Development and Deployment at Facebook

Article *in* IEEE Internet Computing · July 2013

DOI: 10.1109/MIC.2013.25

3 authors, including:

Eitan Frachtenberg

101 PUBLICATIONS **2,864** CITATIONS

SEE PROFILE

Development and Deployment at Facebook

Dror G. Feitelson
Hebrew University

Eitan Frachtenberg
Facebook

Kent L. Beck
Facebook

Abstract

More than one billion users log in to Facebook at least once a month to connect and share content with each other. Among other activities, these users upload over 2.5 billion content items every day. In this article we describe the development and deployment of the software that supports all this activity, focusing on the site's primary codebase for the Web front-end. Information on Facebook's architecture and other software components is available elsewhere.

Keywords D.2.10.i Rapid prototyping; D.2.18 Software Engineering Process; D.2.19 Software Quality/SQA; D.2.2.c Distributed/Internet based software engineering tools and techniques; D.2.5.r Testing tools; D.2.7.e Evolving Internet applications.

Facebook's main development characteristics are speed and growth. The front-end is under continuous development by hundreds of software engineers. These engineers commit code to the version control system up to 500 times a day, recording changes in some 3,000 files. Naturally,

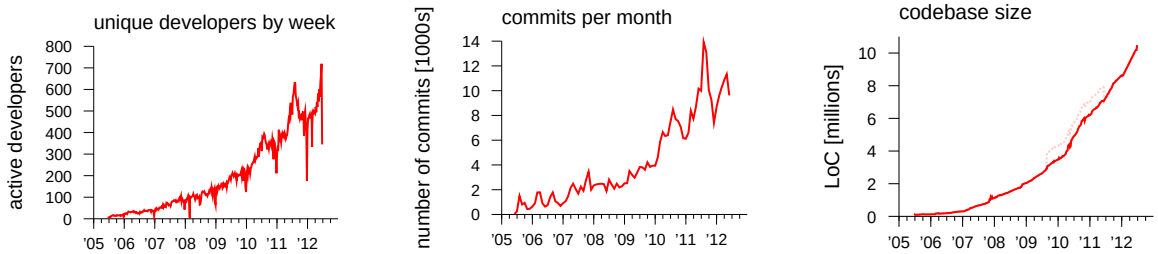

Figure 1: *Different aspects of Facebook growth: growth of the number of engineers working on the code, growth in the total activity of these engineers, and growth of the codebase itself. Dips in the number of engineers correspond to the winter holidays; peaks are caused by summer interns. In the codebase data we removed around 800,000 lines for internal use that existed from 2009 to 2011. The data was extracted from Web front-end git repository, which has more than 360,000 commits since June 2005.*

the rate of development activity has grown tremendously over the years, and so has the codebase itself (Fig. 1). The binary executable file run by Facebook servers to serve incoming requests is now about 1.5 GB in size.

Web companies like Facebook differ from conventional software companies in that the software they develop runs on their own servers, and is not installed at customer locations. This enables rapid updates to the software, and allows fine-grained control over versions and configurations. At Facebook, this deployment has led to a practice of daily and weekly "push" of new code to the servers. Before being pushed, code is subject to peer review, internal use, and extensive automated testing. After the code push, engineers carefully monitor the site's behavior identify any sign of trouble. But such technical facilities are not enough. Facebook also relies on a culture of personal responsibility, where every engineer is responsible for code they write and, when necessary, code they did not write that is affecting users or colleagues. This culture treats failures as an opportunity for improvement rather than as an occasion for assigning blame.

1 Perpetual Development

Facebook, like practically all other Internet-based companies, operates in perpetual development mode, in which engineers continuously develop new features and make them available to users. Consequently, the system also grows continuously, possibly at a super-linear rate. These two attributes, growth and rapid deployment, are the chief challenges that engineers need to overcome.

Software engineering textbooks typically assume a scenario where software is built for hire. In such a situation engineers first need to learn about the application domain and understand the goals for the new software.

At Facebook, the engineers are also users, so they have first-hand knowledge of what the system does and what services it provides. Moreover, internal use of Facebook tends to be more intensive than most use, so there is continuous tension between first-hand knowledge and knowledge derived from examining wide-spread use. Out of this tension programmers generate ideas to improve the product base.

But the fact that engineers have first-hand knowledge of the application is just one aspect of the departure from traditional software development. Even more important is the mind-set of perpetual development. Traditional software products are finite by definition, with delimited scope and a predefined completion date. This is the basis for drawing the contract to produce the software, defining acceptance tests, and the problems that arise when projects fall behind schedule or overspend their budget.

Sites like Facebook will never be completed. The mindset is that the system will continue to be developed indefinitely.

Software that continues to evolve over long time periods actually exists in many domains. For example, the Linux operating system has evolved continuously since its first official release in 1994, growing 80-fold in the process [3]. However, new Linux versions are released two to three months apart. Internet-based companies like Facebook evolve at a much faster pace (Fig. 2).

The development rate is also reflected in the terminology used to describe it. In the context of

waterfall or unified process	evolutionary development	agile development	Facebook	continuous deployment
once	months	weeks	one day	<hour

Figure 2: *Timescales of making new developments available. Facebook typically deploys new code every day, balancing rapid development with foresight and monitoring.*

the waterfall model, the ultimate goal is *delivering* the software product. In the context of agile development or evolutionary systems such as Linux, we would speak of *periodic releases*. But the practices used by Internet companies have come to be known as *continuous deployment*. This reflects the habit of deploying new code as a series of small changes as soon as they are ready [5]. In such companies the software that provides the service resides on the company's web servers, thus deploying new software to the servers immediately makes it available to all users, without any need for downloads and local installation.

A direct result of perpetual development is that the software grows and grows. The codebase for Facebook's front end now stands at more than 10.5 million lines of actual code (without comment lines and blank lines), of which nearly 8.5 million are written in PHP. Moreover, the rate of growth is superlinear with time (Fig. 1). This contradicts Lehman's seminal work on software evolution which predicts that progress will be slowed down when size (and complexity) increase [7]. The contradiction may be explained as coming from different assumptions: Lehman assumed an essentially constant workforce, whereas Facebook enjoys a growing engineer base. The ability to rapidly grow the workforce indicates that the need for communication and coordination between engineers is probably not as restrictive as predicted by Brooks's Law. Similar superlinear growth trends have also been observed in some open-source projects, notably the Linux operating system kernel [4]. Specifically, our data regarding the Facebook codebase enjoys an excellent fit with a quadratic growth model[1], similarly to many open-source projects.

An important attribute of continuous deployment is that it facilitates live experimentation using A/B testing. The innovations implemented by engineers are immediately deployed, and real users can experience them. This enables a careful comparison of the new features with the base case (that is, the current site) in terms of their effect on user behavior [6]. While this typically involves only a small subset of users, at Facebook's volume of activity even a very small subset quickly generates enough data to assess the impact of the tested features. Thus engineers can immediately identify what works in practice and what does not.

A/B Testing

One important attribute of continuous deployment is that it facilitates live experimentation using A/B testing. The innovations that engineers implement are deployed immediately for real users to experience. This lets engineers carefully compare the new features with the base case (that is,

[1] $\text{LoC} = 317177 - 1148 \times \text{d} + 1.966 \times \text{d}^2$ where d is days since the first data point, with $R^2 = 0.996$. fitting was done on cleaned data (see Fig. 1)

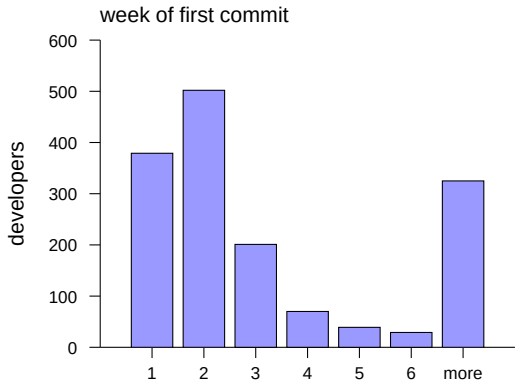

Figure 3: *Distribution of time from start of employment to first commit of Facebook bootcampers. Some employees in the 'more' category do not start with bootcamp right away, e.g., when transferring to engineering from a different department.*

the current site) in terms of how those features affect user behavior [6]. Although this typically involves only a small subset of users, at Facebook's volume of activity, even a very small subset quickly generates enough data to assess the tested features' impact. Thus, they can immediately identify what works in practice and what doesn't. A/B testing is an experimental approach to finding what users want, rather than trying to elicit requirements in advance and writing specifications. Moreover, it allows for situations where users use new features in unexpected ways. Among other things, this enables engineers to learn about the diversity of users, and appreciate their different approaches and views of Facebook. To improve the data obtained from tests, Facebook employs in-house usability tests with user focus groups in addition to testing the deployed product on a large scale [2].

Continuous Deployment

Continuous deployment also has important benefits from a software production viewpoint. Frequent deployments imply that each deployment introduces only a limited amount of new code. This reduces (but doesn't eliminate) the risk that something will go wrong. Frequent deployment approximates serial rollout, which is easier to debug; moreover, all commits are individually tested for regressions. All new Facebook employees undergo a six-week bootcamp in which they're encouraged to commit new code as soon as possible (see Figure 3), partly to overcome the fear of releasing new code. The ability to deploy code quickly in small increments and without fear enables rapid innovation. Another benefit of small and rapid deployments is that we can easily identify the source of and solutions to emerging problems: they're most likely the most recently deployed changes in the code, and still fresh in engineers' minds.

Ostensibly, rapid deployment is at odds with feature development that requires large changes to the codebase. The solution is to break down such changes into a sequence of smaller and safer

ones, hidden behind an abstraction

(a practice aptly called "branch by abstraction" [5]). For example, consider the delicate issue of migrating data from an existing store to a new one. This can be broken down as follows:

1. Encapsulate access to the data in an appropriate data type.
2. Modify the implementation to store data in both the old and the new stores.
3. Bulk migrate existing data from the old store to the new store. This is done in the background in parallel to writing new data to both stores.
4. Modify the implementation to read from both stores and compare the obtained data.
5. When convinced that the new store is operating as intended, switch to using the new store exclusively (the old store may be maintained for some time to safeguard against unforeseen problems).

Facebook has used this process to transparently migrate database tables containing hundreds of billions of rows to new storage formats.

In addition, deploying new code does not necessarily imply that it is immediately available to users. Facebook uses a tool called "Gatekeeper" to control which users see which features of the code. Thus it is possible for engineers to incrementally deploy and test partial implementations of a new service without exposing them to end users.

All front-end engineers at Facebook work on a single stable branch of the code, which also promotes rapid development, since no effort is spent on merging long-lived branches into the trunk. But there is still a distinction between code in development and code that is ready to be deployed. Developers use the git version control system locally for their daily work, until the code is ready to push. The stable version for deployment is maintained using subversion (for historical reasons). When ready to be pushed, new code must first be merged with the stable version in the centralized repository, after which engineers can commit their changes into subversion.

Given the rapid rate of development, it is not surprising that engineers typically commit new code several times each week (Fig. 4). Moreover, the typical intervals between successive commits by the same engineer are a few hours, with a median of 10 hours. However, the distribution of intervals is multi-modal, and intervals of a day or even multiple days also occur.

Determining the optimal deployment cycle in general is outside the scope of this paper. Some of the factors going into the decision are: the cost of each deployment, the probability and cost of errors, the probability and value of incremental benefits, the skill of the engineers involved, and the culture of the organization. Adding to the complexity of the decision is that many of these factors can be optimized, so the optimal cycle can change.

Some Internet companies allow all engineers to deploy their code immediately when they consider it ready, with no need for authorization by anyone else. This may lead to a rate of many new deployments per day. But for a company that handles large amounts of personal data like Facebook, the risk of privacy breaches warrants more oversight. Facebook therefore employs a combination of daily and weekly deployments, as described below.

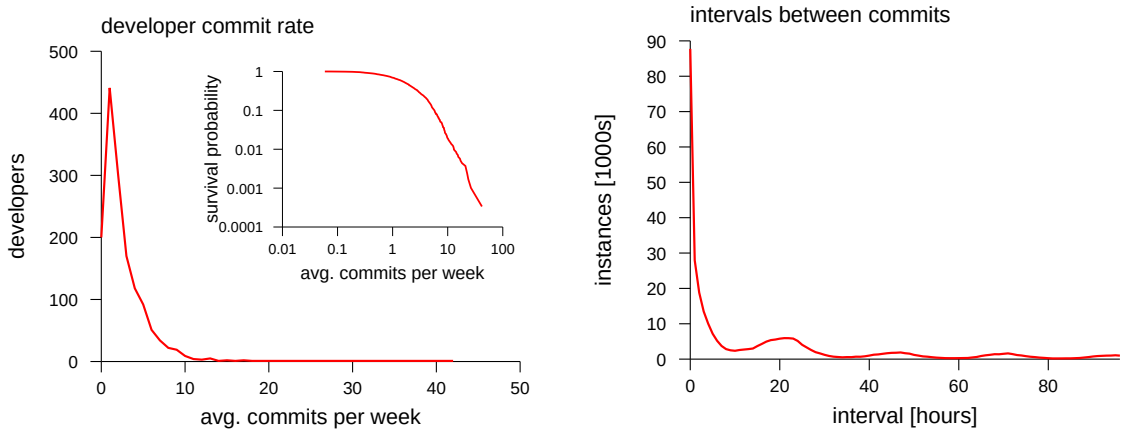

Figure 4: *Distribution of the commit rate of engineers with at least 10 commits (measured as number of commits divided by range of active weeks), and distribution of the intervals between commits by the same engineer. Inset shows the tail of first distribution, indicating that only about 1% of the engineers average more than 10 commits per week.*

2 Pushing New Features

The push process balances the rate of innovation with risk control. Development culture helps control risk just as much as do automated tools. The risks involved in introducing new software grow with scale, which has three main dimensions: more engineers, more lines of code, and more users. With more engineers, more gets done per unit of time, so more new code is generated for each push and must undergo testing. When the system is larger, more interactions occur between different components, and more things can go wrong. More users can employ the system in more ways and increase the volume of data that it must handle. Reducing the risks to zero is impossible, so Web companies must allocate oversight resources judiciously. For example, code concerned with privacy is held to a higher standard than code that deals with less sensitive issues.

Part of the allocation of oversight is the distinction between a daily push and the weekly push. The weekly push is the default, and involves thousands of changes. On Sunday afternoon the code to be pushed is placed in the subversion repository operated by the release engineers. It then undergoes extensive automatic testing, including tens of thousands of regression tests for correctness and performance. It also becomes part of the "latest" build, meaning it is the default version being used by Facebook employees. The push itself then occurs on Tuesday afternoon.

The release engineers responsible for the push process assign engineers with "push karma" based on past performance (namely how often their code caused problems). If an engineer has bad karma, his or her code contributions undergo more oversight before being accepted to the push. Importantly, the goal is to manage risk, not to rank performance, and push karma is not made public. Additional inputs affecting the amount of oversight exercised over new code are the size of the change and the amount of discussion about it during code reviews; higher levels for either of

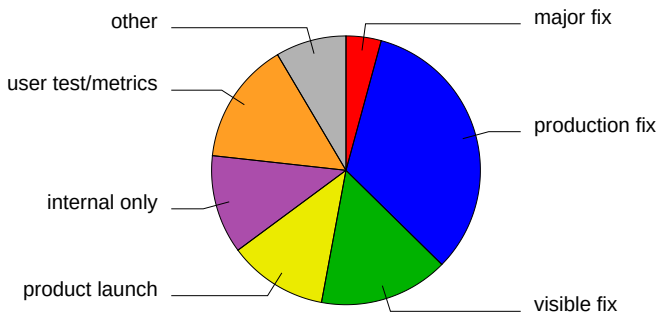

Figure 5: *Distribution of reasons for using a daily push.*

these indicate higher risk.

Release engineers perform a smaller push twice daily on other workdays, for several possible reasons (see Figure 5). In extreme cases, additional pushes might occur during the week or even over the weekend.

When code is accepted to the weekly or daily push, it should have already passed personal unit tests and a code review. At Facebook, code review occupies a central position. Every line of code thats written is reviewed by a different engineer than the original author. This serves multiple purposes: the original engineer is motivated to ensure that the code is of high quality, the reviewer comes with a fresh mind and might find defects or suggest alternatives, and, in general, knowledge about coding practices and the code itself spreads throughout the company. The Phabricator code review tool (`http://phabricator.org`) facilitates many common engineering operations on a large codebase. It enables engineers to:

- Browse current and historical versions of the source code.
- View suggested code changes and discuss them in-line.
- Bug and task tracking.
- Wiki-based documentation.

All these features are integrated with each other and with the source control system to reduce friction incidental to writing and committing code changes.

Engineers and release engineers conduct the code tests and administer a battery of regression tests, including on the user interface using Watir (`http://watir.com`) and WebDriver (`http://code.google.com/p/selenium`). In addition, Facebook employees effectively test the latest code while using it internally. This exercises the code under realistic conditions, and all employees can report any defects they encounter. A helpful property of having all employees double as testers is that as the number of code changes grows with the company, the number of testers follows suit automatically. The outcome of all this testing is increased confidence that the pushed code won't break the system.

Another important testing tool, Perflab, can accurately assess how the new code affects performance before its installed on production servers. Problems that Perflab or other tests uncover that engineers can't resolve within a short time might call for removing a specific code revision from the push and delaying it to a subsequent push, after engineers resolve the problems. Engineers must monitor and correct even small performance issues continuously, because if such problems are left to accumulate, they can quickly lead to capacity and performance problems. Perflab charts let the team visually compare the variance a code change introduces to the variance that's inherent in the existing product and identify emerging problems.

The weekly push itself occurs in stages. The first stage is deployment to H1, a set of internal servers accessible only to Facebook engineers. These servers are used for a final round of testing from the engineers who contributed code to the push.

The second stage is deployment to H2, a few thousand machines that serve a small fraction of real-world users. If the new code doesn't raise any alerts at H2, it's pushed to H3, which is full deployment on all servers. If problems arise, engineers will fix them, and the cycle repeats. Alternatively, the code might be rolled back to the previous version. Two kinds of rollback exist: The typical rollback reverts a single commit and any dependencies (which are few or nonexistent owing to the practice of small and independent commits, as well as the high frequency of commits and pushes). A much rarer rollback occurs when the entire binary must revert to the previous working version.

Facebook operates numerous servers in dozens of clusters spread across four geographical locations. Pushing a new version of the code to all these servers isn't trivial. The deployed executable size is around 1.5 Gbytes, including the Web server and compiled Facebook application. The code and data propagate to all servers via BitTorrent, which is configured to minimize global traffic by exploiting cluster and rack affinity. The time needed to propagate to all the servers is roughly 20 minutes. The Facebook site's responsiveness isnt affected when code is updated; rather, each server in its turn switches to the new version. A small amount of excess capacity helps facilitate the staggered transition.

As a matter of policy, all engineers who contributed code must be available online during the push. The release system verifies this by contacting them automatically using a system of IRC bots; if an engineer is unavailable (at least for daily pushes), his or her commit will be reverted. This means that the number of people on call is proportional to the number of code changes being pushed — again, ensuring that the process is scalable.

Note that for a large and complex application such as Facebook, it isn't always obvious whether a problem has occurred. For example, a small bug in the ranking function that wrongly prioritizes some newsfeed stories over others would be easy to miss. Facebook thus continuously monitors the system's health with a combination of internal tools such as Claspin (`http://www.facebook.com/notes/facebook-engineering/monitoring-cache-with-claspin/10151076705703`) and external sources such as tweet analysis.

As noted earlier, an important component of testing new features is testing them under real use — first, internal use by Facebook employees, and later use by subsets of real users worldwide. It's impractical to perform such testing by deploying code on all the servers and then removing it to stop the test, especially considering that hundreds of such tests could be occurring simultaneously.

Instead, the deployed code includes all that's been developed, both in production and under test, using Gatekeeper to control what code paths are actually active. Thus, engineers can turn tests on and off at will, and also apply them to only select user groups based on criteria such as country or age group. Gatekeeper can also be used to turn off new code that's causing problems, thereby reducing the need to immediately deploy a correction.

Gatekeeper also lets engineers conduct a dark launch, in which code is launched and installed on all the servers, but users don't see it because its user interface components are switched off. Such a launch can be used to test scalability and performance. For example, when Facebook introduced its chat server, it was initially deployed in a version that sent dummy chat messages without any user involvement. This stress-tested the chat servers under a realistic workload at scale, without users knowing about it. When the system was stable enough to support a real workload, the dummy messages were turned off and the user interface turned on.

3 Personal Responsibility

Facebook has roughly 1,000 development engineers and three release engineers who orchestrate the daily and weekly pushes. However, it doesn't have a separate quality assurance (QA) team or any other designated testers. In response to specific complaints, engineers can explore source code completely unrelated to their regular work, submitting fixes or at least detailed defect reports.

The absence of a separate QA team starkly contrasts with most traditional software companies, where engineers develop code and might also write and perform some basic unit tests, but then throw their code over the wall to the QA team. Such teams are composed of professional testers who write, maintain, and administer a whole battery of tests. This separation leads to various problems, including the need for testers to learn the code, and a perception of hierarchy in which development is regarded higher than testing.

At Facebook, engineers conduct any unit tests for their newly developed code. In addition, the code must pass all the accumulated regression tests, which are administered automatically as part of the commit and push process. As mentioned earlier, all new code must be supported by engineers attending the push on IRC in case problems occur with their code.

Developers must also support the operational use of their software — a combination that's become known as "devops." This further motivates writing good code and testing it thoroughly. Developers' personal stake in keeping the system running smoothly complements the engineering procedures and lets the system maintain quality at scale. Methodologies and tools aren't enough by themselves because they can always be misused. Thus, a culture of personal responsibility is critical.

Consequently, most source files are modified by only a few engineers (see Figure 6). Although at least one other engineer reviews all changes before they're committed, a third of the source files have only been edited by one engineer, and another quarter by two. Only 10 percent of the files are handled by more than seven engineers. On the other hand, the distribution of engineers per file has a heavy tail, with the most widely shared file handled by no fewer than 870 distinct engineers. These widely shared files are predominantly library files and also include major configuration and

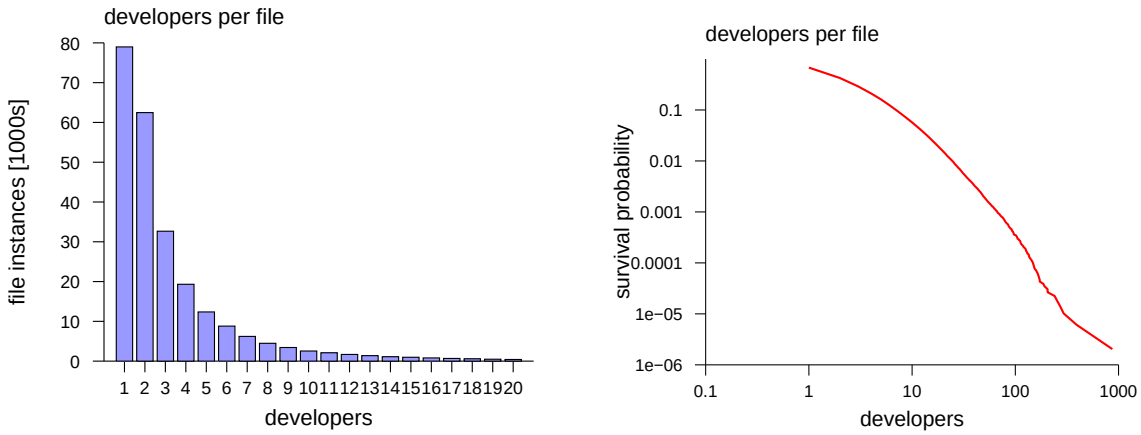

Figure 6: *Most files are handled by only few engineers. However, the distribution has a heavy tail: the probability that more than x engineers will handle a file (the "survival probability") drops off slowly with x.*

top-level PHP files.

Responsibility for personally developed code is just one aspect in a culture of mutual responsibility. Another comes from experimentation with alternative solutions to large-scale challenges. For example, when Facebook identified PHP's performance as a major factor in infrastructure cost, engineers proposed three different solutions with different risks and gains. Initially, All three were developed in parallel, but as more of a collaboration than a competition. In particular, the heads of the different teams identified when their projects were no longer worthwhile because another team's solution was proving to be better.

Eventually, the most ambitious alternative prevailed (producing the HipHop compiler; `http://github.com/facebook/hiphop-php`), but the other two werent a waste: they provided important backup capability while needed and were terminated soon after it was evident that a better option was viable.

In another stark break from traditional practices, even work assignment at Facebook is personally driven by the engineers. All new engineers first undergo bootcamp, where they become acquainted with Facebook's codebase, culture, and processes; then they choose to join the team where they feel they can play to their strengths and enjoy the work, while aligning with the company's priorities, not unlike open-source projects [8]. Naturally, it is also possible to move between teams. One mechanism supporting team mobility is the hackamonth, whereby engineers join another team for several weeks of work on new ideas in that team's domain. Subsequently, they can officially join the team.

On a smaller scale, innovations are encouraged by breaking the routine with frequent, day-long hackathons. Such break-out time occurs in other companies as well — for example, Google lets engineers spend 20 percent of their time on projects of their choice. Facebook hackathons are focused and intensive, and foster interactions among all parts of the company — not just engineers

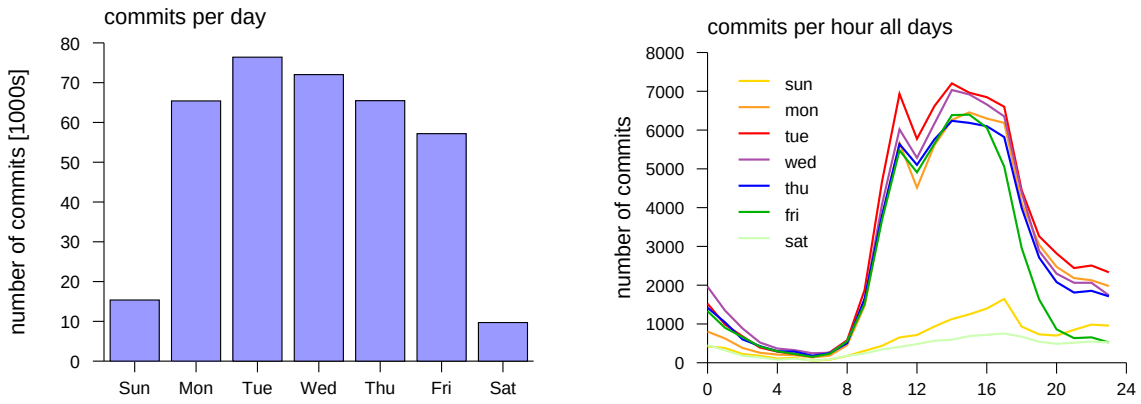

Figure 7: *Sustainable work practices as reflected by the distribution of committing new code on different days of the week and different hours of the day.*

but also finance, legal, and other departments. Many prominent Facebook features began during hackathons, including Timeline, chat, video, and HipHop.

The flip side of personal responsibility is responsibility toward the engineers themselves. Due to the perpetual development mindset, Facebook culture upholds the notion of sustainable work rates. The hacker culture doesn't imply working impossible hours. Rather, engineers work normal hours, take lunch breaks, take weekends off, go on vacation during the winter holidays, and so on (see Figure 7). In particular, daily code pushes arent scheduled for weekends.

4 Summary

Software development at Facebook runs contrary to many of the common practices of the industry. The main points we have covered include:

- There is no detailed plan to achieve a final, well-specified product.
- Engineers work directly on a common codebase with no branches and merging.
- There is no separate QA team responsible for testing.
- New code is released at a high rate, currently twice every working day.
- Engineers self-select what to work on.
- There is no assignment of blame for failures.

But this does not reflect a lack of regard to established procedure. Rather, it is a willful adjustment and optimization of the software development process to the unique circumstances at Facebook:

- The product cannot be specified in advance, and it must evolve continuously at a rapid pace.

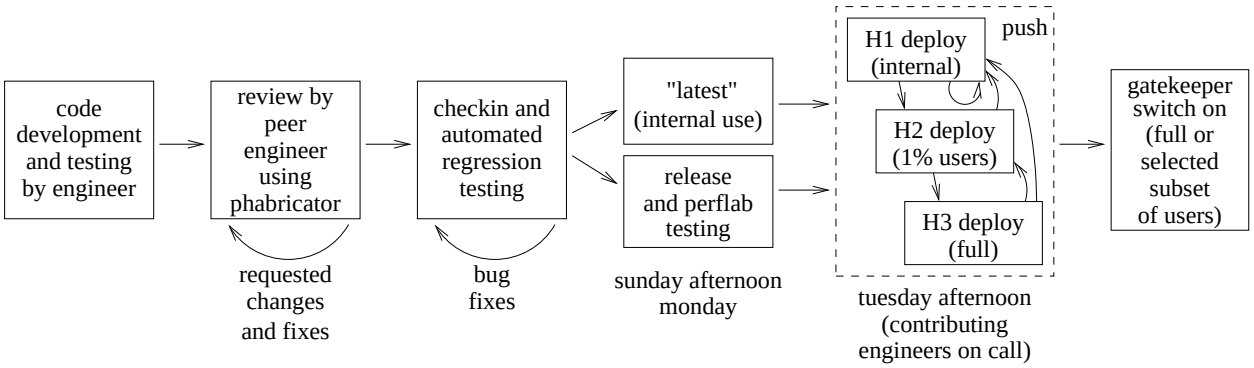

Figure 8: *Facebook's version of the deployment pipeline, showing the multiple controls over new code.*

- Engineers have first-hand experience in the domain, but also need to test innovations on real users to see what works.
- Personal responsibility by the engineers who wrote the code can replace quality assurances obtained by a separate testing organization.
- Testing on real users at scale is possible, and provides the most precise and immediate feedback.
- Learning from experience is more important and beneficial than chastising those responsible for a failure.

Importantly, all these practices aren't just a disjoint set, but rather gel into a coherent engineering culture that combines with a process to provide considerable oversight on new code (see Figure 8). Together, these practices balance the need for quick turnaround with that for oversight, robustness, and correctness. Although some practices are unique to Web-based companies such as Facebook, others are applicable in general. Indeed, the practices Facebook follows have much in common with agile software development.

Perhaps the biggest surprise is how far individual responsibility can substitute for specialization, methodologies, and formalized procedures. Practices chosen to make up for blame and self-protection have no place in a team of engineers willing to take responsibility for the entire system. The time and energy liberated by taking a positive, responsible approach to software development has touched the lives of more than a seventh of the planet.

Acknowledgments

We would like to thank Chuck Rossi, Boris Dimitrov, and Facebook's communication team for their insightful comments.

To Read More

On-line sources about Facebook's software development practices include the following:

- Jolie O'Dell, *Move fast, break things: Four stories for hackers from Facebook (interview with Jay Parikh)*, 26 Jun 2012. `http://venturebeat.com/2012/06/26/facebook-hacker-stor`
- Andrew Bosworth, *Facebook Engineering Bootcamp*, 19 Nov 2009. `http://www.facebook.com/note.php?note\_id=177577963919`
- Steven Grimm, *Facebook Engineering: What kind of automated testing does Facebook do?*, 29 Jun 2010. `http://www.quora.com/Facebook-Engineering/What-kind-of-automat`
- Mike Schroepfer, *Culture of Innovation*, Nov 2010. `http://www.youtube.com/watch?v=DfN1YaYdgRg`
- *Release engineering and push karma*, interview with release engineer Chuck Rossi, 5 Apr 2012. `https://www.facebook.com/notes/facebook-engineering/release-enginee10150660826788920`

References

[1] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. P. zny. Workload analysis of a large-scale key-value store. In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pages 53–64, Jun 2012.

[2] P. Chilana, C. Holsberry, F. Oliveira, and A. Ko. Designing for a billion users: A case study of Facebook. In *SIGCHI Conf. Human Factors in Comput. Syst.*, pages 419–432, May 2012.

[3] D. G. Feitelson. Perpetual development: A model for the Linux kernel life cycle. *J. Syst. & Softw.*, 85(4):859–875, Apr 2012.

[4] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *16th Intl. Conf. Softw. Maintenance*, pages 131–142, Oct 2000.

[5] J. Humble and D. Farley. *Continuous Delivery*. Addison-Wesley, 2010.

[6] R. Kohavi, R. Longbotham, D. Sommerfield, and R. M. Henne. Controlled experiments on the web: Survey and practical guide. *Data Mining & Knowledge Discovery*, 18(1):140–181, Feb 2009.

[7] M. M. Lehman, D. E. Perry, and J. F. Ramil. Implications of evolution metrics on software maintenance. In *14th Intl. Conf. Softw. Maintenance*, pages 208–217, Nov 1998.

[8] E. S. Raymond. The cathedral and the bazaar. URL www.catb.org/˜esr/writings/cathedral-bazaar/cathedral-bazaar, 2000.

[9] Royal Pingdom Blog. Exploring the software behind Facebook, the world's largest site. URL http://royal.pingdom.com/2010/06/18/the-software-behind-facebook/, 18 Jun 2010. (Visited 27 Sep 2010).

[10] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sharma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at Facebook. In *SIGMOD Intl. Conf. Management of Data*, pages 1013–1020, Jun 2010.