DevOps Case Studies

The Journey to Positive Business Outcomes

DevOps Enterprise Forum 2016

DevOps Case Studies

The Journey to Positive Business Outcomes

MODERN TECHNICAL PRACTICES

DEVOPS ENTERPRISE FORUM
2016
IT REVOLUTION
PORTLAND, OREGON

IT REVOLUTION

25 NW 23rd Pl, Suite 6314
Portland, OR 97210

DEVOPS CASE STUDIES: The Journey to Positive Business Outcomes

First Edition

Produced in the United States of America

10 9 8 7 6 5 4 3 2 1

Cover design and interior by Abbey Gaterude
Cover illustration by Mammoth

Table of Contents

PREFACE

In May of this year, IT Revolution once again had the pleasure to host 50 technology leaders and thinkers from across the DevOps Enterprise community at the DevOps Enterprise Forum in Portland, Oregon. The Forum's ongoing goal is to create written guidance, gathered from the best experts in these respective areas, for overcoming the top obstacles in the DevOps Enterprise community.

Gathering feedback and information from the 2015 DevOps Enterprise Summit, we narrowed down the four key areas identified by the community to tackle in this this years Forum papers:

- **Leading Change**: What are effective strategies and methods for leading change in large organizations?
- **Organization Design**: What do the organization charts look like for organizations successfully adopting DevOps look like? What are the respective roles and responsibilities, and how has it changed from more traditional IT organizations?
- **Modern Technology Practices**: What are modern architectural and technical practices that every technology leader needs to know about?
- **Compliance and Security**: What are concrete ways for DevOps to bridge the information security and compliance gap, to show auditors and regulators that effective controls exist to prevent, detect and correct problems?

For three days, we broke into groups based on each of the key areas and set to work, choosing teams, sometimes switching between teams, collaborating, sharing, arguing… and writing. After the Forum concluded, the groups spent the next six months working together to complete and refine the work they started together.

The end result can be found on the Forum page at IT Revolution web site (http://itrevolution.com/devops_enterprise_forum_guidance) and all the forum papers, from both this year and last year, are free to the community.

IT Revolution is proud to share the outcomes of the hard work, dedication, and collaboration of the amazing group of people from the 2016 DevOps Enterprise Forum, our hope is that you will gain valuable insight into DevOps as a practice.

Gene Kim
November 2016
Portland, Oregon

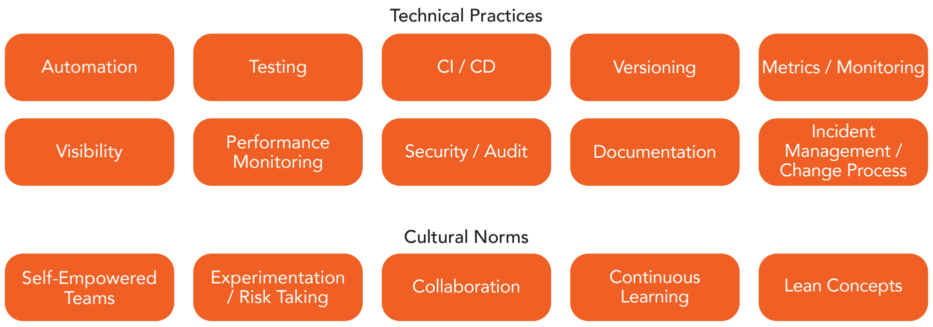
EXECUTIVE SUMMARY

Technology leaders today face a host of issues as IT organizations work to keep up with changing demands, complex business goals, and the need to deliver faster services and results in ever-changing business and technology environments. The goal of this paper is to provide the reader with an understanding of the key modern architectural and technical practices and, more importantly, some patterns of adoption that are successful ways to accelerate any DevOps practice. Modern technology practices, when implemented well, can bring a whole host of benefits. In addition, we will walk the reader through a few case studies that map the journey from adoption of these key modern tech practices to positive business outcomes.

GIVING CONTEXT

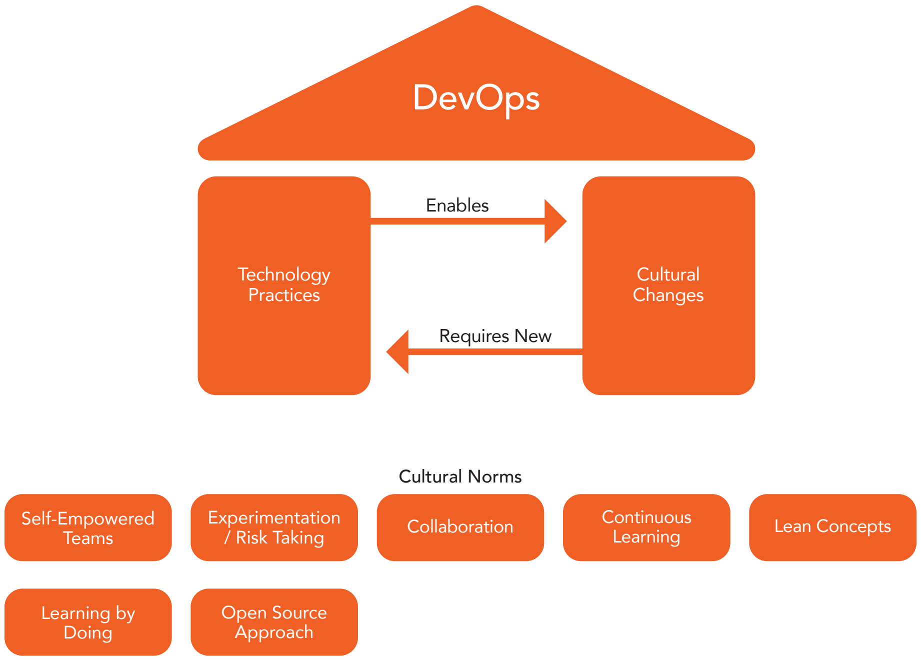
Before we get into the example case studies, we'll ground you in three of the key elements of modern technology practices as exemplified by all successful teams:

- cultural norms to support the transformation
- modern technology practices
- architectural approaches



CULTURAL NORMS

While this document is not meant to delve too deeply into cultural norms, it is clear that the success of any transformation requires changing elements of culture in areas like trust, collaboration, experimentation, and risk-taking to be successful. And technical practices can enable cultural change. For example, teams can only be truly self-empowered when they have the ability to obtain the technology resources they need—ideally, self-service. In this section we will explore the cultural elements we think are important in the context of implementing modern technology practices.



Cross-Skilled Self-Empowered Teams

Teams need to be expanded to include true product owners, architects, security engineers, integrators, testers, and infrastructure engineers. Jason Cox (Disney) has demonstrated how Disney's transformation was aided by embedding engineers with teams to help them become "builders." These teams don't throw the work over the wall to Operations, but are involved in operational aspects of support and feedback on the products that they have built and tested. The culture needs to be changed from one that focuses on delivering interim artifacts to one where teams deliver capabilities. Everyone should understand their role in delivering value to the customer.

Experimentation/Risk-Taking

A key element of DevOps transformation is the ability to test new capabilities or the business hypothesis,[FN1] and get quick feedback on the value that it has provided to end users. Technology practices support risk-taking by providing safety nets for when things go wrong. In addition, a cultural norm of supporting and encouraging experimentation and risk-taking is important in supporting the adoption of new technical practices. For example, a team unfamiliar with automated testing won't get the automation correct on their first attempt. Leadership should focus on what was learned by early attempts to improve, rather than focusing on what went wrong with the adoption of new technologies. Businesses that can "turn the crank" and deliver capabilities and get feedback more quickly than their competitors will be more successful.

Collaboration

Collaboration and trust are important elements of cross-skilled teams. For example, utilizing Operations deputies who have earned trust and who can be granted additional access rights to tools and environments reduces the need to interact with Ops, thus speeding up team processes and saving both time and money. In addition, having the team expand into the business area[FN2] can help them better understand the business value of their products and aid them in designing the experiments discussed above. Capabilities such as ChatOps are also an enabler in this area.

Continuous Learning

Investment in learning is essential to driving continuous improvement. This can take many forms. Pairing and creating cross-teams and doing rotations are ways to achieve this while delivering capabilities. But leaders in this area also set aside time for associates to invest in their development. This includes peer-based learning, teaching katas, internal conferences, and collaboration on continuous improvement (e.g., Google's learning culture).

Application of Lean Concepts

In order to apply automation, it is first necessary to create patterns or repeatable practices. Patterns require the reduction of variance in work practices. For example, architecture patterns that enable infrastructure automation, such as constraining versions of operating systems and configurations that can be selected for your application environments. Another example in the integration area is selecting a single pattern for how to automatically bring work into your backlog.

Some Lean concepts include:

- utilizing value stream analysis to reduce waste
- moving to smaller batch sizes (minimum viable product), which contributes to reducing lead times
- utilizing the concept of sustainable pace to reduce team burnout
- eliminating the hero-mentality, which is essential to scaling practices (some teams go as far as measuring and monitoring "hero work" to minimize this bad practice)
- defining what "success" looks like and what "done" is, which (1) reduces waste from developers doing too much and (2) ensures the capability developed meets the acceptance criteria
- utilizing model lines (see next paragraph)

A model line is one in which you experiment with a new practice. Using model lines enables you to start innovating in a few areas and then expand once the model has been developed. This model can then be scaled and sustained. For example, modeling a practice such as release readiness would involve performing a tool/technology trade study, implementing the release readiness in a model line area/application, looking at the results, and iterating. The output of the model would be a set of reusable practices and processes, as well as the approved tools and technologies needed to implement the practice.[FN3]

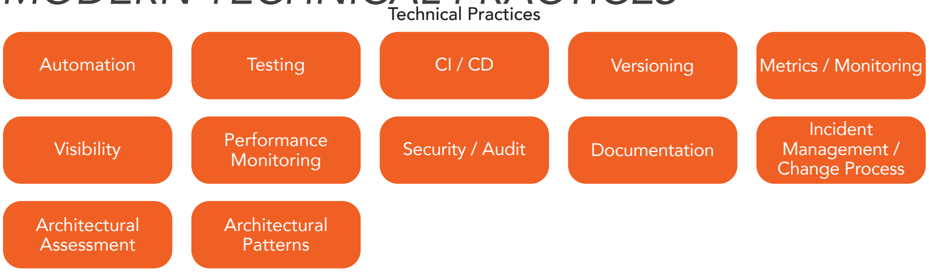
Learning by Doing

Rather than waiting for change to drive action, it's more effective to act to drive change. The 70:20:10 Model for Learning and Development is a commonly used formula within the training profession to describe the optimal sources of learning by successful managers. It holds that individuals obtain 70% of their knowledge from job-related experiences, 20% from interactions with others, and 10% from formal educational events. The model was created in the 1980s by Morgan McCall, Michael M. Lombardo, and Robert A. Eichinger, three researchers and authors working with the Center for Creative Leadership, a nonprofit educational institution in Greensboro, North Carolina, to research the key developmental experiences of successful managers.[FN4]

Open Source Approach

Open source is a model where collaborative development can come from multiple independent sources. This generates an increasingly diverse scope of design perspective that exceeds what any one company is capable of developing and sustaining long term. Another benefit is that security engineers and developers in the open source community regularly search for vulnerabilities and address them. Companies that contribute to open source not only aid the community by sharing their work but also benefit as others improve on what they originally created. This process, again, can reduce development cycles through collaboration and reuse.

MODERN TECHNICAL PRACTICES



Automation

Identify repeated manual work and apply automation. This goes beyond test automation and should be applied to process automation (e.g., automation of change with release and deployment), configuration, and orchestration. A key concept to reduce lead times is the elimination of service-level agreements

(SLAs) for request response processes and turning these into self-service on-demand capabilities. Beware of automating just the easy or low-hanging fruit, since that leaves the truly hard stuff behind to do manually[FN5].

Automated Testing

This needs to be treated like a software development discipline in terms of defining test architects, framework, applying configuration management, and design/reuse principles. Key testing components include test-driven development (TDD), where failing tests are written first and then just enough code is developed to make the tests pass, and acceptance test-driven development, which is important for each feature and is where everyone agrees on an acceptance criteria and develops an automated test in the same iteration that the story is being developed. Other important components include test data management; infrastructure and deployment test automation; code coverage measurement;code security and quality analysis (static and dynamic) automation; and test results repository management.

Continuous Integration (CI)

CI is a development practice that requires developers to integrate code into a shared repository several times a day. Each check-in is then verified by an automated build and tests which allow teams to detect problems early.

Continuous Delivery/Deployment (CD)

Continuous delivery is the capability to automate the delivery pipeline activities to accelerate the delivery of code. Continuous delivery makes the code available to be pulled into production when the customer is ready to accept it, while continuous deployment pushes the code into production when it has been validated as ready.

Methods which can improve the ability to implement CD include:

- canary deployments
- zero-downtime deployments

- automation of the change process
- feature flags and dark launches
- continuous flow

Canary deployments are the ability to deploy to a subset of users to determine if the business capability is having the desired result and then either rolling forward or rolling back depending on the outcome.

Zero-downtime deployment is the ability to spin up active-active clusters to deploy in a rolling fashion without having to make the service unavailable to the end user/customer.

Automation of the change process involves integrating CD and information technology service management, which allows for “release when ready” by generating the change request automatically when the product has achieved all the necessary quality certifications.

Feature flags and dark launches decouple application releases with the ability to turn on/off capabilities in the code base, which can then be enabled at a later time in production when the dependent services are available to be deployed into production. Feature flags and dark launches enable trunk based development, which Jez Humble and David Farley say is “the most important technical practice in the agile canon.”[FN6][FN7]

CD requires more than just the automation. It also requires thinking about and planning your work. Continuous flow, starting with release planning through deployment, is enabled by smaller batch sizes, self-service on-demand environments, and so on.

Versioning—Version control needs to be applied to everything, not just application code base. This includes test scripts and infrastructure. The ability to work from the trunk greatly improves productivity in both development and testing. Versioning is also a great enabler of safe experimentation in that you have two (or more) versions of a deployed artifact running side by side.

Metrics and Monitoring

Metrics key to DevOps are things like deployment frequency, lead time, change failure rate, and mean time to recover. Lead time measures start when the customer has an idea for how to add business value, so any analysis of the delivery value stream via value stream mapping must start well before the Agile teams start to develop and commit code.[FN8]

Ensure the hero metrics are clearly defined. For example, one metric could be the percentage of defects assigned to each person on the team. So, if Brent has 50% of the total defects of the team assigned to him, there's likely an issue. Another example would be metrics that show bottlenecks depending on the same resource, so you might measure work in process as related to a single person (e.g., the Kanban board reflects too many cards associated with Brent).

Many organizations realize that monitoring is important in operations, but business value monitoring is also important in determining if, in fact, the release (experiment) has delivered business value. This cycle through the delivery chain of plan, develop, build, configure, and deploy (sometimes called plan-do-check-act),[FN9] needs to be conducted and monitored in almost real time in order to allow the business/IT team to adapt based on the results. (See the canary deployment section above).

Visibility

Accelerated delivery is enhanced by visibility across the value stream. This takes concepts, such as visual system management, that have been applied to Agile teams across the entire lifecycle. The goal is transparency through reporting/dashboards and production telemetry. There are many ways to expose the status of work, depending on the tools in your pipeline, but the general idea is to automate the collection and visualization of all the steps in the lifecycle and display the dashboard on large wall monitors in the team room, including:

- metrics/measures (including hero metrics)
- feature availability (from customer concept)
- development status
- build status
- security scans
- static analysis
- tests status
- deployment status

Performance Monitoring

Performance testing should be done early in the release cycle to ensure that nonfunctional requirements are being met as the release is being designed and developed and are not being left to be validated in the last iteration. Stories which have the most impact on performance should be developed first, and automated regression tests should be developed and run to ensure nothing impacting performance is being introduced into the code base. It enables you to conduct functional monitoring, allows you to plan how service will be monitored at design time, and lets you begin monitoring in pre-production environments.

Documentation

Documentation takes many forms and should be located nearest to where it is used, not centralized in a locked cabinet that nobody can access. Design documentation includes requirements, functional specifications, and technical specifications for systems, application program interfaces (APIs), architecture, design, and tests. Code should include documentation that explains what is not obvious from reading the code. Changes should be traceable, with source code repositories including comments for each change, documenting why and who made the change. In addition, style guides should exist for each language, file type, or document type. Doing this helps new people get started, and it brings consistency, which improves readability.

Single Source Code Repository

A single source code repository with version control allows for sharing of code across the enterprise community. This enables teams to collaborate, which ultimately improves code quality, drives reuse, and can accelerate delivery. Instead of forcing teams to wait for the code "owner" to make needed changes, they can be made by the areas that need and have the capacity to make them. These changes can then be reviewed and approved (e.g., via pull requests) before they are accepted into the code base.

Incident Management/Problem/Change Process

Integrating processes and their associated tools, such as change/release/deployment, can greatly improve productivity and reduce lead times. The ability to automate change requests when a release has been certified to be ready is an example of this. Other examples include automatically updating configuration management database based on configurable items included in a deployment, automating the orchestration of incidents into defects and work items for Development teams, and automating intake of all development work into team backlogs.

It is also critical to reduce or remove change management formalities. Each change board in your process slows deployment of a new feature considerably. There is a dependency between removing the use of change boards and automating testing, security scans, and so on. Those automated tests give you confidence that the change will be successfully deployed, which many DevOps practitioners would argue is more effective than change review boards. A metric for the number of successful changes gives you visibility into how well your automated tests are performing.

Security Practices
Code audits and security testing (penetration tests) enable you to deploy security patches in a timely manner, including vendor patches for OS and third-party applications. They also allow you to deploy new releases of locally developed applications when security issues are resolved. Until security fixes can be deployed, code audits and security testing give you the ability to mitigate security issues by dynamically disabling features or deploying temporary firewall rules (packet filtering).

Security static code analysis capabilities can also be built into the CI process, along with other code quality analysis, to flag any critical violations that should be treated as breaking the build and need to be fixed accordingly. This ensures that the introduction of any vulnerability is immediately addressed at the source.

Architectural Assessment
The goal of an assessment is to allow you to:

- measure how monolithic you are
- measure the level of technical debt
- prioritize resolving technical debt
- measure code complexity[FN10]

Architectural Patterns
There are several architectural patterns shown below that facilitate the goals of DevOps.

- **Microservices**—This is a method of developing a software application as a suite of independently deployable, small, modular services in which each service runs a unique process and communicates through a well-defined, lightweight mechanism to serve a business goal.

- **APIs and Service Discovery**—An API is a set of subroutine definitions, protocols, and tools for building application software that define methods of communication between various software components. APIs facilitate reuse of common business logic across mobile and other front end systems of engagement. In the DevOps paradigm, they can also enable individual feature delivery.

  Service discovery is a way for services and microservices to remain stateless and discover other services that they can interact with. But, it can be useful even with monolithic systems. By moving configuration data (IP addresses, host names, ports, etc.) into a repository service, discovery allows you to build machine images that can be changed and redeployed without disrupting the system. It can be a good first step for moving to the cloud.[FN11]

- **Scalability/Auto Scaling/Horizontal Scaling**—Horizontal scalability is the ability to increase capacity by connecting multiple hardware or software entities so that they work as a single logical unit. Auto scaling helps you maintain application availability by scaling your capacity up or down automatically. A common DevOps practice is to allow the number of instances to be automatically increased during demand spikes to maintain performance and decreased during lulls to reduce costs. Capabilities like this allow for active configurations where code can be deployed without downtime (zero downtime deployments). This eliminates or minimizes the need for formal change windows and allows for a "release when ready" technique, since the end customer is not impacted. The ability to do this may require architectural refactoring.

- **Redundancy and Disaster Recovery (DR)**—Internal redundancy provides the capability to survive individual hardware failures (e.g., RAID and load balance services). If the main facility is down or destroyed, DR sites have the ability to run the service utilizing data backups, which are tested periodically.

- **Twelve-Factor App**—The twelve-factor app is a methodology for building software-as-a-service apps. This enables the building of applications in a way that can be run in a cloud environment. Applications built this way can scale and deploy rapidly, allowing their Development teams to add new features and react quickly to market changes.[FN12]

Putting It All Together

Embracing modern technology practices and moving to a high-performance culture takes into account many of the elements discussed above. Shifting from large "big bang" releases to more frequent, smaller releases has been shown to be a key enabler for high performing organizations. Reducing dependencies, which create friction, is a key to accelerating delivery.[FN13]

Agile teams are often in wait states: waiting for work, waiting for environments, waiting for other teams to deliver a service or capability. By providing continuous flow and smaller batch sizes, the Agile team can continue to be productive. Likewise, having an architecture that can translate the smaller batch size into decoupled, smaller changes in the system allows the Agile team to more quickly move through design, development, and automated testing and reduces the need for larger and more complex datasets in downstream environments. Self-service, on-demand environments (enabled by utilizing standard architecture patterns) ensure environments are available for security and performance testing as the product is being built. And integrating security, audit, and monitoring capabilities into the pipeline and demonstrating these in earlier environments again assures that any feedback is amplified into what changes may need to be made.

Having an integrated delivery pipeline that provides visibility and automated deployment (including automated build, configure, and test) assures that product deployment can occur as soon as the product has been deemed ready for release. An integrated delivery pipeline also provides the safety net that in traditional organizations is provided by safety checks, change review boards, approval signatures, and many other process steps that result in a pause or wait state in the delivery of value to the customer. All of these advantages of integrated delivery pipelines contribute to a culture of self-empowered teams who are highly engaged. Taken together, these capabilities help teams realize the Three Ways discussed in The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win by Gene Kim, Kevin Behr, and George Spafford: the First Way, continuous flow (systems thinking); the Second Way, amplified feedback loops; and the Third Way, continuous experimentation and improvement.[FN14] The First Way allows for fast left-to-right work, the Second Way enables fast right-to-left feedback, and the Third Way supports a high-trust, risk-taking culture.

DEVOPS ROADMAPS

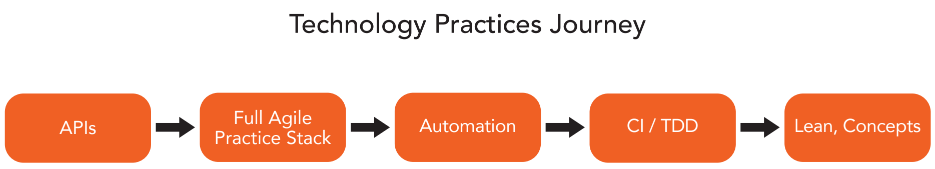
Five Case Studies that Showcase the Adoption of Modern Technology Practices

What is clear from looking at the journey of several organizations is that there is no one specific playbook to adopting modern technology practices—there are many paths to success. When an organization comes to the realization that it needs to make significant changes to its technology practices, the initial work can be difficult, and teams often feel like they are working against the way things have been done for years. Technology practices become so ingrained that even the idea of changing things creates anxiety and pushback. People resist change.

But as the writers of this paper know, change is not only possible, it can mean interesting and innovative things for the organization. Each of these five case studies showcases modern technology practices and offers a unique story detailing how the organization set about adopting them.

CASE STUDY 1

Retail DevOps: *Rebuilding an Engineering Culture*



This study focuses on the different phases of driving DevOps and rebuilding the engineering culture at a retailer. The retailer required a lot of technology and thousands of applications to support its business, and it had begun changing from a large legacy enterprise delivery machine to a more modern, nimble Agile technology organization. The organization had lost sight of the importance of engineering a while ago, and there was a culture of stopping changes whenever something broke. Because of this culture, IT would freeze production whenever problems arose, which in turn resulted in them needing to push changes in big batches. The IT organization was too complex, with silos inside of silos. Server provisions took many teams, and there was no end-to-end accountability. The system was way too complex, and over the years they had built up a large amount of technology debt. It is here that, driven by a commitment to customers and engineers, the organization's DevOps journey and the rebuilding of their engineering culture began.

There are four stages in this journey: enabling and unleashing change agents, cultivating and growing a grassroots movement, getting top-down alignment, and figuring out how to scale across the enterprise.

A few years ago, IT leaders realized that the organization needed to step up its game in the multi-channel guest experience. That was a complicated and difficult thing to do. Core data was locked in legacy systems, and it would take three to six months to develop the integrations needed to get at core data. There were multiple sources of truth. The dotcom was different from in-store systems. When they finally did get at the data, there were months of manual testing. In addition, the retailer's technology organization was very project focused and complicated. It was heavy on the project management side, and there were 20 to 30 different teams, each with conflicting priorities. Development was an exercise in waiting in queues rather than delivering results. They needed to build APIs and expose core and essential retail data, and they wanted to scale APIs and deliver new capabilities in days, not months.

It was then that IT leadership started talking about using Agile, not waterfall, systems. Team members, not contractors, began doing the engineering work, and team leaders insisted on full staff ownership, so that queuing and waiting were decreased, enabling them to get control of the ecosystem. They brought in more modern tools. Then, a couple of years in, they added technologies such as Cassandra and Kafka to give them scale and allow them to keep up with volume. The retailer's API platform was scaled as the speed of new customer experiences grew. As a result of these changes, they now do 80 deployments per week. The net present value on investment is amazing.

Digital sales were up 42% last holiday season. At the peak of this year's holiday season, 450 stores will help fulfill orders. APIs and platforms continue to scale. The organization is now investing heavily in tech because leaders understand how essential it is to a successful business.

When the IT organization's leaders saw the successes resulting from the implementation of DevOps processes, they knew they needed to scale across the entire organization. So, they started a grassroots movement, which included beginning an internal DevOps program. Right away, over 100 people showed interest in learning more about DevOps. The organization went from nobody wanting to talk about DevOps because it seemed like only Silicon Valley web companies were doing it, to increasing numbers of people talking about the amazing successes resulting from it. They also started getting engineers together to talk about infrastructure as code and began to connect to the larger tech community by bringing in external speakers.

In addition, because rebuilding the engineering culture meant rebuilding the tech brand, they had the opportunity to have fun branding around their commitment to technology.

After all this, it became obvious that senior leadership buy-in was necessary. The bottoms-up phase needed to work into top-down engagement. To achieve this, they did a lot of momentum building. DevOps and Agile became core pillars and goals, as well as part of the daily conversation, and they did town hall huddle meetings. They also aligned with Thoughtworks' CI/CD Maturity Model to baseline measure products. They set goals and assigned DevOps champions to help drive practices within their spaces and be champions within their teams. Despite these efforts, they still had hundreds in middle management who had not yet been exposed to the thinking. So, they decided to do an internal mini-DevOps enterprise summit. The retailer's management became involved in these discussions, and it energized a lot of people.

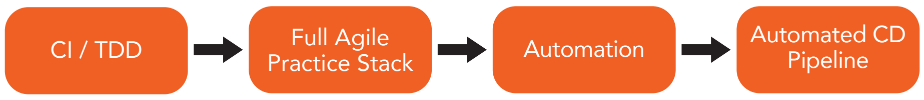
Once all of these other steps were in place, the next question was how to actually scale across a large organization. First, they focused on structural changes. They moved from a COBIT-based, highly-segmented model, to a product and service model; simplified accountabilities and established key practice areas across the organization; and shifted their delivery model from waterfall and project-based to product-based using Scrum and Kanban models. They also embarked on a tech modernization strategy. Because they had a lot of legacy and tightly-coupled integrations, they needed key strategic capabilities to move to a more modern tech architecture: API based, loosely coupled, lightweight tooling, self-service, and optimized for cloud based and CI/CD practices.

They also needed to change how people worked. They began this process by converging the Agile and DevOps movements. What was once loosely connected became more tightly connected. They then pulled in an effectiveness organization to look at how they could scale learning. Traditional training was combined with a focus on coaching and hands-on immersive experiences. They also created an internal incubator environment, like a transformation emergence center, called the Dojo, which became the place they went to practice their craft. The idea was that teams would come to the Dojo with real work, and coaches would work in an immersive environment to show them how to apply new practices. By doing this, they began creating what an engineering environment should look like.

CASE STUDY 2

Technology Changes in Government Agencies (A Compilation of Cases): *Lessons in Legacy and DevOps*

Technology Practices Journey



This story explores companies that provide IT services to large government organizations. IT for agencies like this includes enterprise applications, such as email, HR systems, finance, and literally thousands of little applications. There are 240 unique websites: 40 applications and 200 content for various constituencies. For the purposes of this compilation, we will refer to the organization in the singular even though it contains information synthesized from numerous of examples.

The IT organization was structured into nine service delivery teams, each focused on a different application (email, finance, etc.) or technology (networking, data centers, security). These divisions were arbitrary and isolated; PMs were responsible for figuring out which subset of teams were needed for each project.

The process for each project was waterfall. It took a minimum of nine months to gather requirements, design, develop, and finally deploy the result. This waterfall was enforced through gate review meetings and a strong change control process, which required prior gates to have occurred before updates could flow between phases (e.g., before code could be moved from Development to Stage or Stage to Production).

In addition to this central IT organization, many customers had shadow IT departments or staff who took care of anything that didn't require shared services provided by the nine teams. These shadow IT departments sometimes also built systems that were simple enough that IT could be convinced to host and manage them.

Historically, the central IT team was highly successful because they offered reliability. The teams were fairly good at delivering results on time, though with nine months being the minimum for a project, reliability did not mean speed or efficiency.

The process for adopting DevOps practices in this governmental organization took many tries and was completed in different departments with varying levels of success. This case study illustrates how people's resistance to change and the tension between legacy and DevOps are two key factors to consider and work with when planning a move to adopting DevOps principles.

The Cloud Brings Change

The appearance of Amazon Web Services (AWS) meant that the IT division had competition for the first time in their history. All the teams considered AWS to be a threat. It was only a direct threat to the Datacenter team, however. Even if clients worked around the IT department by using AWS, they still required the services of the Security, Network, Development, and other teams.

First Attempt at DevOps Adoption

A project manager desired to deliver services faster by adopting DevOps techniques, such as CI/CD, and new technologies, such as AWS, which meant getting all the various teams on board to launch their first project on AWS.

The front end developers and middle-tier support team were enthusiastic about being involved. They didn't care where their code ran, and AWS was a modern technology that looked good on their resumes. The Security team was neutral. The leader of the Datacenter team actively tried to sabotage the effort. He was the acting lead of the Network team, which gave him additional leverage as AWS required that team's VPN and network services support to extend the domain to the new cloud environment. The Datacenter lead was also the owner/co-chair of the change control board and, prior to the introduction of AWS, had full authority to accept or deny any system or project being installed in the datacenter.

The sabotage and other political moves created an "us versus them" situation. People were picking sides, often based on who they thought would "win" rather than what was best for the organization.

There were pockets of enthusiastic supporters, even on the teams managed by the saboteurs. They were reading books and doing unofficial projects to learn new technologies. Unfortunately, these individuals struggled to work in the chaotic environment being created by the transformation program.

The PM ultimately realized she was fighting against insurmountable headwinds and that this was not the way to win the hearts and minds of those involved. A new strategy was needed.

Second Attempt

The second attempt at adopting DevOps principles involved a strategy of consolidating passion, skills, and willingness into one team. The PM hoped to leverage the success of this team to encourage the spread of new ideas and ways of working. The people who had been supportive during the first attempt were cherry-picked and brought together into a new team. This was done under the cover of reorganizing "by technical skill," so that the effort would fly under the radar of those that might attempt to sabotage it.

Their first project was estimated to take eight months. However, the CIO needed it done in four.

The team lead the migration of their portfolio of apps. They used tools that were not approved by the central IT organization, but it was easier to beg forgiveness than get permission. This included using GitHub Enterprise, an open source Agile backlog tool, and AWS.

They were radically successful. After a six-month lead-in period, the team started doing weekly incremental deployments to AWS. They were able to launch in four months and stayed within budget. They also experienced a high profile success during the project when there was a news event that created the need for an ad-hoc website update. The organization wanted to be able to quickly offer a simulcast feed of a news conference on their website. The DevOps team was able to rearrange their priorities and respond to this feature request in under a week and, once tested, deploy the update in 24 hours to the website using their automation tools. They were also able to quickly engineer the website to scale elastically to support the high-traffic, high-profile announcement.

This convinced a lot of people that this was the better way to work. Once they realized that AWS wasn't taking their jobs away (for example, the Security team's role wasn't diminished no matter where code ran), more people got on board.

Short-Lived Success

Sadly, the success created an organizational cleaving. It was the successful DevOps people versus the legacy people. The legacy people would say, "That's how *they* work; that wouldn't work for us." This became alienating. They denigrated anything that sounded like DevOps, such as weekly deployments, IaC, and cloud platforms.

In support of their position, the legacy people pointed out that they had never had a failed launch. Their waterfall-style projects took years, but their launches were always a "success." However, this was because any possible failure was postponed rather than deployed or rolled back before go-live. Thus, no failed site ever "deployed" under the old model. In contrast, the DevOps people were doing weekly launches. Occasionally a launch would fail, but given the blue/green deployment model, a failed launch could be reset quickly to the old site within seconds. Nonetheless, the legacy people would point to the rollbacks as proof that DevOps was chaotic and immature. They used the idea that they had never had a failed launch to justify why people used their services. The problem was, they had plenty of failed launch attempts, but through individual heroics and many late nights, the service would eventually be up and running. They considered these heroics to be a normal part of launches and didn't count them as failures.

The DevOps team understood that it was important to lean forward and push things. It wouldn't hurt if there were occasional mistakes. In fact, it would help because they would then be able to learn from them. They could go further using this new way, even with occasional rollbacks.

The following examples illustrate this.

The legacy team was involved in launching a new mobile app. Press releases had already gone out with the launch date; therefore, there could be no delays. Production got the software delivery two days before the launch, leaving no time for load testing. Response was larger than expected, and the system became overloaded, making it unusable. Media criticism was huge. It was a black-eye for the client. However, the legacy team considered this a successful launch because it was on time and met the requirements as specified. Additionally, the system was technically "up" the entire time, even though end users could not access it. The legacy team measured uptime based on their SLA, which was entirely based on total server uptime and not end-user availability.

Around the same time, the DevOps team launched a new website that wasn't expected to receive a large amount of traffic. However, the PR department learned of news that was going to go public in a few hours, which, based on a similar situation, was going to increase the traffic by 70x. The news hit the media about an hour into the team's efforts to scale the site, but the website was never slow enough to be newsworthy. The media's coverage of the event focused on the event.

The legacy team pointed at the DevOps team scaling as a failure because a correction had to be made after launch. What the legacy team didn't understand was that value added to the end user is more important than the team meeting their milestones.

The legacy team's process for handling such a situation would have been to convene a change control board and find hardware that could be scavenged from other projects and repurposed to add capacity. They would have reduced features by replacing the main page with a static HTML page, substituting smaller or lower-resolution images, and so on. The technical changes wouldn't have started for at least a day. In today's 24-hour news cycle, the solution wouldn't have been implemented until well after the media had moved on to the next big story.

On the contrary, the DevOps team didn't reduce functionality, they added it. They did an early launch of video and other content that they found related to the event. These were features they hadn't planned on launching yet, but due to their CI/CD methodology, they were confident in delivering them early.

Success Doesn't Always Last

Overall, the DevOps transformation at this governmental IT organization was wildly successful, but the organizational antibodies were fierce. While the website team has gone DevOps, the rest have dug in and gone full waterfall. The legacy group now only gets customers that are forced to use them due to externalities. The legacy group has shrunk, closed ranks, and lost budget.

Sadly, the CIO that had been supporting the DevOps efforts retired, and the leader of the legacy team was selected to replace him. This was a major step backwards.

To save the progress made so far, the PM of the DevOps group moved quickly, arranging things so that the team reported directly to the customer group they were serving. As a result, they were no longer under the (new) CIO's control.

Hindsight Is How We Learn

What could this organization have done differently?

- Transforming a new team every few months rather than waiting for the DevOps experiment to be complete would have helped. This would have built a community of expertise that would have snowballed into a larger, more sustainable success.

- Isolating the DevOps team let the other teams off the hook about following the better practices. While it was good that the DevOps team was successful, it created an "us versus them" situation. It would have been better to make the others be "fast followers," partnering them with stakeholders faster by including them in daily stand-ups, involving them in Scrum or Kanban activities, and so on.

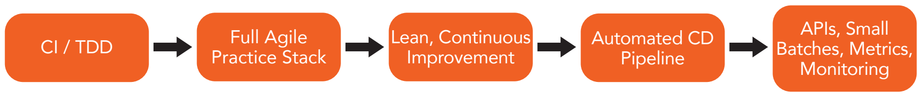
- They could have focused on rewarding not just the first team, but all teams. The first team got the top cover and did it out of love. The other teams took more personal risk because they weren't "true believers." People can relate better to non-trailblazers winning the award.

- Finally, they could have started all new projects and organizations with DevOps practices. New-hires are more open to the practices in general, and there is no transformation needed if they arrive and find that DevOps exists already.

CASE STUDY 3

Agile Implementation in a Large, Regulated Industry: *DevOps and Accelerating Delivery*

Technology Practices Journey


CI / TDD
Full Agile Practice Stack
Lean, Continuous Improvement
Automated CD Pipeline
APIs, Small Batches, Metrics, Monitoring

At this organization, the starting environment consisted of approximately eight thousand IT staff supporting two thousand applications for twenty business-facing IT areas. The delivery model being used was waterfall, with some established CI and TDD practices in advanced areas. There was large variation in how development was done across the enterprise, resulting in inconsistency in quality and challenges in delivery.

As a result of this, the board was considering more outsourcing. A senior vice president in an area that was doing some Agile work was given a year to run an experiment to demonstrate results that these Agile practices, which were showing promise, could be established and scaled to drive in-sourcing of work in a centralized development organization.

CI/TDD and Full Agile Stack Adoption

The journey started by establishing consistent Agile management practices, which were applied across all technologies (e.g., standups, visual system management, iterations, show and tells, iteration planning, etc.). Engineering practices were adapted as applicable based on technology (version control for all source code, CI for Java/.Net, TDD, automated accepted testing for all stories). Continuous peer-based learning and improvement processes were put into place to drive sustainability and scaling across the enterprise.

The results within one year demonstrated that the experiment was a success. Significant results were achieved in quality (80% reduction in critical defects), productivity (82% of Agile teams were measured to be in the top quartile industry side), availability (70% increase), and on-time delivery (90%, up from 60%). As a result of this, the amount of work being done by Agile teams increased to over 70% of the total development work being done in IT (growing from three Agile teams to over two hundred teams) over five years.

Adopting Lean Engineering Practices (DevOps)

The result of the Agile implementation was that work moved very efficiently and quickly in the middle of the development cycle, but there were wait states at the beginning and the end of the cycle, creating a "water-Scrum-fall" effect.

There were wait states at the beginning of the life cycle, waiting for work to flow into the backlog of Agile teams, and wait states downstream waiting for dependent work to be done by teams and for environments. Sixty percent of the time spent on an initiative was prior to a story card getting into the backlog of an Agile team. Once work left the final iteration, there were high ceremony, manual practices leading to increased lead time for deployments. This was due to disparate release and deployment practices, technologies, and dependences due to large release batch sizes.

At this point, the focus moved to reducing variation in processes such as work intake, release, deployment, and change across the delivery pipeline, and applying Lean practices (A3, value stream analysis) to identify areas for improvements to reduce end-to-end lead time. The results of this was the construction of an integrated CD foundational pipeline to provide visibility and an accelerated delivery capability. Other initiatives to reduce wait times included infrastructure automation and more test automation, including service virtualization and test data management.

Small Batch Sizes, APIs, Microservices, Monitoring, and Metrics

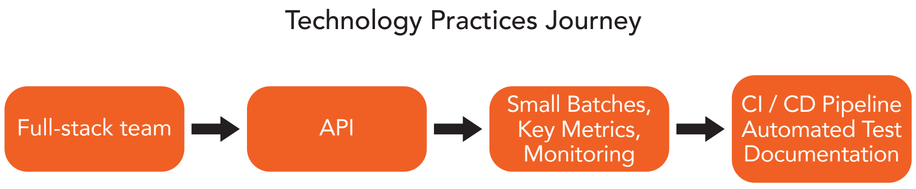
By itself, the creation of a CD foundation with automated infrastructure would not significantly reduce lead times to accelerate delivery without reducing dependencies, which led to the wait states that the Agile teams encountered. As such, the next step was to focus on small batch sizes and modern web architecture practices (APIs, microservices). Additional methods to remove dependencies included the implementation of dark launching, feature flags, and canary testing to move from releasing based on a schedule to releasing based on readiness. More emphasis was placed on creating infrastructure as code, including containerization.

In order to improve lead times, it is first necessary to measure the entire end-to-end lead time. Having an integrated pipeline and workflow allowed the implementation of metrics to baseline lead time and measured improvements, which also shed light on where wait states existed within the delivery value stream.

As delivery becomes accelerated, it also becomes more critical to get real time monitoring and feedback for both operational performance and customer feedback to determine if the business case "hypothesis" (including multivariate A/B testing) is in fact driving more customer value.

CASE STUDY 4

DevOps and Moving to Agile at a Large Consumer Website: *Getting Faster Answers at Yahoo Answers*


Technology Practices Journey
Full-stack team
API
Small Batches, Key Metrics, Monitoring
CI / CD Pipeline Automated Test Documentation

Yahoo Answers was created in 2006 as a place to share knowledge on the web and bring more knowledge to the Internet. It's basically a big game: people compete to answer visitor questions, and the most approved questions help them work their way up to higher levels.

In 2009, their growth was flat, at around 140 million monthly visits. In addition, they had declining user engagement, flat revenue, and a contentious team of employees. They used waterfall development with four to six week cycles because there were quality issues in Operations and Development, and people were obstructing releases. Fourteen months later, Yahoo Answers was getting over 240 million monthly visits and over 20 million people answering questions.[FN15] It was also available globally in twenty languages. It was a very large-scale property and a significant part of Yahoo traffic. They were able to grow traffic by 72%, user engagement was up 3x, and revenue was up 2x. They had daily releases and better site performance, and they had moved from a team of employees to a kick-ass team of owners.

So, what is the backstory of the transition? Yahoo Answers had an amazing team of four to five leaders across Engineering, Product, Design, and Operations who helped transform the business. Everyone sat down together and came to the conclusion that they could no longer run the business like this. They all developed a plan. The first step was to get everybody closer together. When Jim Stoneham arrived in 2009 as VP of Communities, they had people in London and France, while Jim was in the United States. The odd triangle created slow movement and arguments, and people were constantly missing each other due to being in different locations and different time zones. Because there was not a lot of technology to help facilitate remote teams in 2009 (Slack, etc.), it was essential to come together geographically. To that end, they consolidated the functional teams in London and France by bringing them all to London.

Once they were all in the same place, they decided it was necessary to focus on a few key metrics. Their old dashboard tracked every single metric, meaning, of course, that nobody paid attention to anything. So, they simplified. They asked customers what mattered. The responses they received revealed that customers were primarily concerned with time to first answer, time to best answer, upvotes per answer, answers/week/person, second search rate, and trending down (negatively correlated). Revenue was not a key metric was pageviews. Those would follow if the other metrics were doing well.

Then they knew they needed to architect to enable velocity of deploying and independence. In 2008, Answers was built on top of an Oracle RAC database and five-year-old legacy code. They re-architected in place. They couldn't shut down the business, and they didn't want to build a new system next door and migrate, so they built a MySQL-based read cache to take stress off RAC systems in back, built data access layer for read/write to core database, and refactored one page at a time. There was a lot of interacting going on. It was essential to start with less-used pages. The last page refactored was the actual question/answer page, which carried most of the traffic. They broke down a monolithic app into a service oriented architecture, which paid dividends with their Agile process, all while serving billions of pages each month. Altogether, the transition took four months, with sixty days of planning before they began actually writing code.

The next step was to reduce the size to small units of work focused on a key metric, which would have the effect of making the unit of work smaller and smaller. Yahoo had been working waterfall at four to six week releases. The Operations team at the time viewed Agile as a whole bunch of people throwing stuff at them. They were very resistant to trying it out. To make matters more difficult, the Operations team was in a different functional organization thanthe business leadership, which meant that leadership had no actual organizational power over Operations.

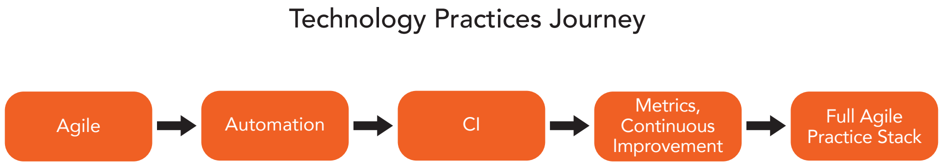
To overcome this, they got everyone into a room and came up with a process that would work for all of them all as a team and would quickly drive experiments so that people would own the quality. This involved all stakeholders. Their new product process included weekly sprints; daily deploys (except Fridays); reviewing metrics daily or more, which was key to moving from a team of employees to a team of owners and helped create a cultural transformation; and weekly iteration planning. This weekly planning kept up a cadence of looking forward and backward by a week. The new process also included monthly business reviews (all hands) during which they would take five core metrics and revenue and look at the information together. IT was made up of extended Operations people and community managers, which turned out to be a group of eighty people or so. It took them 60 to 90 days to get the process working well.

Another big step in this type of transformation is to allow people to screw up, to roll forward or back. If you want to take risks and hit for the fences, you have to give people the permission to make mistakes, and you need to make it really easy to recover from those mistakes. In 2009, rollbacks looked awful. It was like putting out a major fire. In 2010, they implemented changes that allowed rollbacks to happen really quickly. By using Hudson, they could roll back a set of scripts. Ninety percent of the time they would roll forward because it was easy to deploy from trunk. It was essential to give people an environment where they could roll back or forward quickly. They also rewarded people for taking risks, failing, realizing failure, and killing things when they knew those things had failed. This all encouraged experimentation.

Of course, no plan is exact, and anytime you lay one out, it never ends up being as simple as a five point plan. Other things always come into play. For the Yahoo Answers team, some of those things were coaching managers on "soft skills," exiting people who weren't on board, utilizing an A/B testing framework, reporting a few more metrics upward, tooling for change monitoring.

CASE STUDY 5

Real-Time Embedded Software: *DevOps Practices for an Unhappy Customer*


Technology Practices Journey
Agile
Automation
CI
Metrics, Continuous Improvement
Full Agile Practice Stack

As with many DevOps journeys and transformations, this one began with the customer. The customer was providing constantly-changing requirements, increasing demands, and wanting things done faster and faster, which translated to failed waterfall schedules and budgets, as well as increased pressure on Development and Operations to do exceedingly impossible things.

For a little background, it is important to understand the starting environment and the challenges faced by Development and Test that led to the decision to begin to overhaul standard practices. The organization was steeped in the traditional waterfall, long-cycle system approach, with risk averse management, and it had begun to cause the usual problems: the customer not really knowing what they wanted or needed at the beginning of the process; difficulty turning customer ideas into working products; and Development and Operations being separate and often working at cross purposes. In fact, Development itself was made up of separate Requirements, Software, Integration, and Test teams.

As a result of all of these difficulties, the project manager had been replaced four times due to failing to meet schedule and budgets. The customer was unhappy because the organization often wasn't able to deliver what the customer needed, and even when it was delivered, it wasn't on time. The Development and Test teams were exceedingly frustrated because no matter how many hours they worked, they had zero automated tests and no automated builds, and there never seemed to be a path to success.

Something had to change. As in many cases where waterfall methods are helping to create customer and employee dissatisfaction, the organization began to consider DevOps practices. They decided to begin the move to an Agile approach using Scrum. As they did, they started to build trust with the customer, and customer communication increased exponentially.

The organization's Agile approach using Scrum helped to overhaul customer relationships and created a dynamic working environment where the customer felt supported and understood the processes. The Software team started with monthly roadmap reports to the customer to keep them apprised of business implications and performance indicators for any given project. In addition, the customer began participating in sprint reviews, with the option to get engineering releases at the end of every sprint. These steps helped the customer understand the value in the Agile process, and it meant a huge shift in how the customer worked with Development. The customer began allowing the Development team to help prioritize requirements, and the Development team started participating in user groups. All work was estimated in story points, and the paperwork between the customer and the Development team was significantly reduced.

As all of these changes played out, there was a lot of pushback from the teams developing requirements and testing the system. After two years, a new Requirements team manager was hired who was open to change and really listened to arguments for becoming "one team." Almost three years in, they managed to break down the siloes and become a cross-functional team.

They also began automated testing, which was a game-changer. Sixty-five percent of requirements are now covered by automated tests using Ranorex. The team moved to a policy that required all new capabilities and all touched code to have automated unit tests. As a result, they achieved 50% code coverage with automated unit tests in a Visual Studio unit test framework. All of this forced the team to think more about the unit tests. Even though it was more time-consuming to create the automated unit tests, since they were part of CI, the return on investment made it worth the time. And now, three years into the process transformation, the team is getting to the point of wanting to review tests for value added.

Initially, in the waterfall environment, builds were completely manual, complex, and error-prone. They took four to five hours, mostly in build set-up time. Now automated builds mean that the process takes about five minutes. In the Unix environment, they are using Bash shell scripts for building the system and running all unit tests, and they run at least nightly if a change is made to the code. In the Windows environment, they are now using Team Foundation Server build capability. Here is a basic breakdown of the overall picture:

Starting environment:

- frustrated customer
- Windows C# and Unix C++ Embedded Code
- 2K organization
- 40 people in change org; co-located
- Requirements/Test were in a different org, reported to different leader, were not on-board

DevOps practices—move to Agile/automated build and test:

- focus on engaging the customer
- focus on testing earlier and more often
- measure effect and value of automated test

- measure effect and value of customer engagement, short cycles, reducing work in process

Break down the silos:

- work more closely with the customer
- break down the barriers between the Requirements engineers, software developers, and testers—cross-functional teams
- entire team focused on delivering capability to the customer

Outcomes:

- The customer is now much happier and is an advocate for the team.
- The customer gets to change needs and priorities without push-back from the team.
- The team works a reasonable schedule with significantly reduced overtime—a much happier team means higher retention.
- The team consistently over-delivers (more than planned) on schedule.
- Team members have become owners of the process.
- The team lost some team members who just weren't willing to change or couldn't get used to the new process.
- It took 12 to 15 weeks for the team to really transition to the new processes and reach their new velocity.

SUMMARY

As you can see, leaders in DevOps transformations face a host of challenges when they decide that adopting modern technology practices is an important next step for their organization. These challenges, while initially daunting, can be faced using a whole host of tools. Furthermore, as exemplified by these case studies, the adoption and acceleration of DevOps practices can occur successfully in many different industries.

We hope you have found these examples useful and that you are able to use them as inspiration as you approach the idea of transforming your own team and processes. When faced with new innovations, ever-changing environments, and the need to make delivery of services much faster and more efficient, all while remaining competitive and successful, implementing new DevOps practices can seem a daunting endeavor. However, keep in mind that there are three elements we have seen in all our examples that create a successful environment for adopting DevOps practices: cultural norms to support the transformation, modern technology practices, and architectural approaches. As we have shown you, with these in place it is possible to adopt DevOps practices one step at a time. Before you know it, you will have created a map of your journey that you can share with others as proof that DevOps practices can mean positive business outcomes, personal successes, and fully-functional teams who believe in the work they are doing because the positive business outcomes keep piling up.

Contributors

Authors

Jim Stoneham, *VP of Product Management at New Relic*, jstoneham@ne-wrelic.com

Paula Thrasher, *Application Delivery Lead at CSRA*, pthrasher@csc.com

Terri Potts, *Technical Director at Raytheon IIS Software*, terri.potts@gmail.com

Heather Mickman, *Senior Director of Platform Engineering at Target*, heather.mickman@target.com

Carmen DeArdo, *Technology Director at Nationwide Insurance*, cdeardo@gmail.com

Thomas A. Limoncelli, *SRE Manager, StackOverflow.com and author*, tom@limoncelli.com

Other Contributors

Kate Sage, Developmental Editor, Writer