2016 IEEE/ACM 38th IEEE International Conference on Software Engineering Companion

Continuous Deployment at Facebook and OANDA

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

Mitchell Douglas
Dept. of Computer Science
Stanford University
Stanford, CA, U.S.A. 94305
mrdoug95@stanford.edu

Michael Gentili
OANDA Corp.
140 Broadway
New York, NY, U.S.A. 10005
gentili@oanda.com

Laurie Williams
Dept. Computer Science
NC State University
Raleigh, NC, U.S.A. 27695
williams@csc.ncsu.edu

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

Michael Stumm
ECE Department
University of Toronto
Toronto, Canada M8X 2A6
stumm@eecg.toronto.edu

ABSTRACT

Continuous deployment is the software engineering practice of deploying many small incremental software updates into production, leading to a continuous stream of 10s, 100s, or even 1,000s of deployments per day. High-profile Internet firms such as Amazon, Etsy, Facebook, Flickr, Google, and Netflix have embraced continuous deployment. However, the practice has not been covered in textbooks and no scientific publication has presented an analysis of continuous deployment.

In this paper, we describe the continuous deployment practices at two very different firms: Facebook and OANDA. We show that continuous deployment does not inhibit productivity or quality even in the face of substantial engineering team and code size growth. To the best of our knowledge, this is the first study to show *it is possible to scale the size of an engineering team by 20X and the size of the code base by 50X without negatively impacting developer productivity or software quality.* Our experience suggests that top-level management support of continuous deployment is necessary, and that given a choice, developers prefer faster deployment. We identify elements we feel make continuous deployment viable and present observations from operating in a continuous deployment environment.

1. INTRODUCTION

Continuous deployment is the process of deploying software into production as quickly and iteratively as permitted by agile software development. Key elements of continuous deployment are:

1. software updates are kept as small and isolated as reasonably feasible;
2. they are released for deployment immediately after development and testing completes;
3. the decision to deploy is largely left up to the developers (without the use of separate testing teams); and
4. deployment is fully automated.

ICSE '16 Companion, May 14-22, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2889160.2889223

This practice leads to a continuous stream of software deployments, with organizations deploying 10s, 100s, or even 1,000s of software updates a day.

Continuous deployment has been embraced by a number of high-profile Internet firms. Facebook was utilizing continuous deployment as early as 2005. Flickr was one of the first organizations to publicly embrace continuous deployment; it reported an average of 10 software deployments a day in 2009 [1]. At Etsy, another early adopter which reported over 11,000 software deployments in 2011 [2], newly hired software developers are assigned a simple bug to find and fix on their first day of work, and are expected to deploy their fix to production servers within a day or two — without supervision and without a separate testing team. Netflix utilizes continuous deployment at scale in the cloud [3].

Potential benefits of continuous deployment that are often mentioned include improved productivity and motivation of developers, decreased risk, and increased software quality. Often stated potential drawbacks include lack of control of the software cycle, increased instability, and unsuitability for safety- or mission-critical software. It is open to debate whether these stated pros and cons are valid, complete, and lead to a compelling answer about the well-foundedness of continuous deployment.

In this paper, we present both quantitative and qualitative analyses of the continuous deployment practices at two very different firms over a period of 7 years and 5 years, respectively. Facebook has thousands of engineers and a set of products that are used by well over a billion users; its backend servers can process billions of queries per second. OANDA, the second firm, has only roughly 100 engineers; it operates a currency trading system that processes many billion dollars worth of trades per day and is thus considered mission critical. The continuous deployment processes at both firms are strikingly similar even though they were developed independently.

We make two key contributions in this paper:

1. We present quantitative evidence that (i) continuous deployment does not inhibit productivity or quality even when the size of the engineering team increases by a factor of 20 and the code size grows by a factor of 50; (ii) management support of continuous deployment is vital; and (iii) developers prefer faster deployment of the code they develop.

2. We identify elements that, based on our experience, make continuous deployment viable and present observations from operating in a continuous deployment environment. In doing so, our aim is to help software development organizations better understand key issues they will be confronted with when implementing continuous deployment.

Section 2 provides background on continuous deployment. Section 3 describes our methodology. Section 4 presents findings derived from our quantitative analysis, and Section 5 presents insights and lessons learned from our qualitative analysis. We close with limitations, related work and concluding remarks.

2. CONTINUOUS DEPLOYMENT

Continuous deployment is a software engineering practice in which incremental software updates are tested, vetted, and deployed to production environments. Deployments can occur within hours of the original changes to the software.

A number of key developments have enabled and motivated continuous deployment, the primary one being *agile software development* [4, 5, 6] that began in the late 1990s and that is now used in some form in many if not most organizations. Agile development methodologies embrace higher rates of change in software requirements. Software is developed iteratively with cycles as short as a day [7]. Agile development has been shown to increase productivity, and it is arguably one of the reasons software productivity has started to increase some 10-15 years ago after decades of stagnation [8, 9]. Continuous deployment is a natural extension of agile development. Other developments include lean software development [10], kanban [11], and kaizan [10]. *DevOps* is a movement that emerged from combining roles and tools from both the development and operations sides of the business [12].

For Web-based applications and cloud-based SAAS offerings, software updates can occur continuously intraday because the updates are largely transparent to the end-user. The process of updating software on PCs, smartphones, tablets, and now cars, has largely been automated and can occur as frequently as daily, since the updates are downloaded over the Internet. In these cases, software is deployed continuously to a beta or demo site, and a cut is taken periodically (e.g., every two weeks for iOS) to deploy to production. HP applies continuous deployment to its printer firmware, so that each printer is always shipped with the latest version of software [13].

2.1 Continuous deployment process

Two key principles are followed in a continuous deployment process. First, software is updated in relatively small increments that are independently deployable; when the update has been completed, it is deployed as rapidly as possible. The size of the increments will depend in part on the nature of the update and will also differ from organization to organization.

Second, software updates are the responsibility of the software developers who created them. As a consequence, developers must be involved in the full deployment cycle: they are responsible for testing and staging, as well as providing configuration for, and supporting their updates post-deployment, including being on call so they can be notified of system failures [14]. This broad responsibility provides the fastest turn-around time in the event of failures. A quick feedback loop is an important aspect of the process, because developers still have the code design and implementation details fresh in their minds, allowing them to rectify issues quickly. Moreover, this ensures a single point of contact in the event of a problem.

Continuous deployment is a team-oriented approach sharing common goals but having decentralized decision-making. Developers have significant responsibility and accountability for the entire software lifecycle and in particular for decisions related to releasing software. Proper tools reduce risk, as well as make common processes repeatable and less error prone.

A continuous deployment process includes the following key practices:

Testing. Software changes[1] are *unit-* and *subsystem-tested* by the developers incrementally and iteratively, as they are being implemented. Automated testing tools are used to enable early and frequent testing. A separate testing group is not generally used. The developers are tasked with creating effective tests for their software.

Developers begin *integration testing* to test the entire system with the updated software. For this, developers may use virtual machine environments that can be instantiated at a push of a button, with the target system as similar to the production environment as possible. At any given time, each developer may have one or more virtual test environments instantiated for unit and system testing. Automated system tests simulate production workloads and exercise regression corner cases.

After successful completion of system testing, *performance tests* are run to catch performance issues as early as possible. The performance tests are executed by the developers in non-virtual environments for reproducibility. Automated measurements are made and compared with historical data.

The developers are responsible for and perform these tests. When problems are identified, the process loops back so they can be addressed immediately.

Code review. Code reviews are prevalent in continuous deployment processes. Because developers are fully responsible for the entire lifecycle of the software, code reviews are taken more seriously and there is far less resistance to them.

Release engineering. Once the developer determines the software is functionally correct and will perform as expected, she identifies the update as ready to deploy. This identification might occur by committing the update to a specific code repository or an update might be made available for deployment through a deployment management tool. The update may be handed off to a separate *release engineering team.*[2]

Release engineering is a team separate from the development group. The mission of release engineering is to compile, configure and release source code into production-

[1] We use the term software "change" and "update" interchangeably; in practice, a software update that is deployed may include multiple changes.

[2] A number of organizations do not use such a release engineering team; e.g., Etsy and Netflix. Instead, the developers are able to initiate deployment themselves by using a tool that deploys the software automatically. However, based on our experiences at Facebook and OANDA, we have come to believe that having a separate team is valuable when reliable software is important.

ready products. Release engineering ensures traceability of changes, reproducibility of build, configuration and release while maintaining a revertible repository of all changes. Tasks include full build from source to ensure compatibility with the production environment, re-verification of developer evaluation of the tests done by developers and a full install/uninstall test. Software with issues is rejected for release and passed back to developers. Otherwise, the software is deployed into production. The deployment is highly automated to prevent errors, to make it repeatable and to have each step be appropriately logged.

The release engineering group begins its involvement early, when development begins. The group communicates with teams to learn of new updates in progress to identify high-risk updates so they can provide advice on best practices to increase the probability of a smooth deployment. The group also identifies potential interactions between software updates that could cause issues when deployed together and handles them accordingly. For example, two software releases that require opposing configurations of an individual parameter may not have been noticed by the two separate teams but should be noticed by release engineering. The release engineering group assesses the risk of each upgrade, based on the complexity of the upgrade, the caliber of the team that created the update, and the history of the developers involved. With upgrades deemed to have higher risk release engineering group may do extensive tests of its own.

When the release engineering group is ready to deploy an update, it coordinates with the appropriate developers to ensure they are available when deployment occurs.

Deployment. Software is deployed in stages. Initially software updates are deployed onto a beta or a demo system. Only after the new software has been running without issue on the beta or demo site for a period of time, are they pushed to the final production site. Beta/demo sites have real users and are considered production sites. Where possible, organizations use a practice commonly referred to as "dog fooding" whereby a portion of the development organization uses the most updated software before the changes are pushed to external users. Generally, the release of deployed software occurs in stages to contain any issues before general availability to the entire customer code base. Staged deployment strategies might include:

- **blue-green deployments**: A deployment strategy where a defective change to a production environment (blue) can be quickly switched to the latest stable production build (green) [15]. The change may initially be made available to, for example, 1% of the client base in a specific geographical location, thus limiting exposure (and with it, reputational risk), and only when confidence increases that the software is running properly is the fraction gradually increased, until it ultimately reaches 100%. When problems are detected, the fraction is quickly reduced to 0%.
- **dark launches**: A deployment strategy where changes are released during off peak hours; or where code is installed on all servers, but configured so that users do not see their effects because their user interface components are switched off. Such launches can be used to test scalability and performance [14] and can be used to break a larger release into smaller ones.
- **staging/baking**: A stage in the deployment pipeline where a new version of software is tested in conditions similar to a production environment. An example of this is called shadow testing where production traffic is cloned and sent to a set of shadow machines that execute newer code than production. Results between production and shadow environments can be automatically compared and discrepancies reported as failures.

Configuration tools are used to dynamically control (at run time) which clients obtain the new functionality. If issues occur after the update is deployed, the release engineering group triages the issue with the developer that created the update. Possible remedies include: reverting the deployed update to the previous version of the software through a deployment rollback, rapid deployment of a hotfix, a configuration change to disable the feature that triggers the issue using for example *feature flags* or *blue-green deployments* [15], or (for lower priority issues) filing a bug report for future remediation.

2.2 Transition to Continuous Deployment

Introducing continuous deployment into an organization is non trivial and involves significant cultural shifts [16]. A number of requirements must be met before continuous deployment can be successful. Firstly, buy-in from the organization, and in particular from senior management is critical. Without full support, the process can easily be subverted. This support is particularly important when a major failure occurs, at which point organizations often tend to gravitate back to more traditional processes.

Secondly, highly cohesive, loosely coupled software makes small changes more likely to be better isolated. Small deployment units allow updating of software with higher precision and give the release engineering team flexibility in not releasing problematic updates.

Thirdly, tools to support the process are important, but they require appropriate investment. Tools not only increase the productivity of developers, but also decreases risk because they reduce the number of manual operations (where errors are most likely to occur) and make deployment operations repeatable. Beyond standard tools, such as revision control and configuration management systems (as described, e.g., in [15]) we highlight a few tools that are particularly important for continuous deployment:

- **automated testing infrastructure**: testing functionality, performance, capacity, availability, and security must be fully automated with the ability to initiate these tests at the push of a button. This automation enables frequent testing and reduces overhead. Testing environments must be as identical to the production environment as possible with full, realistic workload generation. As mentioned earlier, virtual machine technology can play an important role.
- **deployment management system (DMS)**: helps the release engineering group manage the flow of updates. The DMS has various dashboards that provide an overview of the updates progressing through the development and deployment phases. For each update, the DMS links together all the disparate systems (the change set from source control, the bug tracking system, code review comments, testing results, etc.) and the developers responsible for the update. Ultimately, the DMS is used to schedule the deployment.

- **deployment tool**: executes all the steps necessary for single-button deployment of an update to a specified environment, from initial compilation of source code, to configuration, to the actual installation of a working system and all steps in between. The tools can also roll-back any previous deployment, which may be necessary when a serious error is detected after deployment. Automating the roll-back minimizes the time from when a critical error is identified to when it is removed, and ensures that the rollback is executed smoothly.
- **monitoring infrastructure**: a sophisticated monitoring system is particularly important to quickly identify newly-deployed software that is misbehaving.

Some of these tools are not readily available off-the-shelf because they need to be customized to the development organization. Hence, they tend to be developed in house. For example, the DMS is highly customized because of the number of systems it is required to interface with, and it also automates a good portion of the software deployment workflow, thus making it quite specific to the organization. The deployment tool is also highly customized for each particular environment. For each deployment module, automatic installation and rollback scripts are required.

3. CASE STUDIES

In this section, we present information about our case study companies, Facebook and OANDA, as well as the methodology we used.

3.1 Facebook

Facebook is a company that provides social networking products and services, servicing well over a billion users. The case study presented here covers the time period 2008-2014, during which time, the software development staff at Facebook grew 20-fold from low 100's to 1000's. The vast majority of the staff is located at Facebook headquarters in Menlo Park, CA, but staff from roughly a dozen smaller remote offices located around the world also contributes to the software.

The case study covers all software deployed within Facebook during the above stated time period, with the exception of newly acquired companies that have not yet been integrated into the Facebook processes and infrastructure (e.g., Instagram that is in the process of being integrated). The software is roughly partitioned into 4 segments:

1. Web frontend code: primarily implemented in PHP, but also a number of other languages, such as Python,
2. Android frontend code: primarily implemented in Java
3. iOS frontend code: primarily written in Objective- C
4. Backend infrastructure code that services the front-end software: implemented in C, C++, Java, Python, and a host of other languages.

In Facebook's case, the beta site is the Facebook site and mobile applications with live data used by internal employees and some outside users.

3.2 OANDA

OANDA is a small, privately held company that provides currency information and currency trading as a service. Its currency trading service processes a cash flow in the many billions of dollars a day for online customers around the world. OANDA has about 80 developers, all located in Toronto, Canada; this number stayed reasonably constant during the period of the study.

The trading system frontend software is implemented in Java (Web and Android), and Objective-C (iOS). Backend infrastructure software servicing the front end is primarily implemented in C++, but also Perl, Python and other languages.

OANDA also "white-labeled" its trading system software to several large banks, which means the software ran on the banks' infrastructure and was customized with the banks' look and feel. The banks did not allow continuous updates of the software running on their servers. As a result, the authors had a unique opportunity to compare and contrast some differences between continuous and noncontinuous deployment of the same software base.

In OANDA's case, the demo system is a full trading system, but one that only trades with virtual money (instead of real money) — it has real clients and, in fact a much larger client base than the real-money system.

Table 1: Facebook commits considered

	commits	lines inserted or modified
WWW	705,631	76,667,915
Android	68,272	10,635,605
IOS	146,658	12,671,047
Backend	238,742	30,828,829

3.3 Methodology

For our quantitative analysis, we extracted and used data from a number of sources at both Facebook and OANDA. At Facebook, Git repositories provided us with information on which software was submitted for deployment when, since developers committed to specific repositories to transfer code to the release engineering team for deployment. For each commit, we extracted the timestamp, the deploying developer, the commit message, and for each file: the number of lines added, removed or modified.

Commits were recorded from June, 2005 (with Android and iPhone code starting in 2009). For this study we only considered data from 2008 onwards up until June 1014. Table 1 lists the four repositories used along with the number of commits recorded. The table also provides an indication of the magnitude of these commits in terms of lines inserted or modified. In total these repositories recorded over 1 million commits involving over 100 million lines of modified code.[3]

All identified failures at Facebook are recorded in a "SEV" database and we extracted all errors from this database. Failures are registered by employees when they are detected using an internal SEV tool. Each failure is assigned a severity level between 1 and 3: (1) critical, where the error needs to be addressed immediately at high priority, (2) medium-priority, and (3) low-priority.[4] In total, the SEV database contained over 4,750 reported failures.

[3]In our analysis, we did not include commits that added or modified more than 2,000 lines of code so as not to include third party (i.e., open-source) software packages being added or directories being moved. Not considering these large commits may cause us to underreport the productivity of Facebook developers.

[4]The developer that developed the software does not typically set the severity level of the error.

For our analysis, a developer was considered "active" at any given point in time if she had committed code to deployment in the previous three weeks. We only considered the developer that issued the commit, even though other developers may have also contributed to the code being committed. This may skew our analysis, but since we found that a Facebook developer deploys software once a day on average, we believe the skew is minimal.

OANDA used a (inhouse-developed) deployment management system (DMS)to keep track of every stage of software as it progressed through the deployment process. In aggregate, over 20,000 deployments were recorded between April, 2010 and June, 2014. Git repositories, which deployment records refer to, provided information with respect to the number of lines of code added/changed/deleted with each deployment.

Unfortunately, OANDA did not maintain a failure database as Facebook did (and only started to use Redmine and JIRA relatively recently). Instead, detected failures were typically communicated to the developers responsible for the code through email/messenger in an ad hoc way. As a result, OANDA failure data is largely unavailable. However, we extrapolated critical (i.e., severity level 1) failures using the following heuristic: if software is deployed a second time within 48 hours of a first deployment, then we assume that the second deployment was necessary to fix a critical error that became visible after the first deployment.

4. QUANTITATIVE ANALYSIS

Wherever possible, we present both OANDA and Facebook data together. However, some observations could be made at only one company. Facebook understandably had a larger sample size and more accurate production failure data, but OANDA had better data related to management and human aspects.

.

4.1 Productivity

We measure productivity as number of commented lines of code shipped to production (while realizing that LoC per person week can be controversial). The metric was chosen largely because it is easy to count, readily understood and the data was available.

Each Facebook developer releases an average of 3.5 software updates into production per week. Each update involves an average of 92 lines of code (median of 33) that were added or modified. Each OANDA developer releases on average 1 update per week with each update involving 273 lines of code on average (median of 57) that were added or modified. OANDA has a far higher proportion of backend releases than Facebook, which may explain some of the productivity differences.

Figure 1 depicts the average number of new and modified lines of code deployed into production per developer per week at Facebook for the period January 1, 2008 to July 31, 2014.[5] The figure shows that productivity has remained relatively constant for more than six years. This is despite the fact that the number of developers increased by a factor of over 20 during that period.

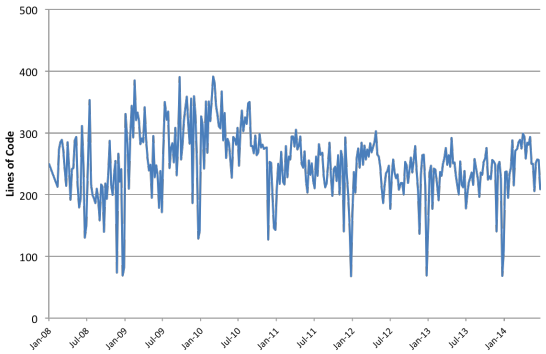

Figure 1: Lines of modified or added code deployed per developer per week at Facebook.

Observation 1: *Productivity scaled with the size of the engineering organization.*

Our experience has been that when developers are incentivized with having the primary measure of progress be working software in production, they self-organize into smaller teams of like-minded individuals. Intuitively, developers understand that smaller teams have significantly reduced communication overheads, as identified by Brooks [17, 18]. Hence, one would expect productivity to remain high in such organizations and to scale with an increased number of developers.

Moreover, productivity remained constant despite the fact that over the same period:

1. the size of the overall code base has increased by a factor of 50; and

2. the products have matured considerably, and hence, the code and its structure have become more complex, and management places more emphasis on quality.

Observation 2: *Productivity scaled as the product matured, became larger and more complex.*

Note that we do not claim that continuous deployment is necessarily a key contributor to this scalability of productivity since productivity scalability is influenced by many factors, including belief in company mission, compensation, individual career growth, work fulfillment etc.— an organization needs to get most (if not all) of these factors right for good scalability.

However, we can conclude from the data that continuous deployment does not prevent an engineering organization from scaling productivity as the organization grows and the product becomes larger and more complex. Within Facebook, we consider this observation a startling discovery that to the best of our knowledge, has not been shown for other software engineering methodologies.

In the authors' opinion, a focus on quality is one factor that enables the software development process to scale. A strong focus on quality company-wide with buy in from management implies clear accountability and emphasis on automating routine tasks to make them repeatable and error free. High degrees of automation also make it easier to run

[5]OANDA's software development productivity is shown later in Figure 3, where it is depicted in terms of number of deployments per week. Since the size of OANDA's engineering team did not change materially over the period studied, no conclusions can be drawn from the OANDA data as it relates to scalability.

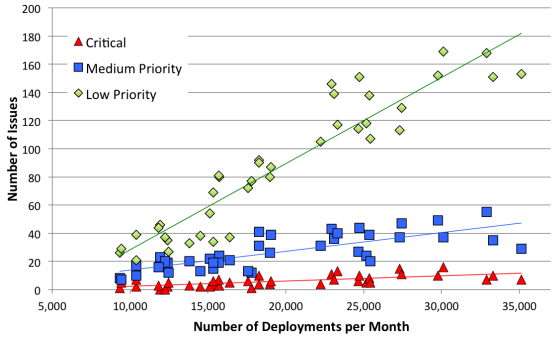

Figure 2: Number of production issues per month as a function of the number of deployments at Facebook.

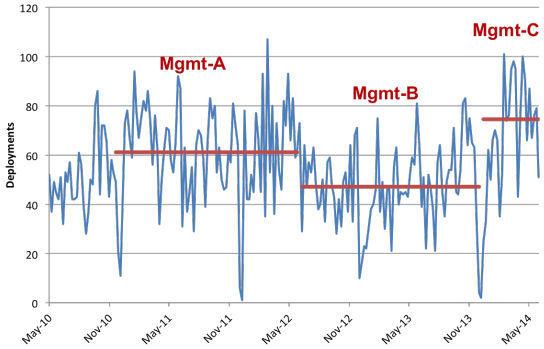

Figure 3: Number of deployments at OANDA per week over three management regimes. The red lines depict the average over the time periods shown.

tests and collect metrics, making it more likely for developers to take these initiatives.

4.2 Quality

The quality of deployed software is of interest because continuous deployment does not have the same checks and balances that more traditional software engineering processes have with their independent test teams. (Neither Facebook nor OANDA had a test team, although both had release engineering teams.) Rather, continuous deployment relies on the accountability of high-caliber developers who are responsible for software throughout its entire lifecycle, including idea generation, architecture, design, implementation, testing and support in production; developers independently decide when to deploy their code and hence are responsible if production issues arise by being part of an on-call rotation.

Production failures are used as an indicator of quality. At Facebook, each failure is categorized by severity: (i) critical, where the issue needs to be addressed immediately at high priority, (ii) medium-priority, and (iii) low-priority. The graph in Figure 2 depicts the number of failures for each severity level as a function of the number of deployments per month. The triangular marks in red represent critical failures. The number of critical and medium priority failures grow significantly slower than the number of deployments per month, with trendline slopes of 0.0004, 0.0013, and 0.0061, respectively; and this despite the growth in size and complexity of the product.

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

This observation demonstrates that it is possible to manage quality with continuous deployment without having separate independent test teams. The number of low-priority issues increases linearly with the number of lines of code deployed (which is not much of a surprise given Facebook engineering's focus on productivity). Perhaps more interesting is the fact that the release engineering team did not increase in size as the number of deployments increased (primarily because of the focus on automating the process as much as possible).

Each organization will have to make its own tradeoffs with respect to quality and productivity. Often these tradeoffs are difficult if not impossible to make a priori. They are influenced many factors such as customer tolerance, companies values, the type of product etc. and can vary over time. Finding the appropriate quality vs. productivity tradeoff can require iteration. For example, Figure 2 is a representation of the quality vs productivity tradeoff Facebook made. During the approximate period January 2009 – July 2010, there was a surge in productivity (Figure 1). Figure 4 shows there was a degradation in quality during this same period as measured by the number of hotfixes (used as a proxy for quality). Efforts to improve quality then came at the expense of some productivity.

4.3 Human Factors

Management buy-in

Management buy-in and competence in continuous deployment can significantly affect the productivity of the engineering organization and possibility the viability of the continuous development process. As an example, OANDA had a number of executive changes in the past four years with most other factors relatively constant. It started with management team (Mgmt-A) having an engineering background and supporting continuous deployment. In mid 2012, management was replaced with executives having a business background (Mgmt-B). Their inclinations were more towards more traditional software engineering processes. At the end of 2013, executive management was replaced again (Mgmt-C). This team had a Silicon Valley background and was well versed with and supportive of continuous deployment.

Figure 3 shows the effects of management on productivity levels. For example, the average number of deployments dropped by 23% after Mgmt-B took control, despite other factors such as the engineering team remaining largely unchanged. Perhaps most surprising is how quickly productivity changes occurred after new management was introduced.

Observation 4: *Management support can affect the productivity of an engineering organization.*

Developer motivation

Some argue that developers are more motivated if their software is deployed into production quickly. While we do not know how to measure motivation directly, at Facebook, developers can choose whether their software is deployed as part of the next weekly release (which is the default) or, more quickly, as part of the next daily release. Figure 4 indicates that developers chose to deploy their software as

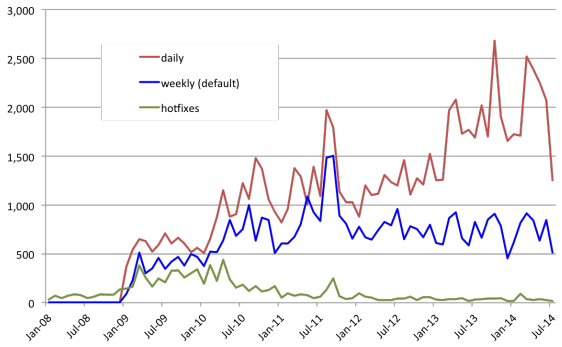

Figure 4: The number of deployment requests per week by type of request.

quickly as possible when given the choice. Figure 4 depicts the release rate over time for daily and weekly releases. Initially, they are fairly similar. However, as time progresses, the number of deployments in the weekly release decreases, even in the presence of rapid engineering growth rates. With the choice of a daily or weekly deployment entirely up to the individual engineer, we conclude the preference is towards shorter release cycles.

Observation 5: *Developers prefer faster deployments over slower ones.*

5. LESSONS LEARNED

Combined, the authors have 13 years of experience with continuous deployment at Facebook and OANDA. In this section, we discuss some of the insights and lessons learned at the two organizations. These are obviously subjective — other organizations that have embraced continuous deployment may have had different experiences and may draw different conclusions. The objective of this section is primarily to present some of the issues an organization supporting continuous deployment will likely be confronted with.

Desirability of continuous deployment

OANDA used continuous deployment for its own systems, but also rebranded and "white-labeled" its systems to several large, international banks. The banks' policies did not allow continuous deployment. This allowed a side-by-side comparison of experiences between continuous deployment and non-continuous deployments of the same software base.

A first observation was that white-labeled trading systems were at a competitive disadvantage. What their clients considered irritating bugs , but deemed by the bank to be non-critical, were not fixed for many months. New functionality that became standard in the fast moving currency trading market were made available late.

A second observation was that each deployment in the white-labeled product incorporated thousands of changes, which increased anxiety surrounding the deployment and led to much more extensive testing than deployments of individual changes. Moreover, failures identified at deployment time were significantly harder to isolate to identify their root cause. Developers had mentally moved on to work on software several generations later, and as a result worked on fixing the fault with a negative attitude, knowing they were working on software that was already obsolete. Fixing these faults was clearly less efficient and caused more overhead.

Considerable and continuous investment is required

Continuous deployment requires continuous investments in educating and coaching developers as well as in building, improving, and maintaining tool infrastructure.

Education is needed because the software process with continuous deployment is sufficiently different from what is traditionally taught and practiced. Both companies offer 6-12 week training periods for new developers. Developers joining OANDA are typically first assigned to the release engineering team for several months for broad training.

Both organizations have separate teams developing and improving tools that enable continuous deployment (Section 2.2). Although some tools are readily available, many had to be developed or adapted to suit organizational needs. Existing tools are often too limited or force a specific process that does not fit the organization. For example, both Facebook and OANDA found it necessary to develop their own deployment management systems.

Developing, improving, and maintaining these tools takes considerable effort. Facebook had approximately 5% of its engineers dedicated to the task, while OANDA had 15% at time of writing. In our experience, it is more difficult for smaller companies to use continuous deployment because a larger percentage of its engineering resources are needed for internal tool development.

Versatile and skilled developers required

To thrive in an continuous deployment environment, developers need to be: (*i*) generalists with the ability to understand many aspects of the system; (*ii*) good at firefighting and systems debugging; (*iii*) bottom-up capable, willing and able to be responsible (and accountable) for making sound deployment decisions, taking quality, security, performance, and scalability into account; and (*iv*) able to work in an organization that some see as disorganized and perhaps chaotic; taking appropriate initiatives when necessary. This makes finding suitable candidates to hire more challenging.

Being a generalist is particularly important when working on complex software, such as backend systems. Generalists require a broader set of skills and a mindset to be able to reason with systems that they haven't developed and don't have intimate knowledge of. In our experience, there are talented developers who are not well suited, or interested, in this type of role and may be better suited in more structured organizations.

Technical management essential

As with other agile methodologies, organizations embracing continuous deployment tend to have a strong bottom-up culture with developers making many key decisions. Organizations are flatter, as managers can have an increased number of direct reports. However, a different type of manager is needed because they play a different role: they influence rather than operate within a chain of command. We find that in this environment, it is critical that managers be respected by the developers. Being technologically excellent makes this easier. Filling management roles with suitable candidates has been a key challenge in both organizations. This is exacerbated because the continuous deployment process is not yet practiced in many other organizations. As a result, management roles are often filled by promoting from within.

Empowered culture
A suitable culture within the organization is critically important to make continuous deployment effective. Developers need to be appropriately supported and given freedom along with their responsibilities. At both Facebook and OANDA, developers are given almost full access to the company's code base and are empowered to release new code within days of being hired. At Facebook, developers are given much freedom over which team to join and are encouraged to move to different teams periodically. Open and transparent communication across the company is particularly important so that developers have context in which to make informed decisions in a decentralized way.

Developers must be encouraged to take calculated risks, and the presence of effective testing and deployment infrastructure makes it possible to identify and correct errors quickly while supporting developers who move fast. They should not be penalized for failures that occur as a result of the risks taken (as long as the taken decisions were reasonable and prudent given the circumstances).

Managing the risk-reward tradeoff
Because deployment decisions rest with each individual developer, he or she evaluates risks and rewards independently. Examples of risks include insufficient testing, unrepeatable release procedures and the overhead and inconvenience of dealing with defects in deployed software. Rewards include benefits to the company, incentives, and self-fulfillment.

An important role of management is to improve the risk-reward tradeoff so that better decisions can be made by the developers. In particular, management should play a role in systematically reducing risks for the continuous deployment process. Examples include allocating resources for building tools to improve repeatability of deployments and testing as well as having a no-blame culture. This risk reduction allows engineers to execute at a faster pace because it frees them up from mitigating their own risks. At the same time, proper incentives encourage developers to move even faster. For example, at Facebook, the impact of working code in production is a principle parameter of incentives.

Need for objective retrospective when failures occur
The culture of a software development organization is often defined after a serious failure occurs. One of the most challenging aspects is to retain management support after such an event. It is easy to lose confidence when the approach differs significantly from the "textbook" approach. A common reaction is to blame the process as the root cause and, in turn, increase the complexity of it. Over time, this makes the process more cumbersome, resulting in a more conservative, less productive organization. This phenomenon is commonly referred to as runaway process inflation [19].

The authors experienced this first hand at OANDA: after using continuous deployment successfully for several years, a major failure occurred that cost the company a significant amount of money within seconds. Senior management's initial reaction was to add engineering hierarchy, a more complicated approvals process and a full-blown, separate and traditional testing team. A postmortem analysis of the event identified what had gone wrong: the output of failing unit tests (that correctly identified the bug) were disregarded because of too many false positive error messages. Fortunately, engineering was able to convince management to instead promote a renewed focus on automated testing frameworks, and require that the output of those testing systems be part of the deployment decision.

Having a continuous deployment champion in the senior management team is, in our view, critical, especially for non-technical management teams.

Dependencies considered harmful
In our experience, engineering teams that practice continuous deployment work best if divided into small, cohesive and largely autonomous groups that are not architecturally dependent on (many) other teams. According to Brooks, communication overhead increases at a rate of n^2, with n the number of people in a group [17]. Software that is modularized into smaller, loosely coupled units reduces the need for inter-team communication. Inevitably, the product architecture and the team communication structure are aligned, per Conway's law [20]. Cohesive teams typically have the characteristic of being self-organized with like-minded individuals that work well together.

Extra effort for comprehensive software updates
Changes, bug fixes and enhancements localized to one system or one organization are generally handled well by continuous deployment. However, as per Conway's Law [20] "organizations that design systems are constrained to produce designs which are copies of the communication structures in these organizations." In our experience, this occurs in one of two circumstances. First, if the individual group has a high friction release process. Second, if the change spans multiple teams with poor communications between them. We describe both in further detail.

Instituting continuous deployment requires effort, infrastructure and iteration over a period of time. For example, OANDA spent 2 years perfecting continuous deployment for mission critical infrastructure trading billions of dollars a day. OANDA's backend infrastructure release process was more sophisticated than Facebook's resulting from a concerted engineering effort. However, OANDA's database schema changes always were always ad hoc and full of fear. OANDA never spent the time to streamline its database schema changes which resulted in changes that added tables and columns resulting in a superset of what was needed over time. Database schema changes have been shown to be possible with a concerted effort.

Changes spanning multiple teams are more difficult to coordinate without agreed upon procedures. For example, Facebook has had difficulty with configuration changes in part because they spanned multiple groups. The company didn't have the appropriate communication channels in place for configuration changes to be properly vetted. As a result, a group might make changes that seem reasonable when viewed locally but had undesirable consequences globally.

Absence of separate testing team
Whether to have a separate testing team or not is one of the more controversial questions and still an active area of debate. Some engineers and managers argue that a test team would benefit a continuous deployment organization while others argue strongly against it.

One tenant of continuous deployment is clear accountability between code in production and the engineer who created it. A testing group dilutes this accountability, especially during times of system failures – opinions often disagree in whether the failure was due to a system test escape or poor

development practices because not everything is testable. In our experience, diluting responsibility of ownership increases the time to resolution of system failures.

Proponents of testing groups claim that individuals who are not versed in the code and have a broader product perspective are better testers, and that career testers will always be more effective than developers having less interest but more subject matter knowledge. Moreover, context switching between development and testing also yields inefficiencies. In other words, software developers are not best at testing and specialization of tasks is the best approach.

In the experiences of the authors, developers have a higher degree of influence and better software development skills. By being less interested in testing, developers are motivated to find creative means to automate it. Both Facebook and OANDA have pre-software release systems (internal beta used by Facebook employees and a virtual money system used by OANDA's customers) that serve effectively as test systems. Both were conceived by developers to improve test coverage by leveraging user resources. In the case of OANDA, the systems also provides a marketing benefit.

Convergence on local minima
The iterative process may lead to local minima in products when continuous, but incremental, improvements are made. Teams may have resistance to make larger, bolder moves (perhaps because of the fear of risks involved). OANDA's trading GUI is an example. The product is reasonable, but the number of users using it makes it difficult to make dramatic changes. Getting out of the local minima may require an initiative that comes from outside the culture, perhaps through a separate skunks work project (or by acquiring another firm).

Sub-standard product releases
Developers may deploy products or product features too early with the understanding that they can be improved later through quick iterations. The continuous deployment process can make the use of this argument too easy, yet releasing too early can alienate end-users (who often will not give the released product a second chance). Hence, the flexibility the continuous deployment process offers should be used with care. A stakeholder representing end users (e.g., a product manager) can play an important roll in educating and convincing developers.

Susceptible to resource and performance creep
With continuous deployment, each deployed update is relatively small, so increases in system resource usage (e.g. CPU, memory etc.) due to the update are expected to be small. However, aggregated over a large number of deployments, the additional resource usage may become substantial. That is, continuous deployment can suffer from a "death by a thousand cuts," because release decisions are decentralized to individual developers rather than having a central coordinating body as in the waterfall model.

Consider a hypothetical example where a set of updates each consume an additional 0.05% of CPU. The 0.05% additional CPU may seem reasonable and justifiable given the added functionality. However, having 100 such releases per week (for example) means that CPU utilization would increase by 260% within a year.

Decentralized release decisions may lead to aggregate decisions that do not make sense. We have been able to solve this by making developers aware of the issue and continuously monitor for resource creep. Tools that allow precise measurement (e.g., static analysis, lint rules, run-time measurements) of incremental changes give developers feedback on the implications of their releases. We have also found it useful to dedicate a group to look at variance in traffic patterns and propose performance optimizations and efficient architectural generalizations of common usage patterns.

Susceptible to not-invented-here syndrome
We observed that developers working in a continuous deployment environment do not like to be held back. They find it difficult to work with other parts of the organization, or outside organizations, that work at slower speeds or with a different culture. As a result, teams may reinvent components that would otherwise be available from outside the team. In particular, third-party software vendors have their own release schedules and product philosophies typically aimed at the masses, which developers feel impede their freedom. Thus, developers often implement their own version, which allows them to march to their own schedule and allows them to create customized highly optimized solutions most appropriate for their needs, however duplication can result. We have even encountered similar components being developed in different groups concurrently.

Variability in quality
The structural and functional quality of software produced can vary substantially from developer to developer. Similarly, the ability to effectively deal with operational issues in the production system will also vary from developer to developer. With smaller teams, the quality of the contributions each developer makes is much more noticeable. The release engineering team and management can play a valuable role in mentoring and advising to equalize quality.

6. LIMITATIONS OF STUDY

This study has several obvious limitations. Firstly, the data, findings and experiences are from two companies and thus represent only two data points. While we believe that the two data points are interesting and important, they may not be representative, and other firms may have different experiences and or use different approaches. Secondly, while our quantitative analysis allows us to identify a number of findings, we can only hypothesize the root cause of our observations. Thirdly, our insights and the lessons we learned are fundamentally subjective; other organizations may well have had different experiences and come to completely different conclusions.

7. RELATED WORK

Few studies have evaluated continuous deployment. Humble et al. provide an early overview of continuous deployment [21]. Ollson et al. [22] and Dijkstra [23] present several case studies identifying barriers in transitioning from agile development to continuous deployment. Pulkkinen discusses strategies for continuous deployment in the cloud [24]. Neely et al. describe their experiences introducing continuous deployment at Rally [25].

Continuous deployment impacts the work arrangements of the software development team. Marschall reports that continuous deployments change developers' perceptions on quality when they can better see the link between their con-

tribution and the result "in the wild" [26]. Developers are responsible for making and testing changes that reach the customer rapidly and for responding to failures that result as a consequence of deploying those changes. Claps et al. found that stress may be increased, due to perceived pressure to deploy [16]. Hence, tools automating deployment are essential in keeping demands on developers realistic.

Hoff describes how Netflix's focus on high velocity delivery has caused them to virtually abandon standard software engineering processes [27]. Instead, small teams of developers are independently responsible for the development, support, and deployment of their software. Lehman's seminal work on software evolution predicts that progress on a product will slow when size and complexity increase [28]. In a prior study, Facebook has found that the code base of their product has grown superlinearly over time [14].

8. CONCLUDING REMARKS

We compared experiences with continuous deployment at two very different companies. The data reveals that continuous deployment allows scaling of the number of developers and code base size. For example, Facebook has been able to maintain productivity and keep critical-rated production issues relatively constant for over half a decade despite an order of magnitude increase in developers and code base size. Findings also reveal that developers, given the choice, prefer short release cycles over longer ones.

Continuous deployment is highly dependent on the inclinations of management, which may affect productivity, or even the viability of the approach. Customized tools, including deployment management, testing environments, and performance monitoring etc. are important and may consume a significant portion of engineering resources. Finally, continuous deployment has the limitation of resource creep — small changes' incremental resource demands may seem reasonable individually but not in the aggregate.

9. REFERENCES

[1] 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

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

[3] B. Schmaus, "Deploying the Netflix API," 2013. [Online]. Available: http://techblog.netflix.com/2013/08/deploying-netflix-api.html

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

[5] 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/

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

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

[8] F. Maurer and S. Martel, "On the productivity of agile software practices: An industrial case study," in *Proc. Intl. Workshop on Economics-Driven Software Engineering Research*, 2002.

[9] V. Nguyen, L. Huang, and B. W. Boehm, "An analysis of trends in productivity and cost drivers over years," in *Proc. of the 7th Intl. Conf. on Predictive Models in Software Engineering*, 2011, pp. 3:1–3:10.

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

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

[12] D. Edwards, "The history of DevOps," June 2009. [Online]. Available: http://itrevolution.com/the-history-of-devops/

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

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

[15] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Addison-Wesley, 2010.

[16] G. G. Claps, R. B. Svensson, and A. Aurum, "On the journey to continuous deployment: Technical and social challenges along the way," *Information and Software Technology*, vol. 57, pp. 21–31, 2015.

[17] F. P. Brooks, *Mythical Man Month: Essays on Software Engineering*. Addison-Wesley, 1975.

[18] ——, "No silver bullet: Essence and accidents of software engineering," *IEEE Computer*, vol. 20, no. 4, pp. 10–19, 1987.

[19] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2003.

[20] M. Conway, "How do committees invent?" *Datamation*, vol. 14, no. 5, pp. 28–31, April 1968.

[21] J. Humble, C. Read, and D. North, "The deployment production line," in *Proc. AGILE Conf.*, 2006, pp. 113–118.

[22] H. Olsson, H. Alahyari, and J. Bosch, "Climbing the "Stairway to Heaven" – a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software," in *Proc. 38th Euromicro Conf. on Software Engineering and Advanced Applications*, 2012, pp. 392–399.

[23] O. Dijkstra, "Extending the agile development discipline to deployment: The need for a holistic approach," Master's thesis, Utrecht University, 2013.

[24] V. Pulkkinen, "Continuous deployment of software," in *Proc. of the Seminar no. 58312107: Cloud-based Software Engineering*. University of Helsinki, 2013, pp. 46–52.

[25] S. Neely and S. Stolt, "Continuous delivery? Easy! Just change everything (well, maybe it is not that easy)," in *Proc. Agile Conf.*, 2013, pp. 121–128.

[26] M. Marschall, "Transforming a six month release cycle to continuous flow," in *Proc. Agile Conf.*, 2007, pp. 395–400.

[27] T. Huff, "Netflix: Developing, deploying, and supporting software according to the way of the cloud," 2011. [Online]. Available: http://highscalability.com/blog/2011/12/12/netflix-developing-deploying-and-supporting-software-accordi.html

[28] M. M. Lehman, D. E. Perry, and J. F. Ramil, "Implications of evolution metrics on software maintenance," in *Proc. Intl. Conf. on Software Maintenance*, 1998, pp. 208–217.