Continuous Deployment of Mobile Software at Facebook

Chuck Rossi
Facebook Inc.
1 Hacker Way
Menlo Park, CA 94025
chuckr@fb.com

Elisa Shibley
University of Michigan
2260 Hayward Street
Ann Arbor, MI 48109
eshibley@umich.edu

Shi Su
Carnegie Mellon University
PO Box 1
Moffett Field, CA 94035
shis@andrew.cmu.edu

Kent Beck
Facebook Inc.
1 Hacker Way
Menlo Park, CA 94025
kbeck@fb.com

Tony Savor
Facebook Inc.
1 Hacker Way
Menlo Park, CA 94025
tsavor@fb.com

Michael Stumm
University of Toronto
10 Kings College Rd
Toronto, Canada M8X 2A6
stumm@eecg.toronto.edu

ABSTRACT

Continuous deployment is the practice of releasing software to production as soon as it is ready. It is receiving widespread adoption in industry and has numerous perceived advantages including (i) lower risk due to smaller, more incremental changes, (ii) more rapid feedback from end users, and (iii) better ability to respond to threats such as security vulnerabilities.

The frequency of updates of mobile software has traditionally lagged the state of practice for cloud-based services. For cloud-based services, changes can be released almost immediately upon completion, whereas mobile versions can only be released periodically (e.g., in the case of iOS, every two weeks). A further complication with mobile software is that users can choose when and if to upgrade, which means that several different releases coexist in production. There are also hundreds of Android hardware variants, which increases the risk of having errors in the software being deployed.

Facebook has made significant progress in increasing the frequency of its mobile deployments. In fact, over a period of 4 years, the Android release has gone from a deployment every 8 weeks to a deployment every week. In this paper, we describe in detail the mobile deployment process at FB. We present our findings from an extensive analysis of software engineering metrics based on data collected over a period of 7 years. A key finding is that the frequency of deployment does not directly affect developer productivity or software quality. We argue that this finding is due to the fact that increasing the frequency of continuous deployment forces improved release and deployment automation, which in turn reduces developer workload. Additionally, the data we present shows that dog-fooding and obtaining feedback from alpha and beta customers is critical to maintaining release quality.

1. INTRODUCTION

Continuous deployment is the software engineering practice of deploying many small incremental software updates into production as soon as the updates are ready [1]. Continuous deployment is becoming increasingly widespread in the industry [2, 3, 4, 5, 6] and has numerous perceived advantages including (i) lower risk because of smaller, more incremental changes, (ii) more rapid feedback from end users, and (iii) improved ability to respond more quickly to threats such as security vulnerabilities.

In a previous paper, we described the continuous deployment process for *cloud-based software*; i.e., Web frontend and server-side SAAS software, at Facebook and another company[1] [1]. We presented both its implementation and our experiences operating with it. At Facebook each deployed software update involved, on average, 92 Lines of Code (LoC) that were added or modified, and each developer pushed 3.5 software updates into production per week on average. Given the size of Facebook's engineering team, this resulted in 1,000's of deployments into production each day.

At both companies discussed in [1], the developers were fully responsible for all aspects of their software updates. In particular, they were responsible for testing their own code, as there was (by design!) no separate testing team. The only requirement was peer code reviews on all code before it was pushed. However, there was considerable automated support for testing and quality control. As soon as a developer believed her software was ready, she would release the code and deploy it into production.

As demonstrated in the previous paper, cloud-based software environments allow for many small incremental deployments in part because the deployments do not inconvenience end-users — in fact, end users typically do not notice them. Moreover, the environments support a number of features to manage the risk of potentially deploying erroneous software, including blue-green deployments [7], feature flags, and dark launches [8]. As a worst case, stop-gap measure, it is always possible to "roll back" any recently deployed update at the push of a button, effectively undoing the deployment of the target software module by restoring the module back to its previous version.

[1]OANDA Corp.

In contrast with previous work, in this paper, we describe and analyze how continuous deployment is applied to mobile software at Facebook (FB). FB's mobile software is used by over a billion people each day. The key challenge in deploying mobile software is that it is not possible to deploy the software continuously in the same manner as for cloud-based services, due the following reasons:

1. The frequency of software updates may be limited because (*i*) software updates on mobile platforms are not entirely transparent to the end-user, and (*ii*) the time needed for app reviews to take place by the platform owner; e.g., Apple reviews for iOS apps.

2. Software cannot be deployed in increments one module at a time; rather all of the software has to be deployed as one large binary; this increases risk.

3. Risk mitigation actions are more limited; for example, hot-fixes and roll-backs are largely unacceptable and can be applied only in the rarest of circumstances as they involve the cooperation of the distributor of the software (e.g., Apple).

4. The end user can choose when to upgrade the mobile software (if at all), which implies that different versions of the software run at the same time and need to continue functioning correctly.

5. Many hardware variants — especially for Android — and multiple OS variants need to be supported simultaneously. The risk of each deployment is increased significantly because of the size of the Cartesian product: *app version* $\times$ (*OS type & version*) $\times$ *hardware platform*.

Given the above constraints, a compelling open question that arises is: *How close to "continuous" can one update and deploy mobile software?*

Facebook is striving to push the envelope to get as close as possible to continuous deployment of mobile software. The key strategy is to decouple software development and releases from actual deployment; the former occurs frequently in small increments, while the latter occurs only periodically. In particular, developers push their mobile software updates into a Master branch at the same frequency as with cloud software and in similarly-sized increments. The developers do this whenever they believe their software is ready for deployment. Then, periodically, a Release branch is cut from the Master branch — once a week for Android and once every two weeks for iOS. The code in that branch is tested extensively, fixed where necessary, and then deployed to the general public. The full process used at FB is described in detail in §2.

An important follow-on question is: *How does this mobile software deployment process affect development productivity and software quality compared to the productivity and quality achieved with the more continuous cloud software deployment process?*

We address this question in §5.2 where we show that productivity, when measured either in terms of LoC modified or added, or in terms of the number of commits per day, is comparable with what it is for software as a whole at FB. We show that mobile software development productivity remains constant even as the size of the mobile engineering team grows by a factor of 15X and even as the software matures and becomes more complex. In fact, we show how Android has gone from a deployment every 8 weeks to one every week over a period of 4 years, with no noticeable effect on programmer productivity.

Testing is particularly important for mobile apps given the limited options available for taking remedial actions when critical issues arise post-deployment (in contrast to the options available for cloud-based software). We describe many of the testing tools and processes used at FB for mobile code in §4. We show in §5.5 that the quality of FB mobile software is better than what it is on average for all FB-wide deployed software. And we show that the quality of the mobile software code does not worsen as (i) the size of the mobile software development team grows by a factor of 15, (ii) the mobile product becomes more mature and complex, and (iii) the release cycle is decreased. In fact, some of the metrics show that the quality improves over time.

Finally, we present additional findings from our analysis in §5. For example, we show that software updates pushed on the day of a Release branch cut are of lower quality on average. We also show that the higher the number of developers working on the same code file, the lower the software quality.

A significant aspect of our study is that the data we base our analysis on is extensive (§3). The data we analyzed covers the period from January, 2009 to May, 2016. It includes all of the commit logs from revision control systems, all app crashes that were captured, all issues reported by FB staff and customers, and all issues identified as critical during the release process.

The data prior to 2012 is noisier and less useful to draw meaningful conclusions from. Those were the early days of mobile software, and there were relatively few mobile-specific developers working on the code base. In 2012, FB's CEO announced the company's "Mobile First!" strategy to the public at the TechCrunch Disrupt conference in San Francisco. From that point on, the mobile development team grew significantly, and the data collected became more robust and meaningful. Hence, most of the data we present is for that period of time.

In summary, this paper makes the following specific contributions:

1. We are, to the best of our knowledge, the first to describe a process by which mobile software can be deployed in as continuous a fashion as possible. Specifically, we describe the process used at FB.

2. This is the first time FB's testing strategy for mobile code is described and evaluated.

3. We present a full analysis with respect to productivity and software quality based on data collected over a 7 year period as it relates to mobile software. In particular, we are able to show that fast release cycles do not negatively affect developer productivity or software quality.

4. We believe we are the first to show two compelling findings: (i) the number of developers modifying a file is inversely correlated to the quality of software in that file, and (ii) the software changes pushed on the day of the release cut from the Master branch are of lower quality than the files pushed on other days.

In the next section we describe the mobile release cycle used at FB for both Android and iOS. Section 3 presents the collected data that we used for our analysis. Section 4 describes the testing strategies used. Our analysis is presented in §5. Related work is covered in §6. We close with concluding remarks.

2. MOBILE RELEASE CYCLE

In this section we describe in detail the development and deployment processes and activities at FB for mobile apps. The overall architecture is shown in Figure 1. As seen in the figure, there are two classes of activities: development activities and deployment activities.

2.1 Development activities

First we describe development activities. There is effectively no difference between the development activities for mobile-based and cloud-based software at FB. The developer forks a local revision control system branch from the Master branch. Updates to the software are made to the local branch with frequent commits. After proper testing and when the developer believes her software update is ready for deployment, she *pushes*[2] the updates into the Master branch. As we show in §5, each developer pushes such updates into the Master branch 3-5 times a week, a timescale comparable to cloud-based software at FB.

A notable coding practice encouraged at FB is the use of a mechanism called Gatekeeper, that allows one to dynamically control the availability of features on the mobile device. In effect, this mechanism provides the ability to dynamically turn on or off a mobile-side feature from the server side, even on devices in client hands. Hence, if a newly deployed feature misbehaves, it can be turned off. This mechanism can also be used to incrementally turn on new features in a targeted way, say by targeting the OS, the specific OS version, the hardware device type, the specific hardware device model, the country, locale, etc. It can also be used for A/B testing, for example to test whether one feature gets more usage than an alternative.

2.2 Deployment activities

We now describe the lower half of the figure corresponding to deployment activities. FB has a *Release Engineering Team* (RelEng) that is responsible for these activities. The team includes project managers who are responsible for their target platform and who can make informed decisions as to which issues are launch-blocking or not.

Both iOS and Android deploy software on a fixed-date cycle in a pipelined fashion. We begin by describing the specific process for iOS and then describe how the process for Android differs.

iOS

For iOS, a Release branch is cut from the Master branch by RelEng at a two week cadence every other Sunday at 6pm. The release branch is first stabilized over a period of five days, during which issues are fixed or eliminated, and other polish and stabilization updates are applied. Some bugs are

[2]In this paper, we use the term *push* exclusively to refer to the pushing of the local development branch up into the Master branch; we do not use the term here to refer to the act of deploying software.

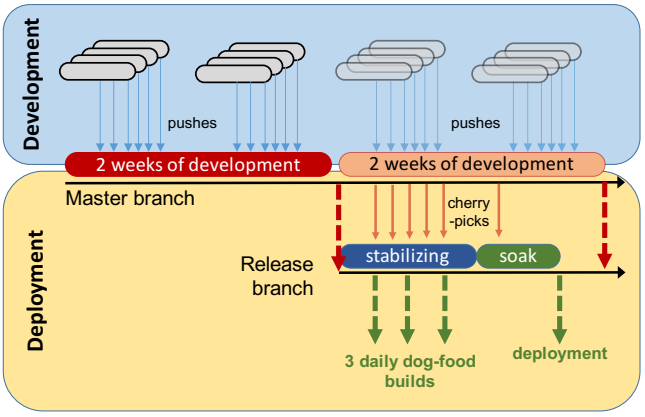

Figure 1: Release cycle for iOS. The process is pipelined with each stage taking two weeks (for iOS mobile code): the stabilization, soak, review, and deployment with the Release branch takes two week, while new software updates are continuously being pushed to the master branch during the same two weeks until the next Release branch is cut from the Master branch.

categorized by RelEng as *launch-blocking*; that is, bugs that would prevent making the app available to end-users and hence have to be fixed before deployment.[3] Finally, RelEng may decide to circumvent some issues by reverting parts of the code back to a previous version.

Developers are responsible for addressing any of the issues raised by RelEng, such as identified bugs. Each update generated by a developer that addresses issues raised by RelEng (regarding bug fixes, polishes, and stabilization) is only ever pushed into the Master branch. The developer must then make a request to RelEng to merge the update from the Master branch into the Release branch. In doing so, she must provide justification as to why the update should be merged into the Release branch.

In turn, RelEng "*cherry-picks*" (i.e. selects), at its discretion, those updates that will be merged into the Release branch. It does this carefully by taking various risk factors into account. Merge requests may be declined for any number of reasons. Two examples of merge requests that would likely be declined simply because the risk is deemed to be too high are: updates having too many dependencies (e.g., more than 5) or updates that make significant changes to core components like networking or display. Ultimately, the decision is a judgement call by RelEng involving a trade-off between risk and impact. For example, a polish update, which may have lower impact, is only accepted during the first 2-3 days and declined after that.

During the above described period of stabilization, the release branch is built and the resulting app is made available to FB internal users as "*dog food*." This occurs three times a day. After five days of stabilization efforts, the code is frozen and a "*soak*" period of three days begins. During the soak period, only fixes for launch-blocking issues are accepted.

Finally, on the second Monday after the Release branch cut, the software is shipped to Apple for review and subsequently deployed sometime later that week.

The overall timeline, shown in Fig. 1, is pipelined: new updates are being pushed into the Master branch concurrently

[3]Note that despite the name, launch-blocking issues never get to the point where they actually block a deployment.

with the stabilization of the current Release branch. Only some of these new updates are cherry-picked to be merged into the Release branch by RelEng. If, for whatever reason, this two week timeline depicted in the right half of the figure cannot be met safely, then the deployment is halted and all changes are deferred to the next regularly scheduled deployment.[4]

Android

The Android release cycle is almost identical, except that it is compressed: branches are cut at a 1-week cadence rather than a 2-week cadence. On Thursday, a code freeze occurs and merge requests are only accepted for launch-blocking issues.

On Monday mornings (one week after a branch cut), a slow rollout of the app to the Play Store occurs: typically first only 20%, then 50%, and finally 100% of the population will obtain access to the new version over the following three days, with periodic stability checks after every increase in the rollout percentage.

For Android, FB also releases Alpha and Beta versions of the app. Alpha is shipped from the Master branch once a day. It is made available through the Google Play Store to a small fraction of external users consisting of several 10,000 users. Beta is shipped from the Release branch once a day and made available to a larger fraction of external users (around 3 million).

Release Engineering Team

The Release Engineering Team plays a critical role in the above process. Despite its importance, this team is perhaps smaller than one might assume — it has fewer than 10 members. The team can be kept small, in part because of the tools it has at its disposal and the degree of automation in the process, and in part because of the partnership it has built up over many years with the development teams.

Senior product developers are continuously being seconded to temporarily work for the Release Engineering Team for one release every two months or so. A key advantage of this arrangement is that it creates a collaborative partnership between RelEng and the development teams. The developers become educated on what RelEng does, what its process flows are, what its problems and pain-points are, etc. The knowledge accrued by the developers during their stint with RelEng is then permeated back to the rest of their development teams when they return.

A further advantage of this arrangement is that developers, after having observed inefficiencies in the release process, in some cases later develop tools for RelEng that allow the process to be further simplified and automated.

3. DATA COLLECTED

Facebook collects a significant amount of data related to each release and each deployment. It retains all of this data. In this section, we describe the data sets we used for the analysis we present in §5.

- **Revision-control system records.** This dataset has a record of each commit and push, the date, the size of the diff being committed (as measured in LoC), the developer who issued the diff, etc.

[4]This has occurred only once in the last 9 months.

- **Crash database**. A crash report is automatically sent to FB whenever a FB app crashes. The crash rate is a direct indicator of app quality, which is why these crash reports are carefully monitored and used to determine the healthiness of a release. Bots automatically categorize reliability issues into crashes, soft errors, hung systems, etc. Whenever a given crash rate metric is higher than a specified threshold, then the crash is automatically submitted as a launch blocker to the *tasks database* (described below). Additionally, the stack trace of the crashed app is automatically compared against recent changes recorded in the revision control system, and if any instruction on the stack trace is close to changed code, then the developer who made the change is informed in an automated way. We aggregated crash reports by date and app version.

- **Flytrap database**. This database contains issue reports submitted by (internal or external) users. Users can submit issues from a "Report a Problem" component on the app which can be obtained from a pull-down menu or by shaking the device. Alternatively, they can fill out a *Help Center* "Contact Form" on a FB Website. Employee-generated flytraps are automatically entered into the *tasks database* (described below). User-generated flytraps are not automatically entered into the tasks database because many users submit unimportant issues, new feature requests, various improvement suggestions, and sometimes just junk. In aggregate, the Flytrap database is rather noisy. For this reason, user-generated flytraps are entered into the tasks database automatically only if a bot is able to identify an issue reported by a sufficient number of users exceeding a given threshold. We aggregated Flytrap issues by date and app version.

- **Tasks database**. This database is a core component of the release engineering management system. It records each task that is to be implemented and each issue that needs to be addressed. Tasks are entered by humans and bots; the number of bot-created tasks is increasing rapidly. Tasks related to a mobile release include:
  - *launch-blocking issues*: critical issues marked by RelEng which need to be addressed before deployment.
  - *crashbot issues*: tasks created by a bot when a crash affects more than a threshold number of users.
  - *flytrap issues*: tasks created by a bot for employee-reported issues and when more than a threshold number of flytrap reports identify the same issue.
  - *test fails*: tasks created for failed autotests (see §4).

- **Cherry-pick database**. All cherry-pick requests are recorded, and those accepted by RelEng are marked as such. We use the number of cherry-picks as a proxy metric of software quality.

- **Production issues database**. All errors identified in production code are recorded in a separate database and each is categorized by severity: *critical*, *medium*

priority, and *low priority*. The recorded errors are entered by humans when they believe a production error needs to be fixed (even if low priority). We use the number of issues in this database as one measure of quality of the software that was deployed to production.

- **Daily Active People** (DAP). This database records the number of users using FB apps. We aggregated the numbers by data and app version. We use this data in certain cases to normalize crash and flytrap quality indicators.
- **Boiler room**. This database contains information about each mobile deployment, including the deployment date, the deployment status, the release branch from which the build was made, the cut date of the release being deployed, the versions of the alpha/beta/-production builds, etc.
- **Staff database**. This data identifies how many developers were working on mobile software per day. The information is extracted from the commit logs: if a developer committed mobile code in the previous two weeks then she is deemed have been working on mobile software.
- **Source and configuration code**. We used information from this source to identify how and when automated jobs are schedule, thresholds bots use to determine when to create a task, etc.

4. TESTING

Testing is particularly important for mobile apps for the following reasons:

- Thousands of updates are made to mobile software each week.
- There are hundreds of device and OS version combinations the software has to run on.
- The options available for taking remedial actions when critical issues arise post-deployment are limited.

As one might expect, FB applies numerous types of tests, including unit tests, static analysis tests, integration tests, screen layout tests, performance tests, build tests, as well as manual tests.

Tools and automation play a key role, supported in part by hundreds of developers devoted to developing tools. Most of the tests are run in an automated fashion. When a regression is detected, an attempt is made to automatically tie it back to specific code changes that were made recently, and an email is automatically sent to the developer responsible for that particular change.

Before we describe some of the tests and when they are used, we briefly describe the testing infrastructure available at FB as well as the principles that guide the FB testing strategies.

Testing Infrastructure

Thousands of compute nodes are dedicated for testing mobile software. Both white-box and black-box testing strategies are applied by running tests on simulated and emulated environments.

For testing on real hardware, FB runs a mobile device lab located in the Prineville data-center [9]. The mobile lab is primarily used to test for performance regressions, with a primary focus on app speed, memory usage, and battery efficiency.

The mobile lab contains electromagnetically isolated racks. Each rack contains multiple nodes that are connected to the iOS and Android devices that will be used for the target tests. The nodes install, test and uninstall target software. Chef, a configuration management tool [10], is used to configure the devices. A wireless access point in the rack supports device Wi-Fi communication. The devices are set up in the racks so that their screens can be captured by cameras. Engineers can access the cameras remotely to observe how each phone reacts to code changes. Thus, the lab effectively offers testing Infrastructure-as-a-Service.

Testing Principles at FB

FB's testing strategy encompasses the following principles:

1. **Coverage**. Testing is done as extensively as possible. Almost every test ever written that is still useful is run as frequently as it makes sense to do so.
2. **Responsive**. The faster a regression is caught, the easier it is to deal with; quickly caught issues are easier to address by the developers because the code and code structure are still top of mind. For instance, parallel builds are used so as to be able to provide feedback more quickly. The objective is to be able to provide the developer with the results from smoke-tests within 10 minutes of her actions.
3. **Quality**. Tests should identify issues with surgical precision. False positives and false negatives need to be minimized. This is important not only to minimize the time developers spend on chasing false alarms, but also to prevent test results from being ignored over time.
4. **Automation**. Tests are automated as much as possible. This makes them repeatable, and it ensures the tests are run on a regular basis. An additional aspect of automation has proved to be useful: automatically identifying the developer that caused a regression with his code changes so that he can be informed immediately with specific details on the detected issue and the code that might have caused the issue. This is done by comparing the location in the code likely to have caused the detected issue with recently applied code changes. Much effort has gone into developing this capability since it is effective only when the quality is high.
5. **Prioritization**. Testing requires a great deal of computing resources if it is to be exhaustive and responsive at the same time. Since resources will invariably be limited, a prioritized testing strategy is imperative. For example, when a change is pushed to the Master branch, integration tests are only done on those portions of the app that might be affected by the changes being pushed, instead of running the full test-suite. This makes it possible to provide key test results to the developer more quickly. Complete integration tests are run every few hours on the Master and Release branches.

Unit tests:	These white-box tests primarily verify the logic of target units. Limited unit tests are run manually on the development environments, XCode or Android Studio, using tools such as XCTest [11], JUnit [12], or Robolectric [13]. More extensive as well as automated unit tests are run on server-based simulators.
Static analysis:	This analysis identifies potential null-pointer dereferences, resource leaks, and memory leaks. Because exceptions are often the root cause of resource leaks, the tool is careful to identify where these particular leaks might occur.
Build tests:	These tests determine whether the code builds properly.
Snapshot tests:	These tests generate images of screen views and components, which are then compared, pixel by pixel, to previous snapshot versions [14, 15].
Integration tests:	These standard (blackbox) regression tests test key features and key flows of the app. They typically run on simulators and are not necessarily device specific. Moreover they employ degrees of scope: "*smoke tests*" primarily target specific changes to the code (diffs) or target high-level functionality; long-tail integration tests run the full suite of regression tests and run against a live server so that they also cover client-server integration.
Performance tests:	These tests run at the mobile lab to triage performance and resource usage regressions as described above.
Capacity tests:	These tests verify that the app does not exceed various specified capacity limits.
Conformance tests:	These tests verify that the app conforms to various requirements.

Table 1: Range of tests performed on mobile software.

Tests and when they are run

Table 1 lists some of the types of testing conducted on the mobile software. These tests are run in all phases of the development and deployment cycle.

Pre-Push Testing. A developer will frequently run unit tests on her PC/laptop while developing code. Given the size of the apps and the limited power of the development PCs and laptops, more extensive unit tests will be run on servers in simulation environments. The developer can also manually invoke any of the other tests listed in the table, with static analysis and some integration testing being the most common ones invoked. Finally, a subtle point, while there is no separate testing team, a code review is required before any code can be pushed to the Master branch.

On Push. When the developer believes her changes are complete and work correctly, she initiates a push of her changes to the Master branch. Before the actual push occurs, a number of tests are run automatically to determine whether the push should be blocked. These include standard unit tests as well as a number of smoke tests that verify that heavily used features and their key flows work correctly. Moreover, a test is run to ensure a build with the changes works correctly. However, since a full build is time and resource intensive, the build test here only tests dependencies a few levels deep.

If all the tests pass, then the changes are pushed onto the Master branch. The merging process may identify conflicts, in which case the developer is alerted so she can address them.

Continuous testing on Master and Release branch. All of the tests are run continuously (every few hours) on both the Master and the Release branch. The most important of these are the full build tests, integration regression tests, and performance tests in the mobile device lab.

As mentioned in §2, alpha versions are built from the Master branch (twice a day for iOS and once a day for Android) and beta versions are built from the Release branch three times a day. Release of the alpha version of software is blocked if a certain percentage of tests fails. This happens very rarely.

Manual testing

A contracted manual testing team (of about 100) is used to test the mobile apps. They do various smoke tests and edge-case tests to ensure the apps behave as expected. The testing team is primarily used to test new features for which automated tests have not yet been created. Finally, they are responsible for UI testing after language translations have been added to ascertain the quality of the look and feel is high.

5. ANALYSIS

5.1 Methodology

Our quantitative analysis of the FB mobile software engineering effort is based on the data described in §3. The data extracted from those data sets was cleaned, interpreted, cross-checked and the conclusions drawn were presented to FB-internal subject-matter experts for confirmation. We describe how the data sets were cleaned in the figure captions where relevant.

Some of the data sets start from January 2009. All continue until 2016. The metrics collected between 2009 and 2012 are rather noisy, to the point where it is difficult to infer useful insights. While the first several graphs in this section depict data from 2009 onwards, later graphs focus on the period from 2012 to 2016. In 2012, fewer than 100 developers worked on mobile code, while by 2016, over 1,500 developers did. Over one billion users use the software analysed here on a daily basis.

5.2 Developer Productivity

Figure 2 depicts developer productivity as measured by lines of code (LOC) that are ultimately pushed and deployed on average per developer and per day. The figure shows that developers produce on average 70 LoC per day for both Android and iOS. This is in line with the 64.5 LoC produced per developer per day company-wide across all software.[5]

[5]Code for backend services is produced more slowly than the average.

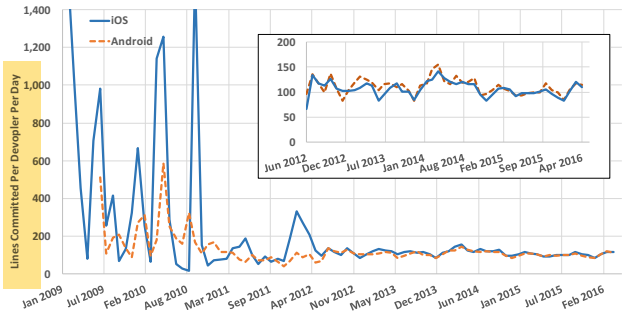

Figure 2: Lines of Code pushed per developer per day as averaged each month. The insert shows the same data from mid 2012 onwards at a larger scale. (Pushes from bots were removed from the data set, as were pushes that changed more than 2,000 lines to avoid including third-party software packages, and directories being moved or deleted.)

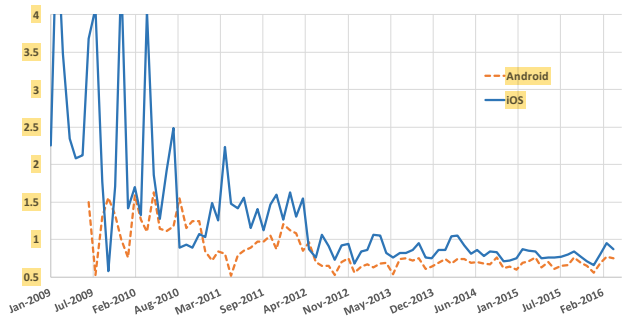

Figure 3: Number of pushes per developer per day as averaged each month. (Pushes from bots were removed from the data set.)

Figure 3 depicts developer productivity as measured by number of pushes per day on average for both Android and iOS. Over the last two years, the average number of pushes per developer per day was 0.7 and 0.8 for Android and iOS, respectively. When averaged over all company-wide software, it is 0.7 pushes per developer per day; hence, iOS has slightly more frequent pushes. While Android has fewer pushes than iOS per developer per day, it has the same number of LoC generated per developer per day than iOS, which implies that Android pushes will be slightly larger than the iOS pushes.

More interesting than the absolute numbers is how productivity changes over time as a function of the number of developers producing code for the mobile platforms — Figure 4 depicts how the number of these developers changes over time:

Finding 1: *Productivity remains constant even as the number of engineers working on the code base grows by a factor of 15. This is the case for both Android and iOS developers, and whether measured by LoC pushed or number of pushes.*

One hypothesis for why this is the case might be that the inefficiencies caused by larger developer groups and a larger, more complex code base are offset, in part, by (*i*) tooling and automation improvements over time, and (*ii*) the agile development, continuous release, and deployment processes being used. One can, however, definitively conclude:

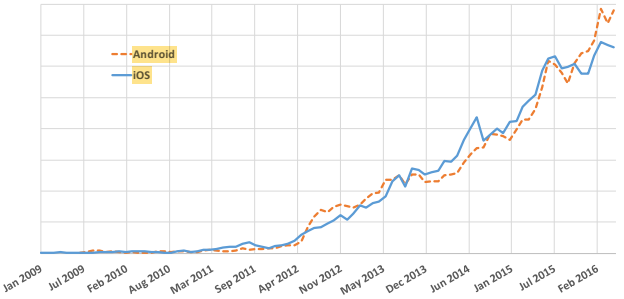

Figure 4: Growth of the size of the Android and iOS development teams. The y-axis has been deliberately left out so as not to divulge proprietary information.

Finding 2: *The continuous deployment processes being used for mobile software at FB does not negatively affect productivity even as the development organization size scales significantly.*

5.3 Quality

At FB, each issue with production code that someone believes needs to be addressed is registered in the *Production Issues Database* and categorized by severity: (i) critical, where the issue needs to be addressed immediately at high priority, (ii) medium-priority, and (iii) low-priority. We use the number of issues found in production code as one measure of production software quality.

Figure 5 depicts the number of issues for each severity level as a function of the number of pushes per month for Android and iOS. The (red) triangles represent critical errors; they have been almost constant since 2012, with each release either having 0 or 1 critical issue.

Finding 3: *The number of critical issues arising from deployments is almost constant regardless of the number of deployments.*

We surmise that the number of critical issues is low for two reasons. Firstly, critical errors are more likely to be detected in testing. Secondly, the company takes critical issues seriously, so after each occurrence, a standing review meeting is used meetings to understand the root cause of the issue and to determine how similar issues of the same type can be avoided in future deployments.

Unsurprisingly, the number of medium-priority and low-priority issues grows linearly with the number of pushes, *albeit with very low slopes*: for medium-priority issues the slope is 0.0003 and 0.00026 for Android and iOS respectively, and for low priority issues the slope is 0.0012 and 0.00097 for Android and iOS respectively. The slope for both medium- and low priority issues is higher for Android than for iOS; the relatively large number of Android hardware platforms may be one potential explanation. Interestingly, company-wide, across all software, the two corresponding slopes are 0.0061 and 0.0013, respectively, which are both steeper than for the mobile code alone. Overall, we argue that these numbers are evidence that the testing strategies applied to mobile software is effective.

Figure 6 shows the number of launch-blocking issues and number of cherry-picks per deployment for Android and iOS.

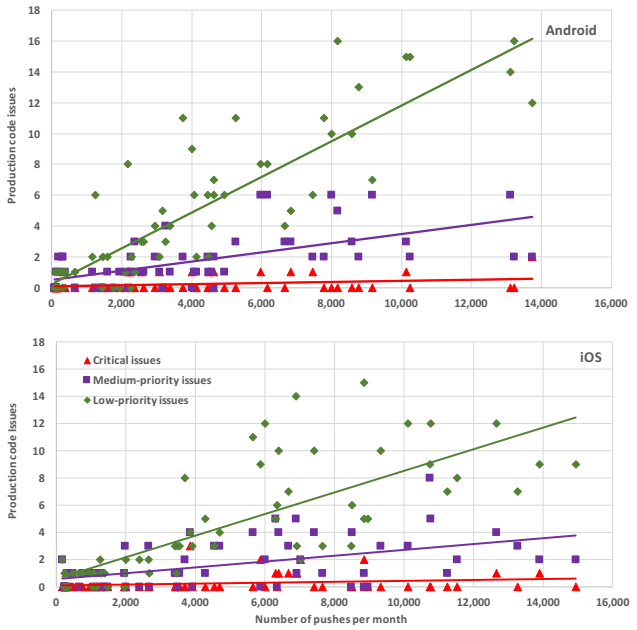

Figure 5: Number of recorded issues with Android and iOS production code as a function of the number of pushes per month for the period between March, 2011 and May, 2016. The top graph is for Android; the bottom for iOS. Note that the scale of the x- and y-axes are very different: all of the linear trendlines have a slope less than 0.0012.

To allow for direct comparison, the numbers are normalized by the length of the release cycle, under the assumption that a release cycle twice as long will have twice as many pushes and twice as many LoC modified. That is, each data point in the graph depicts for each release cycle number of cherry-picks and launch-blockers recorded per day on average. Overall, the number of launch-blockers and the number of cherry-picks seem to oscillate somewhat over time, but with no particular trend: the number of cherry-picks seem to mostly stay within a range of 5–25 for Android and within a range of 5–15 for iOS.

Finding 4: *The length of the deployment cycle does not significantly affect the relative number of cherry-picks and launch-blockers as it decreases from 4 weeks to 2 weeks (and then to 1 week for Android).*

From this we conclude that the length of the deployment cycle does not directly affect the quality of the software produced, with the described continuous deployment process. In particular, there is no discontinuity in the curves at the points where the frequency of deployments changes.

One should note that the number of launch-blocking issues is impacted in at least two ways. First whether an issue is declared to be launch-blocking or not is subjective. RelEng believes that they have become more stringent over time on deciding what constitutes a launch-blocker. This is somewhat intuitive: as more people use the mobile app, standards tend to increase. Secondly, a large number of testing tools were developed in 2015 and 2016. These tools should lead to an improvement in code quality, and hence reduce the number of launch-blocking issues.

Figure 7 shows the crash rates experienced by end users

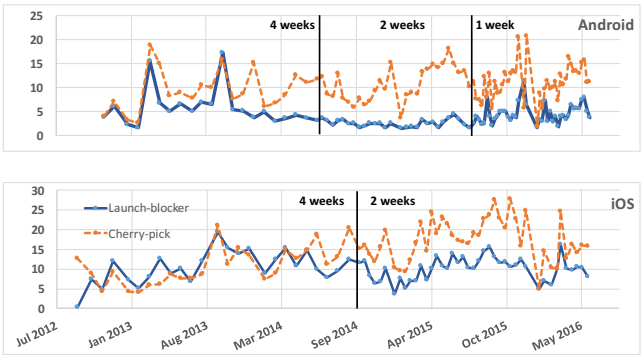

Figure 6: Number of launch-blocking issues (blue solid lines) and number of cherry-picks (orange dashed lines) per deployment for Android (top) and iOS (bottom), normalized by length of release cycle. Each data point in the graph depicts for each release cycle the number of cherry-picks and launch-blockers recorded per day on average. The vertical straight lines show where release cycle was reduced, first from a 4 week cadence to a two week cadence, and then for Android to a one week cadence.

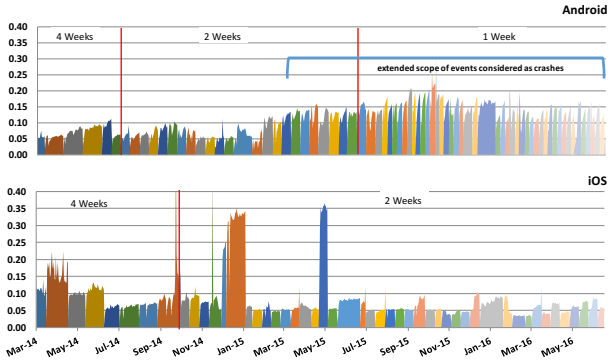

Figure 7: Normalized daily crash rate for Android (top graph) and iOS (bottom graph): proportion of end users experiencing a crash relative to the total number of active users normalized to a constant. The vertical straight lines show where release cycle was reduced, first from a 4 week cadence to a two week cadence, and then for Android to a one week cadence. The curve is normalized to the same constant in order to hide the absolute values but to allow a comparison.

per day, normalized to a constant in order to hide the absolute values. The different colors of the bars represent different releases: the color swaths become more narrow each time the deployment cycle is reduced (but may be wider around holidays).

Finding 5: *The shortening of the release cycle does not appear to affect the crash rate.*

The number of Android crashes increased in the first half of 2015; there are two reasons for this. First, the scope of what was considered to be a crash was extended to also include Java crashes when running a FB app and apps that were no longer responding. Second, a set of third-party software libraries were introduced that increased the crash rate; some of the offending software was removed in the second half of the year.

Internal FB employees play a critical role in helping to test the mobile software through dog-fooding. The iOS platform

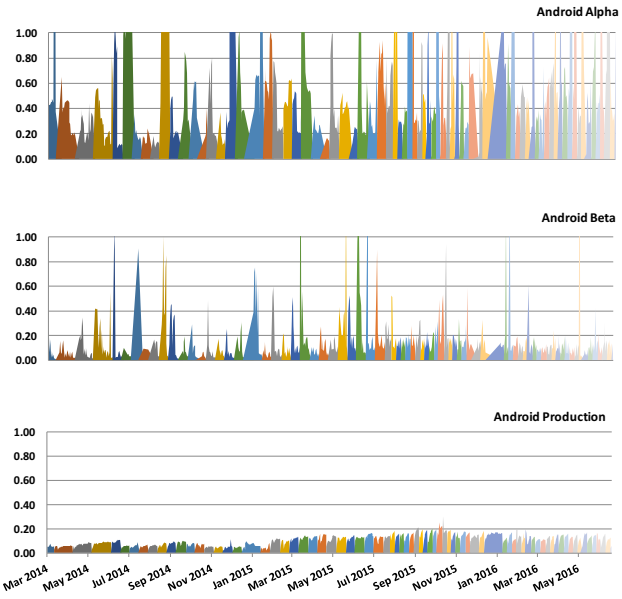

Figure 8: Normalized daily crash rate for alpha (top graph), beta (middle graph), and production (bottom graph) versions of Android FB apps: Proportion of end users experiencing a crash relative to the total number of active users normalized to a constant. Note that the bottom graph here is identical to the top graph of Fig. 7, just with a different scale for the y-axis. These graphs show the importance of having users testing alpha and beta versions of the software to improve software quality.

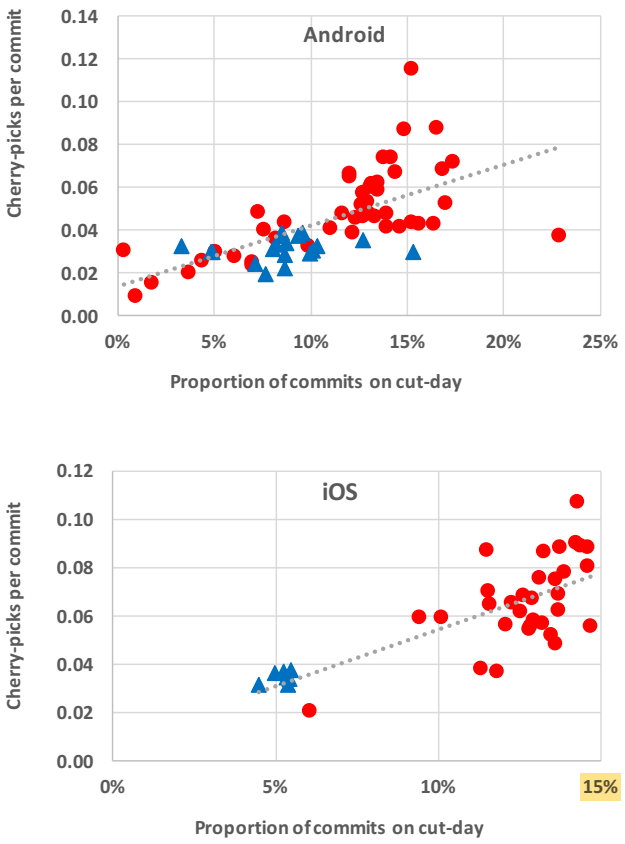

Figure 9: Cherry-picks per push as a function of the proportion of pushes on the release cutoff day. Thursday cutoffs are shown as red circles; Sunday cutoffs as blue triangles. For Android, only releases after June, 2014 are included, which is when the release cycle switched to a two week cadence. For iOS, only releases after September, 2014 are included, which is when the release cycle switched to a two week cadence. Releases up to the end of May, 2016 are shown for both Android and iOS.

gets far better dog-food coverage internally, because employees tend to skew towards iOS (over Android). This is particularly problematic, since there is such a wide variety of hardware that Android runs on. For this reason, Android relies more heavily on alpha and beta releases of its software. Figure 8 shows the effectiveness of the alpha releases with respect to crashes. A significant decrease in the crash rate is visible when going from alpha to beta; crash rate spikes that occur occasionally with the beta version of the software no longer appear in the production version of the software. The crash rate for alpha versions of the software is roughly 10X higher than for production versions.

5.4 Deadline Effects

There is a clear increase in the number of pushes to the Master branch on the day the Release branch is cut. This may indicate that developers rush to get their code in before the deadline, which begs the question of whether there is a decrease in quality of code submitted on the day of release cut.

Figure 9 depicts the number of cherry-picks per push as a function of the proportion of pushes on the day of the Release branch cut. The number of cherry-picks is used as a proxy for software quality. There is a correlation between the percentage of pushes made on the day of release cut and the number of cherry-picks per day: as the number of pushes made on the day of a release cut increases, so do the number of needed cherry-picks.

In absolute terms, Android has a higher proportion of pushes on the cut day. This is primarily because many of the datapoints represent releases that occurred at a one-week cadence, while all iOS datapoints represent releases at a two-week cadence. That is, if all of the Android pushes are evenly distributed across the seven days, one would expect each day to have 14% of all pushes; on the other hand, if the iOS pushes are evenly distributed over the 14 days, then one would expect each day to have 7% of all pushes.

Finding 6: *Software pushed on the day of the deadline is of lower quality.*

Our hypothesis is that the software pushed on the cut day is likely rushed and that rushed software will have lower quality.

When, normalized, Android has a lower proportion of pushes on the cut day, and because of this a smaller proportion of cherry-picks. An intuitive explanation for this goes as follows: if a developer believes her update will not make the cut, they will only have to wait one week and hence do not force a push. However, if the next cut is further out, then some developers rather force a push than to have to wait the longer period of time before their updates get deployed.

Another interesting observation is that moving the cut

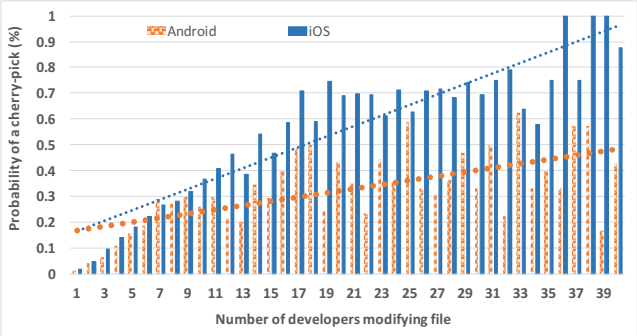

Figure 10: Proportion of files cherry-picked when modified by n developers for $n = 1..40$.

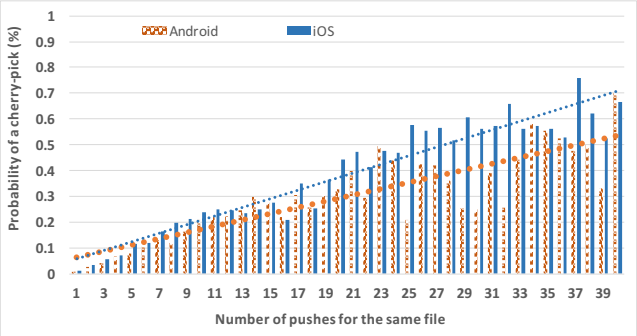

Figure 11: Proportion of files cherry-picked when committed n times for $n = 1..40$.

date to a weekend leads to improved quality, in part because there is a lower tendency to rush to push updates when the cut date is on a weekend. The cut date was moved from a Thursday to a Sunday in August, 2015 for Android and in February, 2016 for iOS. While the average number of pushes on the weekend increased from 0.3 pushes per developer per day to 0.5 for iOS, and from 0.175 to 0.5 for Android (not shown in any figure),[6] the number of weekend pushes is still significantly lower than the number of weekday pushes. As Figure 9 shows, the lower number of pushes for weekend cut days, correlates with a lower number of cherry picks for those releases, a sign of improved quality.

5.5 Factors Affecting Software Quality

An interesting question what factors affect software quality. Above we showed that that software pushed on the cutoff day will be of lower quality. But we also showed that the length of the release cycle, the size of the engineering team, and the number of pushes do not seem to affect software quality directly. We have also not been able to find any correlation between software quality and

- the size of the change of each push (as measured in LoC modified)
- the size of the file being changed

We did, however, find two factors that do appear to affect software quality, although both factors are highly correlated. First, we found that as the number of developers that commit code to a file of a release increases, so does the probability of the file being cherry-picked. This is shown in Figure 10

Finding 7: *The more developers involved in modifying a file for a release, the lower the software quality of the file.*

Some files are modified by a surprisingly large number of developers. An example scenario where this happens naturally are very large switch statements, where the code associated with individual case statements are modified by different developers.

The above finding suggests that files needing modification by multiple developers should perhaps be split up to improve quality.

[6]Moving the cut date to a Sunday did not affect overall developer productivity.

A second factor we found that appears to affect software quality is the number of times a file is pushed between cuts. A high number of pushes leads to a higher probability of the file being cherry-picked (Figure 11). However, note that a high number of developers modifying a file implies a high number of pushes for the file.

6. RELATED WORK

Continuous deployment is a natural extension of the progression from agile software development [16, 17, 18, 19] to continuous integration [20, 21, 22] to continuous delivery [7, 23]. *Agile development*, where software is developed iteratively with cycles as short as half a day, started in the late 1990's and is now used in some form in many if not most organizations. *Continuous Integration* is the practice in which software updates are integrated at the end of each software cycle; this allows integration to be verified by an automated build and automated tests to detect integration errors as quickly as possible. *Continuous Delivery* goes one step further and ensures that the software is built and tested in such a way that the software can be released to production at any time. Continuous deployment deploys each software change to production as soon as it is ready.

Related developments include lean software development [24], kanban [25], and kaizan [26]. DevOps is a movement that emerged from combining roles and tools from both the development and operations sides of the business [27, 28].

In a recent study, McIlroy et al. showed that of 10,713 mobile apps in the Google Play store (the top 400 free apps at the start of 2013 in each of the 30 categories in the Google Play store), 1% were deployed more frequently than once a week, and 14% were deployed at least once every two weeks [29]. While the study is based on a relatively small time window, it suggests that continuous deployment is actively being pursued by many organizations.

However, the literature contains only limited explicit references to continuous deployment for mobile software. Klepper et al. extended their agile process model Rugby to accommodate mobile applications [30]. Etsy, a leader in continuous deployment, has presented how they do continuous integration for their mobile apps in talks [31]; effectively, they deploy mobile software out to their employees on a daily basis [32].

With respect to testing, Kamei et al. evaluate just-in-time quality assurance [33], which is similar to the strat-

egy describe in §4. Gao et al. describe mobile Testing-As-A-Service infrastructure for mobile software [34]. Amazon's Appthwack offers a mobile device farm for that purpose [35].

7. CONCLUDING REMARKS

This paper described continuous deployment for mobile applications at Facebook. Shorter release cycles have numerous advantages including quicker time to market and better ability to respond to product feedback from users, among others. The release cycle was shortened from 8 weeks to 1 week for Android over a period of four years. We described learnings throughout this period.

Shortening the release cycle forces the organization to improve tools, automate testing and procedures. Our observations is that over the four year period when the Android release cycle was reduced from 8 weeks to 1 week, these efforts to improve tools and increase automation represented the bulk of the work to make continuous deployment successful. The data collected show that tools and process improvements increased quality and permitted faster release cycles. However, continuous deployment itself did not offer any productivity or quality improvements.

Releasing software for mobile platforms is more difficult, because it is not possible to roll back/forward in the event of a problem as it is with cloud software. Feature flags are used to enable or disable functionality remotely so that a buggy feature doesn't require a new release. Alpha and beta programs were extremely effective in providing bug reports that reduce problems, provided that reporting of such issues was automated.

Analysis revealed several patterns leading to lower quality software. One was cramming for a release on a branch date: software committed on the day a release branch is cut was found to be of lower quality. Shortening the release cycle improved this, because developers were less likely to rush their software pushes, knowing that another release was upcoming shortly. Another pattern that led to lower-quality software was having too many developers work on the same file concurrently: the number of problems per file was found to increase proportionally to the number of developers who modified the file.

8. ACKNOWLEDGEMENTS

We sincerely thank Laurent Charignon, Scott Chou, Amanda Donohue, Ben Holcomb, David Mortenson, Bryan O'Sullivan, James Pearce, Anne Ruggirello, Damien Sereni and Chip Turner for their valuable contributions, suggestions and feedback. Their input greatly improved this paper.

9. REFERENCES

[1] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm, "Continuous deployment at Facebook and OANDA," in *Proc. 38th Intl. Conf. on Software Engineering, (ICSE16)*, May 2016, pp. 21–30.

[2] C. Parnin, E. Helms, C. Atlee, H. Bought, M. Ghattas, A. Glover, J. Holman, J. Micco, B. Murphy, T. Savor, M. Stumm, S. Whitaker, , and L. Williams, "The top 10 adages in continuous deployment," *IEEE Software, accepted for publication*, 2016.

[3] J. Allspaw and P. Hammond, "10 deploys per day — Dev and Ops cooperation at Flickr," 2009, slides. [Online]. Available: http://www.slideshare.net/jallspaw/10-deploys-per-day-dev-and-ops-cooperation-at-flickr

[4] M. Brittain, "Continuous deployment: The dirty details," Jan. 2013. [Online]. Available: http://www.slideshare.net/mikebrittain/mbrittain-continuous-deploymentalm3public

[5] T. Schneider, "In praise of continuous deployment: The Wordpress.com story," 2012, 16 deployments a day. [Online]. Available: http://toni.org/2010/05/19/in-praise-of-continuous-deployment-the-wordpress-com-story/

[6] G. Gruver, M. Young, and P. Fulgham, *A Practical Approach to Large-scale Agile Development — How HP Transformed LaserJet FutureSmart Firmware*. Addison-Wesley, 2013.

[7] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010.

[8] D. Feitelson, E. Frachtenberg, and K. Beck, "Development and deployment at Facebook," *IEEE Internet Computing*, vol. 17, no. 4, pp. 8–17, 2013.

[9] A. Reversat, "The mobile device lab at the Prineville data center," https://code.facebook.com/posts/300815046928882/the-mobile-device-lab-at-the-prineville-data-center, Jul. 2016.

[10] https://www.chef.io/chef/.

[11] https://developer.apple.com/reference/xctest.

[12] http://junit.org/.

[13] http://robolectric.org.

[14] P. Steinberger, "Running UI tests on iOS with ludicrous speed," https://pspdfkit.com/blog/2016/running-ui-tests-with-ludicrous-speed/, Apr. 2016.

[15] https://github.com/facebook/ios-snapshot-test-case/.

[16] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[17] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Manifesto for agile software development," 2001. [Online]. Available: http://www.agilemanifesto.org/

[18] A. Cockburn, *Agile Software Development*. Addison Wesley Longman, 2002.

[19] A. Cockburn and L. Williams, "Agile software development: It's about feedback and change," *Computer*, vol. 36, no. 6, pp. 39–43, 2003.

[20] L. Williams, "Agile software development methedologies and practices," in *Advances in Computers*, vol. 80, 2010, pp. 1–44.

[21] M. Fowler and M. Foemmel, "Continuous integration," *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, p. 122, 2006.

[22] P. M. Duvall, *Continuous Integration*. Pearson Education India, 2007.

[23] L. Chen, "Continuous delivery: Huge benefits, but challenges too," *IEEE Software*, vol. 32, no. 2, pp. 50–54, 2015.

[24] M. Poppendieck and M. A. Cusumano, "Lean software development: A tutorial," *IEEE software*, vol. 29, no. 5, pp. 26–32, 2012.

[25] D. J. Anderson, *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press, 2010.

[26] M. Poppendeick and T. Poppendeick, *Lean Software Development*. Addison Wesley, 2002.

[27] M. Httermann, *DevOps for developers*. Apress, 2012.

[28] J. Roche, "Adopting DevOps practices in quality assurance," *Communications of the ACM*, vol. 56, no. 11, pp. 38–43, 2013.

[29] S. McIlroy, N. Ali, and A. E. Hassan, "Fresh apps: an empirical study of frequently-updated mobile apps in the google play store," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1346–1370, 2016.

[30] S. Klepper, S. Krusche, S. Peters, B. Bruegge, and L. Alperowitz, "Introducing continuous delivery of mobile apps in a corporate environment: A case study," in *Proceedings of the Second International Workshop on Rapid Continuous Software Engineering*, ser. RCoSE '15, 2015, pp. 5–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820678.2820681

[31] J. Miranda, "How Etsy does continuous integration for mobile apps," https://www.infoq.com/news/2014/11/continuous-integration-mobile/, Nov. 2014.

[32] K. Nassim, "Etsy's journey to continuous integration for mobile apps," https://codeascraft.com/2014/02/28/etsys-journey-to-continuous-integration-for-mobile-apps/, Nov. 2014.

[33] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2013.

[34] J. Gao, W.-T. Tsai, R. Paul, X. Bai, and T. Uehara, "Mobile Testing-as-a-Service (MTaaS) — Infrastructures, issues, solutions and needs," in *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, 2014, pp. 158–167.

[35] https://aws.amazon.com/device-farm/.