Top 10 Adages in Continuous Deployment

Chris Parnin, Eric Helms, Chris Atlee, Harley Boughton, Mark Ghattas, Andy Glover, James Holman, John Micco, Brendan Murphy, Tony Savor, Michael Stumm, Shari Whitaker, Laurie Williams

Facebook has seen the number of developers increase by a factor of 20 over a 6 year period while the code base size increased by a factor of 50. During that period, the developer productivity remained constant as measured in lines per developer. They attribute much of this success to their continuous deployment practice. Continuous deployment is a software engineering process where incremental software changes are automatically tested and frequently deployed to production environments. With continuous deployment, the elapsed time for a change made by a developer to reach a customer can now be measured in days or even hours. Such ultra-fast changes create a new reality in software development.

To study this new reality, a one-day Continuous Deployment Summit was held at the Facebook campus in July 2015. The purpose of the Summit was to share best practices and challenges in transitioning to the use of continuous deployment practices. The Summit was attended by 1 representative from each of 10 companies and facilitated by researchers. These companies were: Cisco, Facebook, Google, IBM, LexisNexis, Microsoft, Mozilla, Netflix, Red Hat, and SAS. Instagram, a Facebook company, started the Summit with a keynote presentation on their continuous deployment journey. The companies represent a spectrum from continuous deployment pioneers whose have mature continuous deployment implementations to those with architectural baggage necessitating a multi-year transition to the use continuous deployment practices. Indeed, the companies ranged from having products that deploy 1000 times per day to those who deploy once or twice per year. The factor that united these 10 companies is that all strive to leverage faster deployment to deliver higher quality products to their customers ever faster, using advanced analytics to translate a deluge of available telemetry data into product improvement information.

The rest of this paper summarizes the workshop and presents 10 adages that emerged during the Summit.

Continuous Deployment Summit @ Facebook

We conducted a pre-summit survey that was answered by 17 teams from 9 of the 10 companies. The questions on the survey focused on which continuous deployment practices each company used, the benefits and challenges realized by the company due to the use of these practices, and on areas of technical challenge.

Figure 1 summarizes the practices used by the companies. Participants were presented with 11 common practices used with continuous deployment and responded with how often the practice was used. While companies had the option of indicating partial usage of practice, none actually indicated partial usage. Either they used the practice all the time, not at all, or were not sure.

As shown, the most frequently used practices used were automated unit testing, staging, and branching. Code review was also often used as a manual signoff in an otherwise highly

automated deployment process. We have observed a resurgence of the code review practice, now often handled through light-weight distributed tools, as engineers are more motivated to have others view their code before it is rapidly deployed and defects become public. The change ownership practice, also commonly used, causes engineers to be “on call” for issues related to the implications of their own defects, rather than having a separate field support group bear the brunt of all defects. As a result, engineers are likely to be more motivated to deploy high quality software.

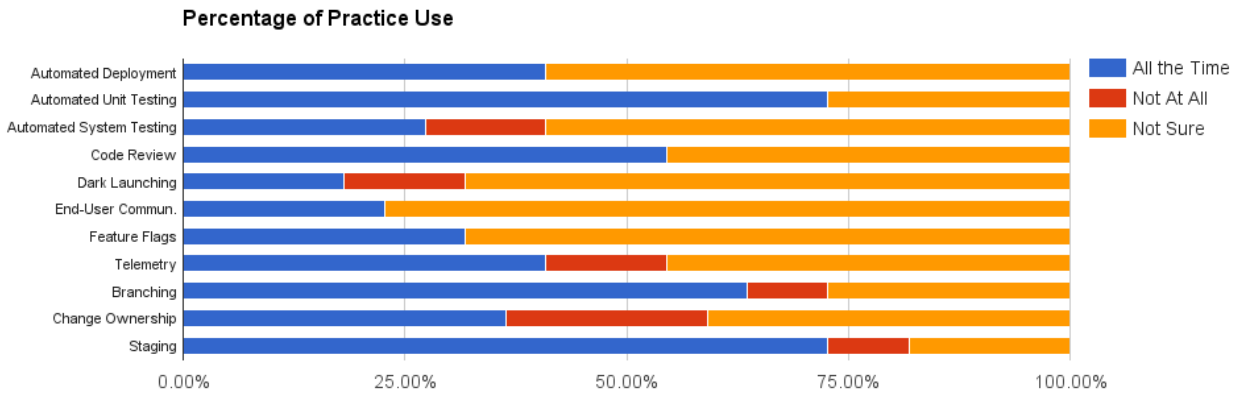

Figure 1: Percentage of Respondents Indicating Practice Use (N=17)

Respondents were asked about the benefits they realized for adopting continuous deployment practices. The most prevalent benefits were improved speed of feature delivery, improved quality, and improved customer satisfaction. Also, employees were happier with more rapid customer feedback on their work and with reduced stress. While having defects become public with rapid deployment might cause more stress, the tradeoff is that the stress of “missing the release” diminishes with when the next “release train” leaves the station soon. Management felt decisions were more data-driven with rapid feedback. Teams also cited higher productivity and better overall collaboration.

Adages

We now present the 10 adages that emerged as a concept during the Summit. While none of these adages applied to all 10 companies, all agree with these concepts.

1. Every feature is an experiment

Jez Humble argues that a key to running a lean enterprise is to "take an experimental approach to product development" [4]. In this view, no feature is likely to persist for very long without data justifying its existence. Previously, features choices were carefully considered and traded off. Those chosen were designed, built, and then delivered. Evidence rarely supported decisions. With continuous deployment, every planned feature is treated as an experiment, of which some deployed features are allowed to die. For example, on Netflix.com if not enough people hovered over a new element, then a new experiment might move the element to a new location on the screen. If all experiments showed a lack of interest, the new feature would ultimately be deleted.

Companies at the Summit reported using several supporting practices. In general, statistics are collected on every aspect of the software. Performance and stability metrics are recorded, such as page render times, database column access, exceptions and error codes, response times, and API method call rates. To collect all this information, the architecture of the software must be designed with a telemetry-first mindset. Instead of keeping localized copies of performance and error logs on a server, metrics are streamed to a centralized in-memory data store. Finally, to support data analytics, several companies employ a large staff of data scientists, reaching as much as a third of engineering staff. A rich set of data exploration tools, including custom data query languages, are created and used.

However, several challenges exist. Some companies quickly outgrew the infrastructure for storing data related to metrics. Netflix reports it initially collected 1.2 million metrics related to its streaming services, but that soon ballooned to 1 billion metrics. Not only could the in-memory data store no longer keep up, but they had to more carefully consider what data was essential for experimentation. Additionally, engineering queries to extract relevant information related to a feature is complex. One Summit participant remarked, “You need a PhD to write a data analysis query”. Significant investment in both telemetry and analytics is needed. Investing in these efforts separately can be costly: participants discussed situations when enormous amounts of data was collected, but because one forgotten essential data point was missing, the entire experiment had to be redone. Finally, not every feature necessarily warrants full experimentation, especially non-user facing features such as those related to storage. Additionally, privacy implications of data collection must be carefully considered.

Moving forward, a general challenge that companies will face is how to establish a culture of feature experimentation. How can we enable teams to consistently collect targeted information throughout the entire lifecycle of a feature without introducing too much overhead or process?

2. **Cost of change is dead**

The cost to change code in production is surprisingly cheap to make in stark contrast with the predictions that fixes in deployed software would become exponentially more expensive. In 1981, Barry Boehm [1] showed that the cost of change increases tenfold with each software development phase. For example, if the cost to fix a change in the coding phase is $100 then in testing it would cost $1000, and in production this change may cost $10,000.

With continuous deployment, the time between development and defect discovery in production is typically very short, on the order of hours or days. For example, a developer pushes a new feature into production after two days of work. The next day, a defect is reported by a user. A fix should be efficient because the developer just finished and can remember what they just did. With continuous deployment, all development phases happen the “same day” by the same person(s) and this exponential cost increase would not happen. As a result, the cost of change curve is flattened. Thus, the $100 cost to fix a change in development is the same $100 cost if found in production.

At Google, they have found that the scope of changes to review when troubleshooting is small which makes narrowing the down the culprit easier and quicker. As well, by deploying changes into a production environment the feedback loop is quicker with respect to making the development team more aware of release process challenges. At Facebook, a developer must confirm via their in-house chat system that they are on standby or their change will not go live during one of two daily production rollouts. Thus, all developers with outgoing changes are able to react to any bugs found minutes after going live. Across the Summit attendees, discussion of the cost of changes was nearly non-existent, indicating that the effects are minimized when compared to other cost concerns.

With more traditional release and deployment models, code is subjected to rounds of quality assurance in an attempt to flush out any defects. If the release cycle is 3-6 months, newly-found defects may not be addressed for users for another 3-6 months. Even shorter maintenance cycles are still orders of magnitude longer than daily deployments. The practice of continuous deployment allows deploying new features and defect fixes at speed.

Continuous deployment does not guarantee a defect will be found immediately. If it's found later, the cost of change is the same as before. However, if a defect is not found for a long time, it is more likely to be in a low usage feature.

3. Fast to deploy, slow(er) to release

Deploying code into production does not necessarily mean user-facing features are available for customer use right away. Sometimes, a new feature may be deployed and evaluated in production for several months before being publically released. For example, at Instagram an engineer may want to build a new feature for threading messages on picture comments. In this case, by deploying code to production, the engineer can evaluate and test the feature in a live environment by running the code but keeping the results "invisible" by not enabling the new feature in the user interface. This practice, called a *dark launch*, allows the engineer to slowly deploy and stabilize small chunks directly in production without impacting the user experience. After stabilization, the engineer can "turn on" the feature and release it.

Companies described several techniques and reasons for not always being so quick to release. At Instagram, dark launches are often used to deploy and stabilize features for up to six months before being officially released. At Microsoft, large architectural changes are often deployed using a combination of feature flags and dark launches. With a feature flag, a feature is deployed but disabled until it is ready for release, which a developer turns on and off via a configuration server. The benefit of this practice is that it allows them to avoid integration issues or maintain long-running feature branches. By simply deploying changes *early, often, and frequently* in production, the overall deployment friction is reduced.

However, many challenges exist with this approach. Dynamic configuration enables developers to quickly react to problems by disabling a feature, but a developer can just as easily cause an outage by inadvertently entering an invalid configuration state. Many companies reported that while code changes went through a rigorous testing and analysis pipeline, sufficient tooling to

test and evaluate configuration changes at the same level of rigor may not have been available. Cleaning up and removing unneeded feature flags is a highly variable practice which often contributed to technical debt. For some Summit companies, creating a duplicate production environment, or *shadow infrastructure,* is too expensive or complicated. They are forced to do testing in production, even if not strictly desired.

Many techniques can control the speed in which a customer sees new changes. A company can release software slowly while still deploying everyday. Extra engineering effort must be put in place to make sure delayed release strategies and testing in production do not impact the user experience.

4. **Investing for survival**

Survival in today's market means investing in tooling and automation. Practices that were once seen as best practices or measures of maturity in software engineering are now the backbone of process that relies on deploying rapidly. Automated system testing used to be a means to run large test suites to verify enterprise applications had not regressed between releases. Now, these system tests are necessary so that developers can get quick feedback, and so that releases can be automated to accept or fail a patch. This tooling enables small teams to manage large infrastructures.

Companies at the top of the continuous deployment spectrum, such as Instagram and Netflix, point to tooling as paying massive dividends. Facebook has found that a small team, focused on tooling and release automation, can empower a much larger team of feature-focused developers. Instagram uses automation to enforce process. Tooling investment allows capturing common workflows and tasks into repeatable, runnable operations that developers or automated systems can perform. By capturing process in tools, the processes can be tested, versioned and vetted like production code.

Instagram faced challenges with partial automation of a process. Partially automating a process has the risk of developers being unaware of implicit steps needed in the deployment process. For example, if a developer forgets to manually obtain an operation lock on a service (via another tool) before running a deployment command.

Practitioners are seeing that to stay competitive, to survive, software engineering best practices such as automated unit testing are now a must. Providing a superior product is now coupled to the speed at which enhancements can be deployed. This change requires companies to have strategic investment in automation as the scope and scale changes over time.

5. **You are the support person**

The developer has the power and freedom to deploy changes at their own behest. With great power comes great responsibility. If code breaks in production, whose responsibility is it: the developer or the operations team? Traditional software methods encourage siloed responsibility. Developers "throw code over the wall" to quality assurance (QA) teams who then throw it over another wall to operations teams. Several Summit organizations discussed

developers that code and don't stop to understand requirements, user stories or what production environments look like. By owning a feature or code change from inception to deployment, the burden is on the developer. This burden means that when things break, the developer is the one getting the support call and responsible for fixing the issue, no matter the time of day.

Developers owning changes "from cradle to grave" means that traditional team structures need to change. Netflix has no dedicated operation teams. While functional roles still exist, such as QA or operations, these roles are embedded in development teams, creating hundreds of loosely coupled but highly aligned teams. Instead of teams dedicated to a particular functional role (e.g. QA, operations, development), teams have a representative cross section of necessary roles. Instagram has found value in having teams with members that focus on areas but are, as a team, ultimately responsible for the life of a feature. Both Instagram and Red Hat have employed the idea of support rotations where each member of the team spend some amount of time in a customer support role, which results in shared pain.

Giving teams autonomy comes with challenges, for example, how do autonomous teams integrate with one another in a reliable manner? Netflix achieves this integration through a microservices architecture that requires teams to build APIs that they maintain and ensure are stable from change to change. Google enforces team service communication through a common API type and through a defined data type that all services are required to use. By defining standards for communication, teams are then free to build what they need to accomplish their task however they feel will be most efficient for them.

From an organizational standpoint, how do teams migrate to this new view of the world? LexisNexis has seen that traditional organization structures mean that different teams report to different parts of the organization with different goals, which makes integrating them that much harder. Further, other areas that require change make tackling the team and the ownership aspect hard such as manual tests and resource constraints. The role of the developer is changing to be less horizontal and more vertical, increasing responsibility but at the same time empowering the developer understand the impact of their changes.

6. **Configuration is code**

Practitioners of continuous deployment are finding that, at scale, not treating configuration like code leads to a significant percentage of production issues. Traditionally configuration has been thought of as a runtime consideration managed by an operations team or system administrators. Changes are made to servers live and often in a one-off fashion that can lead to server drift. For example, an engineer is experimenting with optimizing query speeds for the database and changes the configuration on one of the database boxes. Now this change has to be replicated to four database servers. When multiple servers are intended to represent the same application, having even one undergo configuration drift can lead to unknown or difficult-to-debug breakages.

Modern configuration management tools, such as Ansible, Puppet, Chef and Salt, allow configuration management to be scripted and orchestrated across all server assets.
The new normal for organizations is that managing configuration should be treated the same as managing features and code. For example, at Netflix, for every commit, the build process creates a debian package completely specifying the dependencies needed and then installs them within a newly created Amazon Web Services virtual machine image.

Both Facebook and Netflix noted that despite tooling, configuration changes can still cause difficult-to-debug errors. Netflix does 60,000 of these types of changes a day and has no system for tracking or reviewing those changes. This leads to, as Netflix would put it, shooting themselves in the foot often. Teams at Red Hat have found that, just like large code bases, large configuration suites can be become unruly.

The lesson that can be learned from those at the far end of the continuous deployment spectrum is to consider configuration management from the inception of a new project or the transition of projects with architectural baggage to a continuous deployment model. In other words, configuration management should be a core competency that is treated like code. Treating configuration like code implies using all of the best practices around coupling, cohesion, continuous integration and scale.

7. **Comfort the customer with discomfort**

Not everyone will be ready for fast delivery of software. As companies transition to continuous deployment of their software, they are experimenting with multiple ways for comforting customers with the new pace of delivery. In today's consumer world, as products and devices receive a constant stream of updates, customers often have no choice but to accept them. New generations of customers may, in fact, expect them. If mobile devices are training all of us to receive constant change and even cars and televisions are automatically updating themselves, why not business software? The number of customers willing to wait a year or two for updates will rapidly dwindle. Still, not all customers and companies are ready for this change.

One prominent example of this challenge comes from Microsoft's efforts in releasing Windows 10. Microsoft has shifted from a policy of large, but infrequent updates, to a model of making regular incremental improvements to its operating system. The efforts to migrate users to Windows 10 also been notably more proactive: including prefetching installation files, frequently prompting users to upgrade, and restricting ability to opt-out of updates. These changes may appeal to more savvy customers but burden enterprise customers who may be unwilling or unable to accept frequent changes due to internal integration testing and regulatory concerns.

To comfort customers with changes, one strategy IBM and Mozilla used was to include important stakeholders in unit and integration tests during development. This reduced risk in failed deployments that would occur on premise and help stakeholders feel more comfortable accepting new releases. Cisco has been exploring the idea of using more rapid deployments a model for co-invention with customers.

Often the biggest source of discomfort for customers is the disruption to productivity when upgrading versions. For example, at IBM, the time to migrate one system to a new version would take one entire month while at a customer's site. The primary challenge was coordinating code and databases changes with on-premise instances. Eventually, IBM was able get the process down to one hour. Similarly, at SAS, the biggest deployment barrier was that each deployment required large periods of downtime for the customer and having to support many versions of datasets and deployed systems.

When moving at speed, companies must consider whether they are moving faster than users desire. Still, the best comfort a company can provide is the ability to deliver a change at a moment's notice, whenever the customer *is ready*.

8. **Looking back to move forward**

Continuous deployment requires continuous reflection on the process of delivery.
Almost every company at our Summit had a story about one time bringing down entire operations with accidental mistakes in configuration changes. For example, a malformed json setting once brought down the entire discovery component of Netflix's architecture. To support reflection on production failures, retrospectives were employed by all the companies using continuous deployment. *Retrospectives or post-mortems* are a practice where team members discuss the causes and consequences that resulted from an unexpected operation outage or deployment failure. Potential process changes are also discussed.

Several of the companies described their experiences in performing retrospectives. At Netflix, outages are reported as issues in an issue tracker and rated on severity. By tracking outages, a meta-analysis of outages can be performed to uncover trends and systematic issues with deployment processes. More severe outages are discussed at weekly retrospectives, which are attended by multiple stakeholders across teams. Several companies mentioned that despite its usefulness, the process of holding retrospectives can be quite dreadful. Developers find it hard to hear about the impact a coding mistake had on users and it is difficult to factor out emotions from the discussion. Mentioning victories can help maintain team morale and ease raw emotions. In certain circumstances, it makes sense to leave the responsible party outside of the room, if possible. Still, despite the unease that can be associated with a post-mortem review, several companies observed a considerable drop in errors after starting having post-mortem reviews.

Team practices are not the only thing that changes. As a result of retrospectives, cultural views can shift. At its inception, developers at Facebook were first instilled with a culture of "Move fast, break things". However at a certain point, that message was taken too far, and a new moderating creed emerged, "Slow down and fix your shit". Other companies have reflected on the benefits of a particular practices and have sought data to back them up. For example, after a series of extensive studies of code reviews at Microsoft, the conclusion of the studies has so far not found any significant benefit in reducing defects. Instead, the primary benefits involve knowledge sharing and improving onboarding.

As companies continue to adopt continuous deployment practices, there is a need to not only calibrate how a practice is exercised and how a culture is defined but also to constantly question the benefits and effectiveness of having those items in place in the first place. An important aspect of holding retrospectives is to maintain a “blameless culture”, but this can be difficult when developers are also expected to be fully responsible for their deployed changes from “cradle to grave”.

9. **Inviting PrivSec in**

Most Summit companies indicated that privacy and software security were silos, the responsibility of a security and/or privacy group, not the responsibility of all the developers involved in implementing deployable software. Continuous deployment may increase risk because privacy and security experts cannot review every change that is rapidly deployed.

In the waterfall and spiral software development days, developers would “throw the code over the wall” to testers to deal with defects. With the advent of agile methodologies, testers were “invited to the table” and participated as partners from the beginning of an iteration. Together, then, the developers/testers threw their tested products “over the wall” to operations to deploy the product. With continuous deployment, operations is now “invited to the table,” participating throughout an iteration on the operations implications of feature development. Those in the security and privacy silos are often not invited to the table. Means of establishing collaboration between security teams and development and IT teams has been discussed since 2012 [7]. This collaboration is most commonly referred to as DevSecOps. We propose to go further and explicitly include privacy (PrivSec) and invite these considerations “at the table” to be involved throughout development (DevPrivSecOps). The invitation can both be in the form of increasing security knowledge to developers, testers, and operations as well as through increased partnership of privacy and security experts, breaking down the silos.

Companies can have a separate process and/or oversight for changes with higher security risk and/or that have privacy implications. At Facebook, a code change that is considered to have privacy implications may go through a weekly push process rather than a daily push process [3]. Additionally, a small team creates an access layer for all data and controls that forces the adherence to privacy and regulatory concerns. Google discussed specific controls they have put in place for secure deployment, such as authorizing users who check in code for deployment, strict access control, and checksumming binaries. Google also has a strict division between their production network and their company network. The production network consists of servers only, no workstations to reduce the ability for tampering with deployed code.

10. **Ready or not, here it comes**

Your competitor continuously adds value to their products. Do you? The Summit companies all indicated an urgency for rapidly delivering new functionality to remain competitive. According to a 2015 global survey conducted by CA Technologies indicated that of 1425 IT executives, 88% have adopted DevOps or planning to adopt DevOps in the next five years [2]. DevOps and continuous deployment share similar practices; some informally equate the two practices. According to a survey conducted by Puppet Labs that was answered by 4,976 respondents, IT

organizations that have adopted DevOps experienced 60 times fewer failures, and deploy 30 times more frequently than organizations that have not adopted DevOps [5]. The survey respondents indicate widespread adoption of DevOps throughout the globe and in organizations of all sizes. The top five domains that use DevOps practices are technology, web software, banking and finance, education, and telecommunications. The prevalence and growth of DevOps would only be possible if performance indicators supported business benefits for its use [2, 5]: an increase in customers; collaboration across departments; improved software quality and performance; faster maintenance times.

Software engineering educators must also stand at attention. In the words of Brian Stevens [6], former EVP and CTO at Red Hat, "The legacy model of software engineering just isn't going to survive this transition …" Software engineering education often lags behind the new reality of continuous deployment and focuses on the legacy model. Some fundamental skills that need to be taught in core undergraduate software engineering education include continuous integration and build; automated integration and system testing; and the need follow good validation and verification practices if you don't want to be woken up in the middle of the night to fix the code you deployed that day. Additionally, undergraduates should made aware of the realities of deploying into a live, larger-scale environment, and its related concerns such as data migration, deployment strategies, deployment pipeline, and telemetry coding patterns.

Ready or not, here it comes. Will you be ready to deliver?

Glossary/Sidebar

continuous deployment: A practice where incremental software changes are automatically tested, vetted, and deployed to production environments
configuration management (CM): A process where an inventory of software and production assets are controlled via package managers and tools such as Ansible.
dark launch: A practice where code is incrementally deployed into production but remains invisible to users.
feature flag: A mechanism for dynamically enabling features in production, often controlled by a global in-memory store and cached locally in service instances.
telemetry: A practice where code is instrumented in order to compute metrics about feature usage and software performance and stability.
canary release: A practice where code under test is first released to a small batch of real users. If metrics deviate from nominal ranges, then routing to canary may be automatically halted.
microservices: An architectural style where services are created as small and often stateless instances and connected together via a central discovery service and property store.

retrospectives: a practice where team members discuss the causes and consequences that resulted from an unexpected operation outage or deployment failure.

References

[1] B. W. Boehm, *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.

[2] CA Technologies, "DevOps: The Worst-Kept Secret to Winning in the Application Economy," no. October 2014. http://www.ca.com/us/~/media/Files/whitepapers/devops-the-worst-kept-secret-to-winning-in-the-application-economy.pdf

[3] D. G. Feitelson, E. Frachtenberg, and K. L. Beck, "Development and Deployment at Facebook," *IEEE Internet Computing,* vol. 17, no. 4, pp. 8-17, July 2013.

[4] J. Humble, , , 2015, *Lean Enterprise: How High Performance Organizations Innovate at Scale*. Sebastopol, CA: O'Reilly Media, 2015.

[5] Puppet Labs, "2015 State of DevOps Report," https://puppetlabs.com/2015-devops-report

[6] B. Stevens, "Red Hat Summit Keynote," https://www.youtube.com/watch?v=8B56mdobgZE (note: at 26:50)

[7] J. Turnbull, "DevOps and Security," no. 2012. http://www.slideshare.net/jamtur01/security-loves-devops-devopsdays-austin-2012.