Automating Deployments in Mobile App Development

Advanced Software Engineering (F2015)

Kristian S. M. Andersen

Supervisor: Yvonne Dittrich
Submitted: May 13, 2015

IT University
of Copenhagen

Abstract

This is an abstract

1 Introduction

In the early spring of 2001 Kent Beck and 16 other software developers signed the Agile Manifesto [1]. The manifesto consists of 12 principles that lays the foundation of agile development methods. The first principle the manifesto says:

“Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.” [1]

Now more than 10 years later many compaines adopt many the concepts from agile development and even claim to be agile. Still many of these companies have yet to adopt the first principle.

In the development of large scale web-infrastructures it is common to use Continous Delivery techniques to facilitate small rapid development cycles. These techniques are not as wildely adopted when it comes to smaller products such as Apps installed on mobile devices. Many of the challenges are the same though the tools differ a bit.

I work in a small danish company that develops Apps for the iPhone and iPad. The company has 17 Apps in production which is developed and mainted by 7 developers. Most of the apps in production needs both weekly and daily builds. This constitutes a problem since every app has a different build process and none of the processes are automated. A build can take anything from 10 minutes to 2 hours. [5]

1.1 Structure of the Paper

In chapter 2 I will further refine the problem and highight three key aspects that needs to be adresses by the solution. In chapter 3 I will relate the problem to the literature around the subject. In chapter 4 I will propose a solution to solve the problem. In chapter 5 I describe how to evaluate the proposed solution against the problem. Finally in chapter 6 I will reflect on and conclude the paper.

2 Refine the problem

I work in a small company, called Robocat, where we develop Apps for iOS and Mac OS X. We have around 17 actively maintained products which are developed by 7 developers. At any given moment each developer may be involved with 2 or 3 different products. Each week the entire team and any potential clients needs to receive a weekly build of the updated products. Builds are also requested ad-hoc several times during the week. Producing a new build and deploying it for test is a process that usually takes between 10 minutes to 2 hours depending on the product. It can in some cases take even more time. Everything in the build process an deployment is done manually.

This constitutes a problem for many different reasons. In the following sections I will present the problematic symptoms of this manual process.

2.1 Build Documentation & Streamlining

As mentioned earlier Robocat has 17 active apps currently in production. All these apps are actively maintained and require at least several updates every year just for maintanence. The most problemtatic issue is that every product has a seperate build process. All products are build with the same tools provided by Apple but the individual build steps for each app differ in subtle ways. They have wildely differnet environment variables, runtime setups, libraries, etc. Some apps depend on many 3rd party libraries that needs to be updated with each new build. This means there is absolutely no streamlining of the build process. Each app is build separately in different ways.

To make matters worse there is seldomly any documentation available in the projects for how to build and deploy the apps. Important knowledge pertaining to individual build steps for products often resides with the lead developer of a product. If the developer is sick or on vacation the build may take longer to produce or can't be produced at all. This is especially problematic if an important bugfix update needs to be shipped to customers.

2.2 Deployments break the workflow

Builds can take a lot of time to produce. Some apps only take 10 minutes to build and deploy while others can take several hours. Some apps are build once a month while others are build and deployed daily. Every time an app needs a new build it requires that a developer steps away from what they is currently working on to produce a build. These interruptions are annoying for the developer and degrades the productivity of the team. Even more so when frequent builds are requested by a customer.

Every time a developer is forced to step away from a project to prouce a build he looses valuable development time and he looses his focus. But it doesn't just affect the development immediately, it also delays other processes further down the line. If interruptions are frequent to a key member of the team, the development can easily stall for periods of time.

2.3 The consequences of faulty builds

Due to the way the App Store on iOS works, bad deployments can have greater consequences than other deployments like web services. Once a build is ready for production it needs to be submitted to Apple for review. This process takes an average of 6-9 days but sometimes take as longs as 3 weeks. Once Apple has reviewed the app it is either released into production or rejected (in which case a new build must be submitted). After an app is released into production it cannot be rolled back. If an app with severe defects is rolled into production the only thing that can be done is to submit a new build and wait through app review once again. If the bug in production is big enough it could mean big losses of revenue.

A developer can request an expidted review from Apple if a problem is severe enough. Such reviews are granted only once or twice a year for each developer and must be requested with care. If the expidited review is granted the review process is shortened to 1-2 days. Losses in revenue can however still be big.

This aspect severely amplifies the problem that there is no streamlimed and/or documented process around building and depoying apps into production. It also contributes to a great deal of developer anxiety around deploying apps into production. No one wants to be the developer ultimately responsible for deploying a critical bug into production.

3 Literature Review

Suprinsingly enough there is not a big amount of litterature around the automation of the early development steps in the build process or deployments.

Osterweil [7] describes in his article *Processes are Software too* how humans have an innate facility for indirect problem solving through process specification. He suggest that we not only document our processes through process descriptions but also program them and use them directly as software.

In the most referenced article from the Continous Delivery field, *The deployment production line* [5], Humble et al. describes how to fully automate the testing and deployment process by using a multi-stage automated workflow. They note many of the same issues that are expirenced by Robocat like lack of documentation around deployments.

"[...] often the documentation is incomplete or out-of-date. In some cases, the information needed to deploy resides in the heads of several key members of staff who need to come together to perform the deployment." [5]

In the article Humble et al. enumerates 4 key principles and some practices they motivate in oder to adress the most common challenges facing automation of the build and deployment process.

Humble, together with Farley, 4 years later wrote the book *Continuous Delivery* [4]. In this book they describe all details needed to automate all parts of deployment including risk management strategies. More of the same symptoms described in the chapter 2 are found to be the same by Humble and Farley. The build processes take too long and are mostly performed entirely manually. Build and deployment processes are vastly undocumented and key developers have the necessary knowledge. Developers are nervous on release day because it is easy to make mistakes in the manual processes.

In the article *Continuous Delivery: Huge Benefits, but Challenges Too* [3], Chen covers the implementation of Continous Delivery techniques at a large organization, the huge benefits and challenges involved. By implementing Continous Delivery [3] experienced accelerated time to market with more frequent releases, faster and more usable user feedback, improved productivity and effiency among developers, more reliable releases and decreased anxiety around production releases.

4 Proposed Solution: Continous Delivery

Motivated by Osterweil [7] I have identified a solution that through process descriptions used as software aims to automate the build and deployment process.

Continous Delivery (CD) is a software engineering approach in which a software development team keeps producing software in short cycles and ensure that the software can be reliably released at any time. [3]

CD as a term is relatively new. The first appearances in litterature is by [1] in 2001. Since then the term has gained a lot of traction and has become increasingly popular among developers of web infrastructures where deployments can be delivered to customers in matters of minutes or even seconds. Although popular in web development, CD has not gotten the same amount of traction among mobile development where deployments take longer. The benefits remian largely the same. However it seems the tools are not as widespread or mature for this platform.

In section 4.4 I will give an example of a newer tool that has the required traction and maturaty required.ection

4.1 Automating Builds

This first part of CD is the process of automating the build phase of a software product. All parts of the process must be documented through the process of automation [7] until a binary build can be produced by a single click of a button. The implementation of this automation empowers all users of a team to generate new builds on the fly without any single team member with key knowledge to be present.

It does however take some effort to automate the build process. The cost of implementing the automation up front should be insiginificant relative to the gain of faster builds. With the build automation in place, the development team will spend less time on making builds thus being more productive. This is also one of the noted benefits by [3].

4.2 Automating Deployments

The second and most important part of implementing CD is to automate the process of delivering software. This part is one the four key principles presented by Humble et al. [5]. The steps involved in this part of the process varies somewhat depending on the software type and obviously how it is deployed to customer.

The practice is commonly used in web architectures because changes can be instantly deployed to users withing minutes or even seconds. In app development for Apple's platform this is not the case. As mentioned in section 2.3 deployments to Apple's platform on iOS has to go through a submission process at Apple. Each new build of an app must be reviewed by Apple in a process that takes times. The process of the submission process has a lot of steps where the app must be build and validated in Apple's infrastructure called iTunes Connect. After the app is submitted, screenshots, descriptions, links, promotional material, pricing schemes and more must be manually updated and verified. All of these steps can also be automated to minify the risk of human error.

Deployments of iOS apps are also perfomed elsehwere than Apple's platform. All weekly and ad-hoc builds of an app are deployed to a third-party service that allows beta testers to install the app on their device. This is the tool used for all internal testing and preview to the external customers. The process of deploying these builds is also manually but can be automated the same way it could be for iTunes Connect.

4.3 Continous Integration

CD is often coupled with another term called Continous Integration (CI). In short CI is the process of running tests on the produced software in all stages of the its lifecycle. Most commonly it is used such that all tests run whenever a new version of the code is checked in to source control. The tests will then run to make sure that everything builds properly and check that there we no regressions in the new code. It is recommended that CI is part of your CD pipeline [4] [3].

In the case of Robocat it could be an essential part of making sure that no serious bugs made it into production. This part of CD is probably the most time consuming since it requires a concious effort by all developers to create integration tests to cover the products. This part of the process could be slowly incorporated such that when a bug is found a new test is created to make sure it gets fixed and is not re-introduced into production.

At the very least some kind of automated tests should be in place to make sure the app runs as minimum validation before a deployment to production.

4.4 Using Fastlane

Fastlane is an Open Source suite of development tools to automate builds and deployments for iOS [6]. It provides all the tools needed for building, provisioning and deploying iOS apps right from the developers command line. The tools are configured as build processes in a ruby DSL. All developers on the team share the configuration and can all build, provision and deploy with the same tool. The tool can also be configured to run on a build server and run on a schedule for automated weekly and daily builds.

The tool has many different integrations for external services that may be used by different apps, like Crashreporting, beta builds through 3rd parties. Custom integrations can even be build and the entire source code is open source and small so it can be modified if necessary.

Fastlane neatly fits the needs of automation of builds and deployments needed by Robocat. It has also the necessary components to configure automation for all team members and automatic weekly builds. It is my recommendation to integrate this tool into the development workflow of the team.

5 Evaluation

The evaluation of the implementation of CD in the company would be mainly quantitative. In the following list I outline five identified metrics and how they should be recorded:

- **Number of hours spent on manual builds & deployments**: This is not a current metric, but it could be measured prior to implementation by self reporting from the developers.
- **Number of hours spent on configuring CD**: This metric should be implemented as a self-reporting mechanism by developers to figure out the cost/gain of implementing CD in terms of hours.
- **Number of beta & production deployments**: All deployments are recorded through a tracking interface that is also used for analytics data.
- **Number of regressions found in production**: All bugs found in production are recorded in a bug tracker. Bugs that were fixed earlier are already marked as regressions. Nothing should change about how this metric is quanified.
- **Anxiety around production deployments among developers**: This qualitative measure could be quantified by asking developers how anxious they feel about production deployments on scale. They should be asked both prior to implementation and some amount of time after.

Given that the all the proposed metrics are qunatifiable it helps that at least three of them have prior data that can be compared against to see if there is any measurable improvements.

5.1 Expected Outcome

Given the implementation of Continous Delivery in at least some of the actively maintained products I expect to see improvements to several metrics.

Initially there will an increased number of hours spent configuring and setting up CD in the selected products. After the setup there should be an immediate sharp drop-off in the number of hours spent on manual builds and deployments. This decrease in hours spent will increase the effectiveness and productivity of the team which should hopefully icnrease the number of beta and production deployments that gets produced.

Given the implementation of Continous Integration methods in some products that should be a decrease in the number of regressions found in production. This decrease will however only apply to products where new tests are created to prevent regressions.

Lastly given that all builds and deployments are performed automatically and the system performs well and as deployments are reliable developers should start feeling less anxious about production deployments.

6 Conclusion

Bibliography

[1] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mallor, S., Shwaber, K., and Sutherland, J. The Agile Manifesto. Tech. rep., The Agile Alliance, 2001.

[2] Bellomo, S., Ernst, N., Nord, R., and Kazman, R. Toward design decisions to enable deployability: Empirical study of three projects reaching for the continuous delivery holy grail. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on* (June 2014), 702–707.

[3] Chen, L. Continuous delivery: Huge benefits, but challenges too. *Software, IEEE 32*, 2 (Mar 2015), 50–54.

[4] Humble, J., and Farley, R. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, first ed. Fowler Series. Addison-Wesley, 2010.

[5] Humble, J., Read, C., and North, D. The deployment production line. In *Agile Conference, 2006* (July 2006), 6 pp.–118.

[6] Krause, F. Automating your ios release process using fastlane, March 2015.

[7] Osterweil, L. Software processes are software too, revisited: An invited talk on the most influential paper of icse 9. In *Software Engineering, 1997., Proceedings of the 1997 International Conference on* (May 1997), 540–548.