Continuous Delivery: Bridging Quality Between Development and Customers

Author

John Ruberto (Johnruberto@gmail.com)

Abstract

We were not moving fast enough for our business stakeholders. Development and quality was constantly the bottleneck for delivering the cool ideas from our product managers and designers. Our release cycles took three weeks, with more than half of the effort dealing with technical debt. We typically needed a full hardening sprint for each iteration. Our business team wanted to move faster. We had to bridge the gap between development and deployment.

What our business team wanted was a Continuous Delivery system, or even better a Continuous Deployment system. As soon as a story is complete, it's released to production. Each story does not have to wait for other stories, and our regression testing is completely automated.

We took this on as a challenge, and in six months implemented a Continuous Delivery system – where we push a story live on the day that story is completed. This paper will describe the changes we made with technology, mindset, and processes to achieve Continuous Delivery. It will also describe the principles, practices, and tools/methods which made this possible. Finally, a playbook is provided that can be used by others to make the same transformation.

Since we've implemented Continuous Delivery, our delivery velocity has doubled, the scope of the team has grown, and the engineering team is leading the charge on innovation.

Biography

John Ruberto has been developing software in a variety of roles for 28 years. He has held positions ranging from development and test engineer to project, development, and quality manager. He has experience in Aerospace, Telecommunications, and Consumer software industries. Currently, he is a Director of Quality Engineering at Intuit, Inc. He received a B.S. in Computer and Electrical Engineering from Purdue University, an M.S. in Computer Science from Washington University, and an MBA from San Jose State University.

1 Introduction

“Three weeks is not fast enough. We want to get these designs out now!”

Our team thought we were moving fast. We had a three-week release cycle and used Scrum to manage our projects. However, our Product Management team didn’t think we were releasing fast enough. Our team also supported multiple project teams, with their own independent release schedules. Aligning their cycles with our three-week cycle was frustrating.

We decided to move to a Continuous Delivery model, where we could release any story as soon as that story was complete. Continuous Delivery eliminated the need to wait for system/regression test cycles, thus speeding up the delivery of individual stories. It also allowed us to more easily align with our partners release schedules.

This paper will tell the story about how we made this transition. The story involves changes in people’s mindset as well as changes in processes, technology, and architecture.

2 Definitions

Consider this life-cycle view for the following definitions. Figure 1 shows an iterative Software Development Life-cycle that results in code deployment after System Testing. The “Define” stage is comprised of requirements analysis & definition, as well as iteration planning. For Continuous Integration systems, the Unit Test & Integration Test activities are fully automated. System Test in this context includes Regression Testing, System Testing, and User Acceptance Testing. With Continuous Delivery, these System Tests are automated. The Deployment phase includes staging, deploying to production, and deployment validation testing.

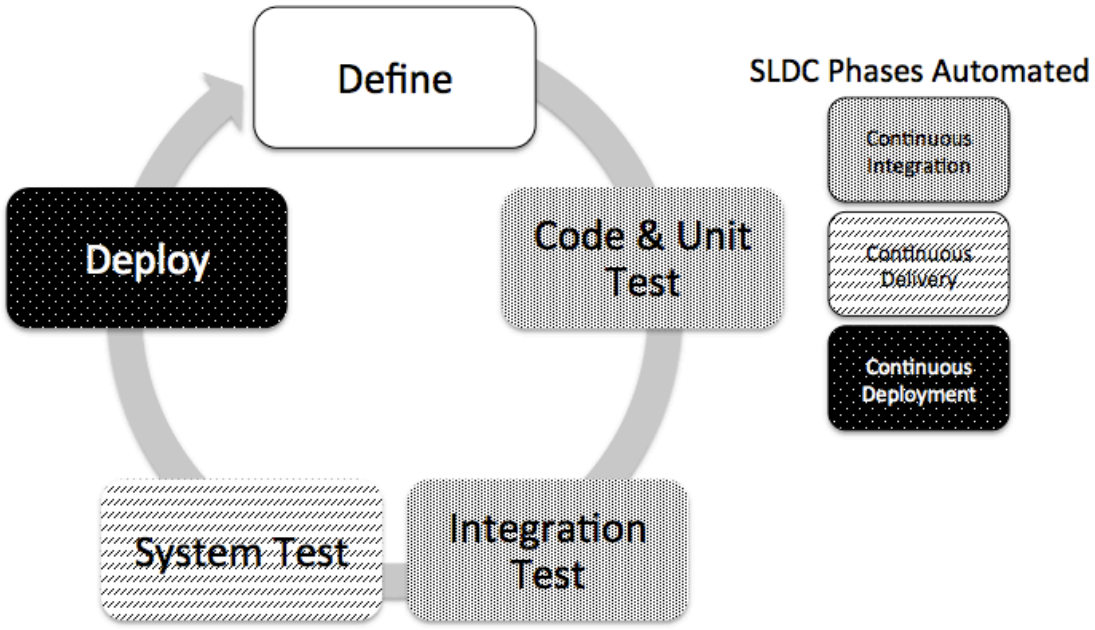

Figure 1. High level Iterative Software Development Life-Cycle with phases impacted by Continuous Integration, Continuous Delivery, and Continuous Deployment. Continuous Integration automates unit tests and integration tests, Continuous Delivery involves automating System test, and Continuous Deployment automates deployment.

2.1 Continuous Integration

According to Wikipedia, “Continuous Integration is the practice, in software engineering, of merging all developer working copies with a shared mainline several times a day”.

In practice, Continuous Integration (CI) means to us a set of automated tests that are triggered with each build. The tests are intended to give confidence that the new code developed for each build functions as expected and does not cause breakages in the existing code. Non-functional tests are also executed, including performance tests and static analysis.

The intent behind Continuous Integration is to give rapid feedback to the development team and to keep the code healthy at all times.

Continuous Integration is a necessary pre-condition for Continuous Delivery.

2.2 Continuous Delivery

With Continuous Delivery (CD), we extend the concept of Continuous Integration to include customer acceptance tests with the regular automated test suite.

According to Wikipedia, “Continuous Delivery is a design practice used in software development to automate and improve the process of software delivery. Techniques such as automated testing, continuous integration, and continuous deployment allow software to be developed to a high standard and easily packaged and deployed to test environments, resulting in the ability to rapidly, reliably, and repeatedly push out enhancements and bug fixes to customers at low risk and with minimal overhead.”

In our practice, while the Customer Acceptance Tests are fully automated, the decision to deploy to production environments is still a decision made jointly by the project team and business stakeholders.

The value of Continuous Delivery, as practiced by our team, is to break the dependence on a regular release cycle (every three weeks) and instead deliver a user story or bug fix when ready.

Paul Duvall *[Duvall, 2014]* defines Continuous Delivery as “With Continuous Delivery (CD), teams continuously deliver new versions of software to production by decreasing the cycle time between an idea and usable software through the automation of the entire delivery system: build, deployment, test, and release.”

2.3 Continuous Deployment

Continuous Deployment extends the concept of Continuous Delivery by fully automating the production deployment of the application as well. Once code is checked-in, it is automatically tested and if the tests pass, is automatically deployed. No humans are part of the deployment decision.

For our application and business context, we didn’t see the need to use Continuous Deployment. The rest of the paper will concentrate on our experiences with Continuous Delivery.

3 Benefits of Continuous Delivery

Building a Continuous Delivery system is quite a significant investment in technology and time. This section describes the benefits of Continuous Delivery and why a team might want to undertake this investment.

3.1 Increase Velocity of Innovation

For the project described in this paper, the primary benefit expected from Continuous Delivery was an increase in the delivery velocity of innovation. Our product management team had a long list of features and enhancements they wanted to build. Each one of these enhancements was thought to improve customer value and reduce our costs. So, delivering at the fastest pace possible was a strong desire. Our baseline release cycle called for a release every three weeks, but the team wanted to release, and benefit from, these features as soon as possible. Continuous Delivery allows for stories to be delivered when the stories are finished with code, testing, and acceptance – eliminating the wait for a "bundle" of stories to be delivered.

3.2 Planning Flexibility

Another benefit of Continuous Delivery is planning flexibility for release planning. The ability to release each story independently, and when that story is complete, takes away the need to plan which bundle of stories work best together. The team can simply work on the highest priority stories and release when ready.

Many of our stories are experimental in nature. We often implement two versions of a solution and monitor which solution performs better with customers. This practice is called A/B testing and is intended to identify the best solution to a given problem. Having the ability to learn which solution is superior soon, and deliver that solution to all customers rapidly adds to the value of our application.

3.3 Avoiding Technical Debt

One of the strongest benefits for building Continuous Delivery is that managing technical debt becomes vital; technical debt is kept to a minimum. One of the principles of Continuous Delivery is that the code base is always ready for production deployment. Each new story has to come with its tests, and code is only promoted to the next stage when all of the tests pass.

This discipline forces the team to rapidly fix bugs, keep automated tests running at high coverage levels, and exposes very quickly any gaps in the test coverage.

3.4 Teamwork Benefits

David Farley and Jez Humble wrote an excellent book on Continuous Delivery *[Farley, Humble, 2011],* where they described several benefits of CD:

- Empowering Teams
- Reducing Errors
- Lowering Stress
- Deployment Flexibility
- Practice Makes Perfect

4 Continuous Delivery might not be appropriate in all cases

Continuous Delivery does have many advantages, but fully implementing CD might not be appropriate for all applications. Some situations may make it difficult to fully implement CD.

4.1 Requires a low cost of deployment

A key benefit of CD is frequent deployments. This advantage is only appropriate where the cost of deploying is low. “Software as a Service” (SaaS) applications seem to be natural fit for CD, where the developer is fully in control of the deployments, and deployment can be automated.

CD might not be the right choice for delivered applications, where the customer has to install or update the application. Frequently updating desktop and mobile apps might send the wrong message to customers, and cause customer frustration with having to apply frequent updates.

Other considerations for the cost of deployment include documentation updates, bandwidth for distributing code, manual installation, and training costs.

4.2 Regulatory environment might not be compatible

In CD, much of the accountability for quality revolves around the team instead of individual decision makers. The practice has few “signoffs” and the documentation trail might not clear. The regulatory environment for your application/industry should be considered before making a choice to use CD.

4.3 Difficult to Retrofit into existing applications

It was much easier to build CD into the new application, instead of retrofitting into the existing application. The technology choices and lack of testability in the older application made easier to start fresh.

4.4 Applications that operate on data

When moving fast with release cycles, and relying on fully automated tests, sometimes problems escape to production, making it necessary to back out a release. With applications that operate on data, this is often difficult to accomplish. If a new enhancement changes the way data used, and customer actually use the new functionality, it might not be easy to roll-back the application to the previous state. Roll back is possible, but must be designed into the application accordingly.

5 Context for our application

5.1 Self-help platform for multiple products

The project described in this paper is the Intuit Unified Self-Help Platform (USH). This is a web application that provides our customers with help information drawn from a variety of sources and content management systems. The information is provided through a web site, in-product help-viewers, and guided workflows. The intent of the system is to help customers find the right information they need to run their business without having to call technical support.

Providing the best information, which is easily digested by end customers, is very valuable to customers and to our company. Our customer gains value by finding the information they need quickly, without having to call support. Our company benefits from not having to field those calls. Also, our research has found that customer satisfaction is much higher for customers that do not have to call support.

With millions of customers, releasing an enhancement a few days earlier has very tangible benefits to the business.

5.2 Release Process

Before undertaking this project to refresh the platform, our team worked with a three-week release cycle. The first two weeks were allocated to planning and development, with the final week dedicated to system testing, fixing bugs, and user acceptance testing.

Figure 3 illustrates our release life-cycle.

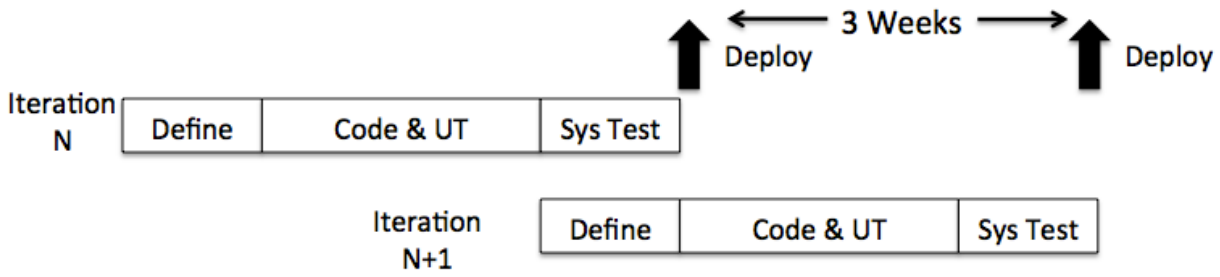

Figure 3. The original release life-cycle, before Continuous Delivery, with three weeks between deployments. Code & Unit Testing took two weeks, while System Test took 1 week.

5.3 Challenges

Even though a week was dedicated to testing and fixing bugs, our system still contained a lot of technical debt from previous projects. Approximately half of the effort for each release was spent fixing bugs and refactoring code. This extra overhead slowed down our ability to release customer-facing stories.

Most of the testing was conducted at the system level, with manual tests. We had very little test automation to aid the release. Executing these tests took time and effort, and any defects found during this time were found late in the release cycle, giving our team very little time to correct them before the release date. Many times, a story was scratched off the release because of bugs found late in the cycle.

Our customer base was growing and the complexity of small business accounting was growing. These factors drew many more customers to the self-help application, causing our costs to increase. Our business teams wanted to mitigate these costs as soon as possible.

5.4 Attributes beneficial for Continuous Delivery

This application had a few attributes that were beneficial for building a Continuous Delivery system. First, this was a web application; we controlled the deployments. We were free to release code when ready without customers needing to install any updates, take any action, or even know if we deployed an enhancement. Deployment was fully in our control.

The application also contained very little state information or business logic. These factors made it easier to implement a roll-back, if that proved to be necessary.

6 How we changed

This section tells the story of how we changed from a three-week delivery model to a Continuous Delivery model. Starting with the business rationale for making the change, the change leadership steps, technology investments, and the tools and practices that helped make this project successful.

6.1 Built the case for change

The business case for making this change started with conversations with the product management team. They were asking for certain stories to be delivered "off-cycle", to be delivered earlier than the full release. This conversation occurred enough times for the development team to see the trend.

We did a thought experiment. Looking at the current process, code-base, and state of test automation, we asked how we could cut the test duration in half (this include unit, integration, system, and acceptance testing). The resulting brainstorm showed that we would have a very large investment and gain just one-week reduction in the release cycle. Continuing the thought experiment, we realized that it would be very difficult, and a long time, to incrementally improve our code & testing practices to create the planning flexibility required by the business.

Another factor that went into the decision was the state of our technology used for the self-help site. The platform contained many home-built components that had to be maintained. These components also lacked the automated tests that would lead to speedy release. Newer technologies were available, with platforms/frameworks that contained much of the needed functionality. Using this opportunity to refresh our technology and adopt an existing platform added to the business case.

Our engineering division also had some technology goals, namely to deliver more functionality in the cloud and to build our capability for Continuous Integration/Continuous Delivery. Our existing platform did not lend itself to hosting in the cloud, nor was it easily updated for Continuous Delivery.

We built the case for change based on the following factors:

- Faster time to delivery of each story (helps our customers faster, which reduces costs sooner)
- Technology refresh to a new platform will allow our developers to spend more time developing customer facing innovations, and require less time maintaining the platform. Over the long term, the productivity gains would offset the switching costs.
- Our hosting costs would be much lower by hosting in the cloud, and this prepared our system for global growth

6.2 Set CD as a goal

Once we had a good business case, we added Continuous Delivery to our annual goals. We made sure that business team set a goal for CD as part of their goals. We also made sure that our technology goals (CD, hosting, platform) were part of the larger engineering division goals.

Having our project on the goals for the business VP and engineering VP gave us the visibility to make this change happen. We were also given some extra budget to build the new system while maintaining the existing system.

6.3 Investments in people

We needed a team of people that shared the values of CD. Both the quality engineering and the development team had to share these values.

Since we were maintaining the existing site, in addition to build the new site, we were able to hire a few extra engineers to enhance the team. This gave us the opportunity to find people with both the technical skills required but also the mindsets described in the following table.

Traditional Software Mindset	Continuous Delivery
Structured hand-offs between functions (PM -> Dev -> Quality -> Operations)	Collaboration between all functions.
Build then test	Test first (Test Driven Development and related practices)
Quality team responsible for testing	Everyone tests
Tests reveal problems	Accountability with each build (builds fail if tests fail, coverage falls, or static analysis detects poor code)
Manual Deployments	Deployment automation built in

Table 1. Summary of mindset changes that helped foster Continuous Delivery.

Building quality in from the beginning is very important. Several of the new developers were “Software Engineers in Test”, and were therefore very adept at creating tests & they were the leaders of the “test first” mindset.

In addition to hiring new people, and transferring people from test roles, we also invested in training the existing team. We brought in a professional training company to give a one-week training for Ruby on Rails, our chosen technology.

6.4 Investments in technology

One of the constraints for releasing faster with the legacy system was the state of our technology. We had built the application several years prior, without designing in testability. Also, we build many of the components used, such as A/B testing framework, ourselves – without unit tests built in.

Here are the factors used for choosing the new technology stack:

- Productive programming language: ability to implement functionality with efficient amount of code
- A large library of components available to integrate, which increases developer productivity, and increases quality by reusing code.
- Popular with developers, to find a good labor pool and entice developers to learn the platform
- Framework that provides the Model View Controller (MVC) pattern, internationalization ready, abstracts the database, and lends itself to test driven development.
- Compatible with other technology initiatives in our company, to leverage learning from other teams.

We chose Ruby-on-Rails for these reasons, with the following major components and tools

Component/Tool	Purpose	Link
Ruby-on-Rails	Main development framework	http://rubyonrails.org/
RSpec	Test/Behavioral Driven Development, Unit Tests	http://rspec.info/
Rubymine	Integrated Development Environment	http://www.jetbrains.com/ruby/
Git and gitflow	Source Control and Branching	http://git-scm.com/ https://github.com/nvie/gitflow
RubyGems	Component Library	http://rubygems.org/
RailsCasts	Training and Learning	http://railscasts.com/
Jenkins	Continuous Integration	http://jenkins-ci.org/
Chef	Automated Deployments and environment configuration	http://www.getchef.com/chef/
Rubocop	Static Analysis and style checker	https://github.com/bbatsov/rubocop/
Collaborator	Code Review	http://smartbear.com/products/software-development/code-review/
Jmeter	Performance testing	http://jmeter.apache.org/
Selenium Webdriver	User Acceptance Testing	http://www.seleniumhq.org/

Table 2. Summary of tools and technologies used by the Unified Self-Help team to implement Continuous Delivery.

6.5 Agile Development Practices

We used Scrum for the overall project management, development, and collaboration process. We had used Scrum before and everyone on the team was already trained. With seven engineers, one scrum team was sufficient.

The Product Manager was a natural fit for the Product Owner role. This gave the product team full control over the stories and priorities, and they had the accountability for our expected business outcomes (reduced call volume).

One of the developers on the team played the Scrum Master role. This person has great interpersonal skills, the ability to follow up with people without adding a lot of friction; he was well organized, and had a strong personal disciple.

The Scrum team was comprised of developers (both front end and back end), quality engineers, and Development Operations engineers. This multi-disciplined team had the skills to tackle all of the stories in the backlog.

6.6 Practices

One of the core differences between the traditional model of software delivery and Continuous Delivery is the automated test coverage, and quality of these tests. Continuous Integration is at the core of Continuous Delivery.

Tests were built in from the start by using Test Driven Development using the RSpec framework to author and execute the tests. The basic practice involves the developer taking the user story and decomposing it into a series of lower level specifications. These are written within the RSpec framework, and become executable tests. Since the code to implement the functionality is not yet created, the tests initially fail. The developer then implements the functionality, and keeps running the tests until they pass.

This practice of creating the test first ensures that the application has high levels of test coverage. The following code example shows both the specification and test for displaying the US flag when the site is using the US English language localization. This specification and test is created before the operational code is created.

```
describe StaticController do
   describe '#index' do
      context 'when provided a specific locale' do
            context 'en_US' do
              before { get :index, region: 'us', language: 'en' }

              it 'show the US flag' do
                expect(assigns(:flag)).to eq 'us-flag'
              end
            end
      end
   end
end
```

Figure 4. Code example of a Behavioral test, which captures the expected behavior and implements a test for that behavior

Keeping the code in a constant releasable state is another benefit of Continuous Delivery. This is accomplished by using the automated test results as gatekeepers for promoting the code to the next level. The following diagram illustrates the four stages of code maturity that we used. These maturity levels correspond to, and are named after, the environments where the code is running. These levels are Developer, Integration, Pre-Production, and Production.

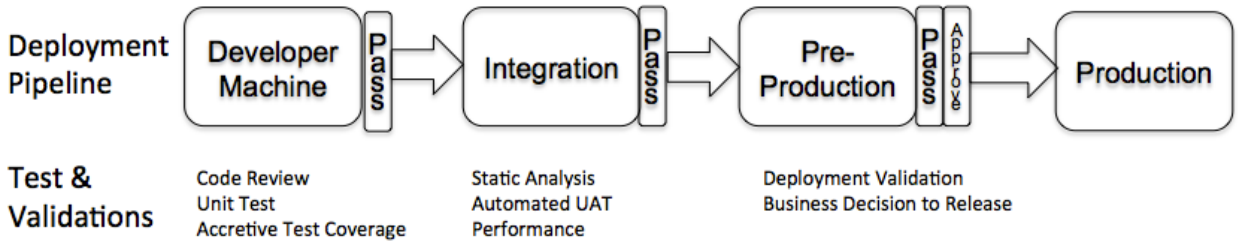

Figure 5. Development workflow showing the deployment pipeline with the tests and validations performed at each stage.

Developer to Integration

For any code changes to be promoted from the Developer stage, the code must have been successfully code reviewed, have passed all of the unit tests, and the measured code coverage does not decline. The team is very diligent and quality focused. This ensures that everyone follows these guidelines; they make themselves available for code reviews, and make sure to have good code coverage. Unit tests are reviewed along with the functional code, to help make sure that we have good tests in addition to tests that execute the code. These promotions happen several times per day, when initiated by the developer.

Integration to Pre-Production

The Integration stage tests are typically executed in the evening, with results ready the next morning. These can be initiated at any time by the quality engineering team in case of emergencies, but typically the automatic nightly execution is sufficient.

These tests include Static Analysis, verifying code style, potential errors, and bad coding practices. The suite of User Acceptance Tests (UAT) is also executed. These UAT are authored with Tekila *[Bhalla, Bhandari, 2010]*, our test automation framework based on Selenium Webdriver. Performance tests are also executed for the key workflows in our application. We used jmeter to create and execute these tests.

With successful Static Analysis, UAT, and performance tests, the code is promoted to the Pre-Production environment. We execute several deployment validation tests and monitor the pre-production environment while waiting for business team to decide on deploying to production.

Release to Production

The Scrum master and product owner decide to release a story by reviewing the test results and sometimes reviewing the business objectives behind the story. They give Dev Ops the go-ahead to release to production.

Post-Production Validation

We monitor the site availability, performance metrics, and key business metrics after production deployment. If any problem arises, we have the option to fix in place, or to roll back then fix. This decision is based on the severity of the problem and the estimated time to fix the problem. In practice, most of these issues are simple and it is just as easy to fix, rather than roll back.

6.7 Oversight/follow-up

Building Continuous Delivery into an application is a lot of work, with a long-term payoff. In business, there are many opportunities and pressures to see results quickly, putting pressure on the team to make shortcuts. We had a few incentives in place to make sure we stayed on track and actually delivered the application, with Continuous Delivery.

Our annual goals were shared through all of the levels of the engineering organization. The VP of Engineering for our division had the goal of building Continuous Delivery capability in the organization. We volunteered our project to be part of that goal. This gave us visibility at his staff meeting, where we gave quarterly progress reports and help when we needed help.

We also included Continuous Delivery as part of our department goals and the leadership team (development and quality managers). Each engineer had as part of their explicit goals the behaviors and outcomes required to make Continuous Delivery a success. These included test driven development, automation test coverage, and the team behaviors listed earlier.

We reviewed the progress of Continuous Delivery at each program review, and the behaviors were included with individual performance reviews.

The DevOps team is vital to the success of Continuous Delivery. We made sure to have dedicated support by converting a developer headcount to the DevOps team, which allowed them to hire someone with experience and dedicate to our project.

7 Summary / Playbook

This paper described the experience that our team gained in creating a CD enabled application. The following list gives the "playbook" for using Continuous Delivery for any project.

1. Make sure you have the business need. CD is quite an investment in time and effort. CD is also a hot topic in software development, which makes it easy to try to force CD into an organization. Having a strong business need to deliver enhancements incrementally should be in place before attempting this.
2. Make sure your application is compatible with CD.
    a. SaaS applications work well, since you control the deployments without customer action.
    b. Your application should have the ability to be deployed quickly and automatically.
    c. Environments were failure might have extreme consequences or the process has high levels of accountability might not be compatible with CD.
    d. Your application also needs the ability to rollback in case of errors.
3. Make sure your organization is committed to CD. The process investment is high and has a long-term payoff. The technical and business stakeholders need to be willing to take this investment.
4. Make sure your team has the right mindset. Everyone on the development team needs to create tests, and to build quality in. The whole team needs to hold each other accountable to maintain this high level of quality. Traditional gatekeepers (QA, Release Management, Product Management) are giving some control to the development team and the process.
5. Make sure your processes are Agile. CD requires quick and efficient decision-making which requires quick feedback loops.
6. Make sure you use the right technology. Testability is vital in CD and this includes functional as well as non-functional tests.
7. Start small and build incrementally. CD is very amenable to incremental improvements. In our case, we started with just unit testing and code review for the very early version of the app. Then we added UAT, Static Analysis, and performance testing.

8 Conclusions

Continuous Delivery worked well with this application. Since starting this project, I've seen more and more teams try to adopt Continuous Delivery and Continuous Deployment, in part because these practices are hot topics in industry. Having experienced this first hand, I've seen the importance to ensure that the business need is in place, the right people and technology investments are planned for, and the application be suitable for Continuous Delivery before embarking on this goal.

Having the ability to release at any time has been a great benefit for our application. Many times we were able to synchronize with an event happening in the company with our supporting functionality, without having to synchronize schedules. We were also able to move faster with more experimental functionality, knowing that we had the opportunity to fix it quickly, or back out, if warranted.

References

Duvall, Paul M, “Continuous Delivery: Patterns and Antipaterns in the Software Lifecycle”, http://cdn.dzone.com/sites/all/files/refcardz/rc145-010d-continuousdelivery_0.pdf. (accessed July 29, 2014)

Bhalla, Kapil and Bhandari, Nikhil. “Web Test Automation Framework with Open Source Tools powered by Google WebDriver.” Proceedings of the Pacfic Northwest Software Quality Conference, 2010

Farley, David and Humble, Jez, 2011, Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation, New Jersey: Addison-Wesley

Wikipedia, “Continuous Integration”, http://en.wikipedia.org/wiki/Continuous_integration, (accessed August 2nd, 2014)

Wikipedia, “Continuous Delivery”, http://en.wikipedia.org/wiki/Continuous_delivery, (accessed August 2nd, 2014)