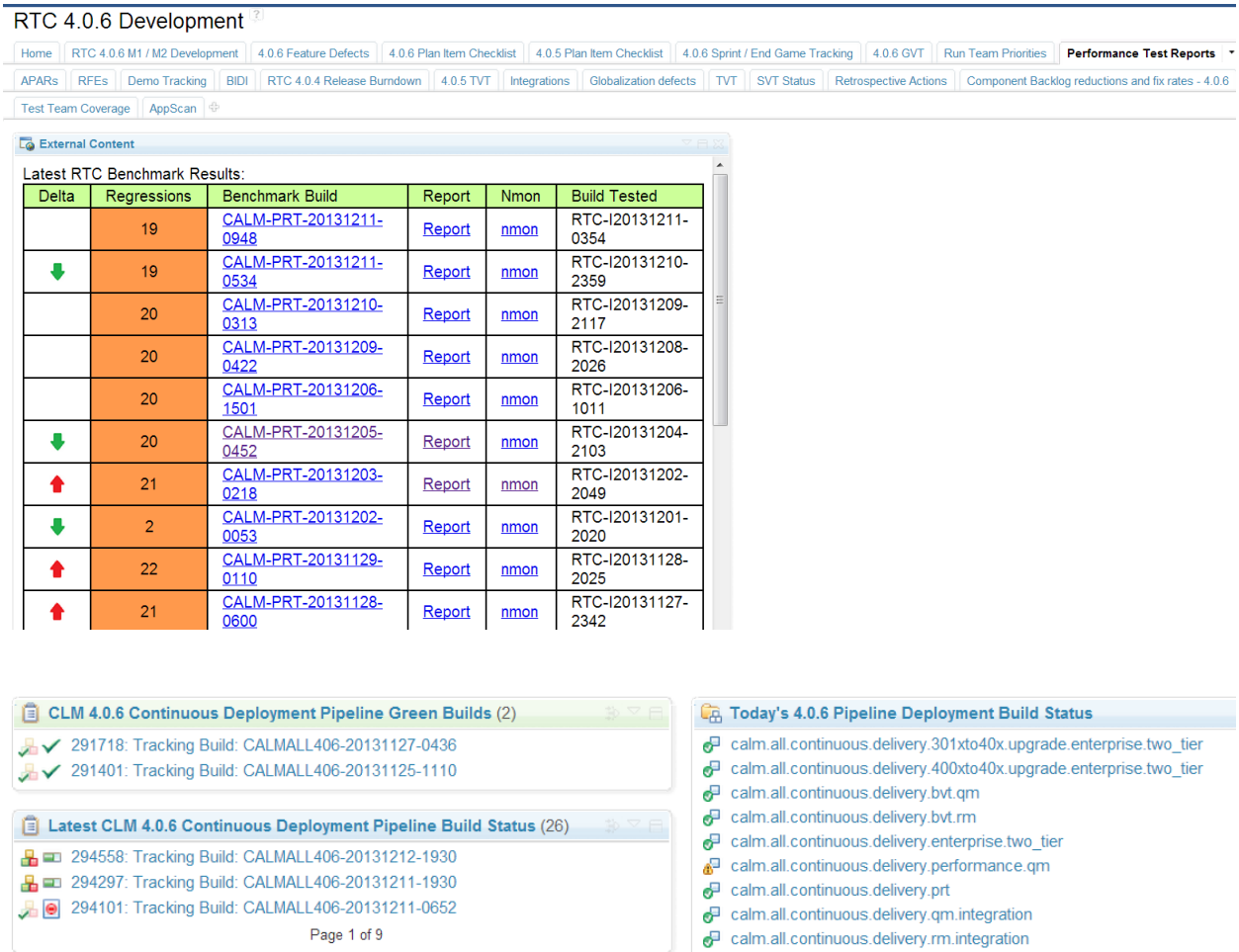
The Deployment Pipeline

We've always had automated builds and testing, but now we're additionally focused on virtualized deployment of our automated builds. Our build orchestration is a home grown one, written as OpsCode Chef recipes in the Ruby programming language. We've used our own tools for build and Cloud Deployment. We decided early on that deployment and test execution were distinct build phases and should be loosely coupled. We have created build pairs in which test execution builds can be paired with different types of deployment builds. A deployment build is associated a virtual system pattern that gets provisioned in the cloud. This type of pairing means that we can easily run our entire suite of tests on any virtualized platform with minimum fuss.

We have a continuously running deployment pipeline that provisions different production-like environments for executing the different types of automated tests: smoke tests, function test, system acceptance, and performance/reliability acceptance. These tests are run as a part of different types of builds and have different types of triggers. Not all of these builds are about building the software - many of them are about picking up pre-built bits and running a variety of tests. Some builds are fired at regular intervals and depend on code delivery. These are our continuous builds, often run for smaller teams working on specific features or defects, and by default on locally provisioned build machines. Our scheduled daily product integration builds run our entire suite of automated tests and take around 3 hours to complete. Our daily performance test builds run for around an hour. We have weekly builds such as system and performance acceptance that run a more elaborate suite of tests, and can take up to 6 hours to complete.

We record results from each deployment and test execution cycles in a common build tracking record that our development team uses for collaborating on the state of a given build. We monitor regressions via automatically generated reports.

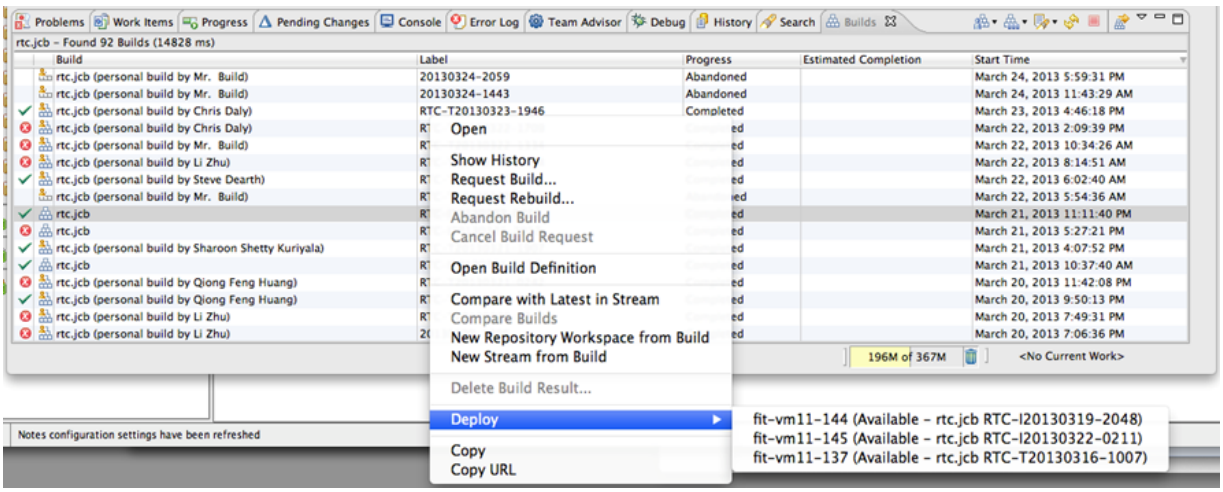
Here are a couple of interesting reports. The first one monitors regressions in performance. The second one shows the status of the various types of test execution builds in the deployment pipeline.



Improved Continuous Integration, Continuous Testing

A recurring theme from our retrospectives last year, from developers and testers alike, was that our way of testing needed an overhaul. Developers were testing features in non-production like environments using borrowed, quickly-munged systems, that required repeated manual setups. Developer testing wasn't as effective as it needed to be and testers were constantly struggling with regressions, broken integration builds, and faulty new features. This put pressure on testing teams and took away their focus from complex tests. Focusing on automated test driven development and continuously deploying builds have helped alleviate this by churning better quality integration builds with fewer breakages.

When a developer fixes a defect now, with the click of a button s/he can start a personal build that will integrate her changes with the rest of her team in a private workspace, deploy it in a virtualized environment, and run the tests she chooses. Once the tests pass, she's confident that her change won't break the build and she'll have more time and willingness to invest in unit tests that improve build quality. An interesting side effect of improved automated testing is that there's also more frequent code deliveries - developers no longer have the urge to hold on to lots of code and test it at one shot before delivering.



Improved continuous integration with continuous testing has positively impacted our defect backlog. From the usual 10% reduction per annual release, this year we've reduced the backlog by 18%, over the 4 quarterly releases we've had so far.

Going Forward

The cool thing about continuous delivery is that there's no status quo, it forces you to continuously improve your process. There are several steps we're taking to make ours more efficient as well.

We're investing quite heavily on automated UI driven testing. With the large number of browsers and platforms our product supports, even running a smoke test across platforms is a big chore. We're hoping to eliminate that altogether. Our goal really is that all tests that do not require any specific domain knowledge or expertise are automated, so that our bits can be continuously deployed with confidence.

Another thing we're looking at is parallel test execution to reduce our build durations. Some of our builds run for 4 or 5 hours and the bulk of that time is spent in testing. The challenge here is that to run in parallel, tests need to be independent and isolated, and need to have been designed with concurrency in mind. So our first step is to analyze our tests!

Finally, we're looking at a "real" continuous delivery with the hosted deployment of our product. The hosted and non hosted variants will share the same code versions, but whereas the first will have frequent deliveries to our external users, the non-hosted variant will follow a somewhat more relaxed release pattern. There too though, we're thinking of having more frequent deliveries via automatic updates via update sites.

To conclude, we're not there yet... but given the nature of continuous delivery, it will always be a sort of moving target We will continue to improve our process to get closer to where we want to be.