The IT Revolution

DevOps Guide

Selected Resources to Start Your Journey

IT REVOLUTION PRESS

Contents

Sponsors

Delphix
Electric Cloud
Microsoft
New Relic.
RALLY
XebiaLabs Deliver Faster

The most commonly asked question that we get at IT Revolution is "How do I get started with DevOps?"

Rather than try to answer all of these questions ourselves, we decided to gather the best resources from some of the best thinkers in the field.

Our goal for *The IT Revolution DevOps Guide: Selected Resources to Start Your Journey* is to present the most helpful materials for practitioners to learn and accelerate their own DevOps journey.

We reached out to several practitioners that we admire for their best ideas on how to get started. In addition, we assembled some of the best material from the vendor community and have highlighted those works as well. We combined these with excerpts from *The Phoenix Project*, the upcoming *DevOps Cookbook*, 2014 State of DevOps Survey of Practice, and 2014 DevOps Enterprise Summit. You'll find a collection of essays, book excerpts, videos, survey results, book reviews, and more.

We hope you enjoy this collection and find it useful, regardless of where you are on your DevOps journey.

– GENE KIM AND THE IT REVOLUTION TEAM

Starting with DevOps

Why Do DevOps?

The competitive advantage this capability creates is enormous, enabling faster feature time to market, increased customer satisfaction, market share, and employee productivity and happiness, as well as allowing organizations to win in the marketplace. Why? Because technology has become the dominant value creation process and an increasingly important (and often the primary) means of customer acquisition within most organizations.

In contrast, organizations that require weeks or months to deploy software are at a significant disadvantage in the marketplace.

COMPANY	DEPLOY FREQUENCY	DEPLOY LEAD TIME	RELIABILITY	CUSTOMER RESPONSIVENESS
Amazon	23,000/day	minutes	high	high
Google	5,500/day	minutes	high	high
Netflix	500/day	minutes	high	high
Facebook	1/day	minutes	high	high
Twitter	3/week	minutes	high	high
Typical enterprise	once every 9 months	months or quarters	low/medium	low/medium

One of the hallmarks of high performers in any field is that they always "accelerate from the rest of the herd." In other words, the best always get better.

This constant and relentless improvement in performance is happening in the DevOps space, too. In 2009, ten deploys per day was considered fast. Now that is considered merely average. In 2012, Amazon went on record stating that they were doing, on average, 23,000 deploys per day.

Business Value of Adopting DevOps Principles

In the 2013 Puppet Labs “State of DevOps Report,” we were able to benchmark 4,039 IT organizations with the goal of better understanding the health and habits of organizations at all stages of DevOps adoption.

The first surprise was how much the high-performing organizations using DevOps practices were outperforming their non-high-performing peers in agility metrics:

- 30× more frequent code deployments
- 8,000× faster code deployment lead time

And reliability metrics:

- 2× the change success rate
- 12× faster MTTR

In other words, they were more Agile. They were deploying code thirty times more frequently, and the time required to go from “code committed” to “successfully running in production” was eight thousand times faster. High performers had lead times measured in minutes or hours, while lower performers had lead times measured in weeks, months, or even quarters.

Not only were the organizations doing more work, but they had far better outcomes: when the high performers deployed changes and code, they were twice as likely to be completed successfully (i.e., without causing a production outage or service impairment), and when the change failed and resulted in an incident, the time required to resolve the incident was twelve times faster.

This study was especially exciting because it showed empirically that the core, chronic conflict can be broken: high performers are deploying features more quickly while providing world-class levels of reliability, stability, and security, enabling them to out-experiment their competitors in the marketplace. An even more astonishing fact: delivering these high levels of reliability actually *requires* that changes be made frequently!

In the 2014 study, we also found that not only did these high performers have better IT performance, they also had significantly better organizational performance as well: they were two times more likely to exceed profitability, market share, and productivity goals, and there are hints that they have significantly better performance in the capital markets, as well.

What It Feels Like to Live in a DevOps World

Imagine living in a DevOps world, where product owners, Development, QA, IT Operations, and InfoSec work together relentlessly to help each other and the overall organization win. They are enabling fast flow of planned work into production (e.g., performing tens, hundreds, or even thousands of code deploys per day), while preserving world-class stability, reliability, availability, and security.

Instead of the upstream Development groups causing chaos for those in the downstream work centers (e.g., QA, IT Operations, and InfoSec), Development is spending twenty percent of its time helping ensure that work flows smoothly through the entire value stream, speeding up automated tests, improving deployment infrastructure, and ensuring that all applications create useful production telemetry.

Why? Because everyone needs fast feedback loops to prevent problematic code from going into production and to enable code to quickly be deployed so that any production problems are quickly detected and corrected.

Everyone in the value stream shares a culture that not only values people's time and contributions but also relentlessly injects pressure into the system of work to enable organizational learning and improvement. Everyone dedicates time to putting those lessons into practice and paying down technical debt. Everyone values nonfunctional requirements (e.g., quality, scalability, manageability, security, operability) as much as features. Why? Because nonfunctional requirements are just as important in achieving business objectives, too.

We have a high-trust, collaborative culture where everyone is responsible for the quality of their work. Instead of approval and compliance processes, the hallmark of a low-trust, command-and-control management culture, we rely on peer review to ensure that everyone has confidence in the quality of their code.

Furthermore, there is a hypothesis-driven culture, requiring everyone to be a scientist, taking no assumptions for granted and doing nothing without measuring. Why? Because we know that our time is valuable. We don't spend years building features that our customers don't actually want, deploying code that doesn't work, or fixing something that isn't actually the problem. All these factors contribute to our ability to release exciting features to the marketplace that delight our customers and help our organization win.

Paradoxically, performing code deployments becomes boring and routine. Instead of being performed only at night or on weekends, full of stress and chaos, we are deploying code throughout the business day, without most people even noticing. And because code deployments happen in the middle of the afternoon instead of on weekends, for the first time in decades, IT Operations is working during normal business hours, like everyone else.

Just how did code deployment become routine? Because developers are constantly getting fast feedback on their work: when they write code, automated unit, acceptance, and integration tests are constantly being run in production-like environments, giving us continual assurance that the code and environment will operate as designed and that we are always in a deployable state. And when the code is deployed, pervasive production metrics demonstrate to everyone that it is working and the customer is getting value.

Even our highest-stakes feature releases have become routine. How? Because at product launch time, the code delivering the new functionality is already in production. Months prior to the launch, Development has been deploying code into production, invisible to the customer, but enabling the feature to be run and tested by internal staff.

At the culminating moment when the feature goes live, no new code is pushed into production. Instead, we merely change a feature toggle or configuration setting. The new feature is slowly made visible to small segments of customers and automatically rolled back if something goes wrong.

Only when we have confidence that the feature is working as designed do we expose it to the next segment of customers, rolled out in a manner that is controlled, predictable, reversible, and low stress. We repeat until everyone is using the feature.

By doing this, we not only significantly reduce deployment risk but also increase the likelihood of achieving the desired business outcomes, as well. Because we can do deployments quickly, we can do experiments in production, testing our business hypotheses for every feature we build. We can iteratively test and refine our features in production, using feedback from our customers for months, and maybe even years.

It is no wonder that we are out-experimenting our competition and winning in the marketplace.

All this is made possible by DevOps, a new way that Development, Test, and IT Operations work together, along with everyone else in the IT value stream.

DevOps Is the Manufacturing Revolution of Our Age

The principles behind DevOps work patterns are the same principles that transformed manufacturing. Instead of optimizing how raw materials are transformed into finished goods in a manufacturing plant, DevOps shows how we optimize the IT value stream, converting business needs into capabilities and services that provide value for our customers.

During the 1980s, there was a very well-known core, chronic conflict in manufacturing:

- Protect sales commitments
- Control manufacturing costs

The principles behind DevOps work patterns are the same principles that transformed manufacturing. ... DevOps shows how we optimize the IT value stream, converting business needs into capabilities and services that provide value for our customers.

In order to protect sales commitments, the product sales force wanted lots of inventory on hand, so that customers could always get products when they wanted them. However, in order to reduce costs, plant managers wanted to reduce inventory levels and work in process (WIP).

Because one can't simultaneously increase and decrease the inventory levels at the plant, sales managers and plant managers were locked in a chronic conflict.

They were able to break the conflict by adopting Lean principles, such as reducing batch sizes, reducing work in process, and shortening and amplifying feedback loops. This resulted in dramatic increases in plant productivity, product quality, and customer satisfaction.

In the 1980s, average order lead times were six weeks, with less than 70 percent of orders being shipped on time. By 2005, average product lead times had dropped to less than three weeks, with more than 95 percent of orders being shipped on time. Organizations that were not able to replicate these performance breakthroughs lost market share, if they didn't go out of business entirely

Velocity 09: John Allspaw and Paul Hammond, "10+ Deploys Pe
O'REILLY
Velocity
Web Performance and Operations Conference
0:00 / 46:22

Where It All Started:
10+ Deploys per Day: Dev and Ops Cooperation at Flickr

presentation by John Allspaw and Paul Hammond at Velocity 2009

This talk is widely credited for showing the world what DevOps could achieve, showing how one of the largest Internet sites was routinely deploying features into production at a rate scarcely imaginable for typical IT organizations who were doing quarterly or annual updates.

How Does DevOps "Work"?

from *Navigating DevOps*

New Relic.

New Relic is a software analytics company that makes sense of billions of data points about millions of applications in real time. New Relic's comprehensive SaaS-based solution provides one powerful interface for web and native mobile applications and consolidates the performance monitoring data for any chosen technology in your environment. More than 250,000 users trust New Relic to tap into the billions of real-time metrics from inside their production software.

Like all cultures, DevOps has many variations on the theme. However, most observers would agree that the following capabilities are common to virtually all DevOps cultures: collaboration, automation, continuous integration, Continuous Delivery, continuous testing, continuous monitoring, and rapid remediation.

Collaboration

Instead of pointing fingers at each other, development and IT operations work together (no, really). While the disconnect between these two groups created the impetus for its creation, DevOps extends far beyond the IT organization, because the need for collaboration extends to everyone with a stake in the delivery of software (not just between Dev and Ops, but all teams, including test, product management, and executives).

Successful DevOps requires business, development, QA, and operations organizations to coordinate and play significant roles at different phases of the application lifecycle. It may be difficult, even impossible, to eliminate silos, but collaboration is essential.

Automation

DevOps relies heavily on automation—and that means you need tools. Tools you build. Tools you buy. Open source tools. Proprietary tools. And those tools are not just scattered around the lab willy-nilly: DevOps relies on toolchains to automate large parts of the end-to-end software development and deployment process.

Caveat: because DevOps tools are so amazingly awesome, there's a tendency to see DevOps as just a collection of tools. While it's true that DevOps relies on tools, DevOps is much more than that.

Continuous Integration

You usually find continuous integration in DevOps cultures because DevOps emerged from Agile culture, and continuous integration is a fundamental tenet of the Agile approach.

Continuous integration (CI) is a software engineering practice in which isolated changes are immediately tested and reported on when they are added to a larger code base. The goal of CI is to provide rapid feedback so that if a defect is introduced into the code base, it can be identified and corrected as soon as possible. The usual rule is for each team member to submit work on a daily (or more frequent) basis and for a build to be conducted with each significant change.

The continuous integration principle of Agile development has a cultural implication for the development group. Forcing developers to integrate their work with other developers frequently—at least daily—exposes integration issues and conflicts much earlier than is the case with waterfall development. However, to achieve this benefit, developers have to communicate with each other much more frequently—something that runs counter to the image of the solitary genius coder working for weeks or months on a module before she is "ready" to send it out in the world. That seed of open, frequent communication blooms in DevOps.

Continuous Testing

The testing piece of DevOps is easy to overlook—until you get burned. As one industry expert puts it, "The cost of quality is the cost of failure." While continuous integration and delivery get the lion's share of the coverage, continuous testing is quietly finding its place as an equally critical piece of DevOps.

Continuous testing is not just a QA function. In fact, it starts in the development environment. The days are over when developers could simply throw the code over the wall to QA and say, "Have at it." In a DevOps environment, everyone is involved in testing.

Developers make sure that, along with delivering error-free code, they provide test data sets. They also help test engineers configure the testing environment to be as close to the production environment as possible.

On the QA side, the big need is speed. After all, if the QA cycle takes days and weeks, you're right back into a long, drawn-out waterfall kind of schedule. Test engineers meet the challenge of quick turnaround by not only automating much of the test process but also redefining test methodologies:

Rather than making test a separate and lengthy sequence in the larger deployment process, Continuous Delivery practitioners roll out small upgrades almost constantly, measure their performance, and quickly roll them back as needed.

Although it may come as a surprise, the operations function has an important role to play in testing and QA:

Operations has access to production usage and load patterns. These patterns are essential to the QA team for creating a load test that properly exercises the application.

Operations can also ensure that monitoring tools are in place and test environments are properly configured. They can participate in functional, load, stress, and leak tests and offer analysis based on their experience with similar applications running in production.

The payoff from continuous testing is well worth the effort. The test function in a DevOps environment helps developers to balance quality and speed. Using automated tools reduces the cost of testing and allows test engineers to leverage their time more effectively. Most importantly, continuous testing shortens test cycles by allowing integration testing earlier in the process.

Continuous testing also eliminates testing bottlenecks through virtualized dependent services, and it simplifies the creation of virtualized test environments that can be easily deployed, shared, and updated as systems change. These capabilities reduce the cost of provisioning and maintaining test environments, and they shorten test cycle times by allowing integration testing earlier in life cycle.

Rather than making test a separate and lengthy sequence in the larger deployment process, Continuous Delivery practitioners roll out small upgrades almost constantly, measure their performance, and quickly roll them back as needed.

Continuous Delivery

In the words of one commentator, "Continuous Delivery is nothing but taking this concept of continuous integration to the next step." Instead of ending at the door of the development lab, continuous integration in DevOps extends to the entire release chain, including QA and operations. The result is that individual releases are far less complex and come out much more frequently.

The actual release frequency varies greatly depending on the company's legacy and goals. For example, one Fortune 100 company improved its release cycle from once a year to once a quarter—a release rate that seems glacial compared to the hundreds of releases an hour achieved by Amazon.

Exactly what gets released varies as well. In some organizations, QA and operations triage potential releases: many go directly to users, some go back to development, and a few simply are not deployed at all. Other companies—Flickr is a notable example—push everything that comes from developers out to users and count on real-time monitoring and rapid remediation to minimize the impact of the rare failure.

Continuous Monitoring

Given the sheer number of releases, there's no way to implement the kind of rigorous pre-release testing that characterizes waterfall development. Therefore, in a DevOps environment, failures must be found and fixed in real time. How do you do that? A big part is continuous monitoring.

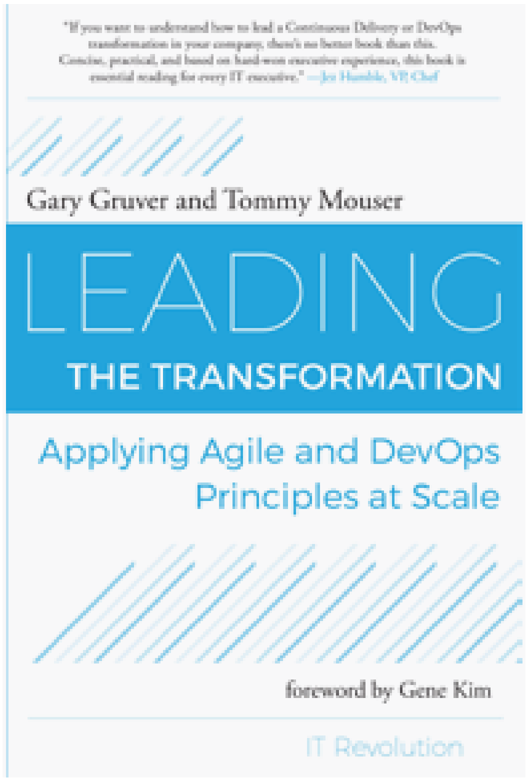
According to one pundit, the goals of continuous monitoring are to quickly determine when a service is unavailable, understand the underlying causes, and most importantly, apply these learnings to anticipate problems before they occur. In fact, some monitoring experts advocate that the definition of a service must include monitoring—they see it as integral to service delivery.

Like testing, monitoring starts in development. The same tools that monitor the production environment can be employed in development to spot performance problems before they hit production.

Two kinds of monitoring are required for DevOps: server monitoring and application performance monitoring. Monitoring discussions quickly get down to tools discussions, because there is no effective monitoring without the proper tools.

→ **For a list of DevOps tools (and more DevOps-related content), visit New Relic's DevOps Hub. Download the entire *Navigating DevOps* ebook.**

Business Objectives Specific to Scaling DevOps


"If you want to understand how to lead a Continuous Delivery or DevOps transformation in your company, there's no better book than this. Concise, practical, and based on hard-won executive experience, this book is essential reading for every IT executive." —Jez Humble, VP, Chef

Gary Gruver and Tommy Mouser

LEADING THE TRANSFORMATION

Applying Agile and DevOps Principles at Scale

foreword by Gene Kim

IT Revolution

Excerpt from
Leading the Transformation: Applying Agile and DevOps Principles at Scale
Gary Gruver and Tommy Mouser
IT Revolution, 2015

The fundamental Agile principle of releasing frequently tends to get overlooked or ignored by organizations that approach Agile transformations by scaling teams. It has been so overlooked by these organizations that new practices called DevOps and Continuous Delivery (CD) have begun to emerge to address this gap. In DevOps, the objective is to blur the lines between Development and Operations so that new capabilities flow easier from Development into Production. On a small scale, blurring the lines between Development and Operations at the team level improves the flow. In large organizations, this tends to require more structured approaches like CD. Applying these concepts at scale is typically the source of the biggest breakthroughs in improving the efficiency and effectiveness of software development in large organizations, and it should be a key focus of any large-scale transformation and is a big part of this book.

In this book we purposefully blur the line between the technical solutions like CD and the cultural changes associated with DevOps under the concept of applying DevOps principles at scale, because you really can't do one without the other. DevOps and CD are concepts that are gaining a lot of momentum in the industry because they are addressing the aforementioned hole in the delivery process. That said, since these ideas are so new, not everyone agrees on their definitions.

CD tends to cover all the technical approaches for improving code releases and DevOps tends to be focused on the cultural changes. From our perspective you really can't make the technical changes without the cultural shifts. Therefore, for the proposes of this book we will define DevOps as processes and approaches for improving the efficiency of taking newly created code out of development and to your customers. This includes all the technical capabilities like CD and the cultural changes associated with Development and Operations groups working together better.

There are five main objectives that are helpful for executives to keep in mind when transforming this part of the development process so they can track progress and have a framework for prioritizing the work.

1. Improve the quality and speed of feedback for developers

Developers believe they have written good code that meets its objectives and feel they have done a good job until they get feedback telling them otherwise. If this feedback takes days or weeks to get to them, it is of limited value to the developers' learning. If you approach a developer weeks after they have written the code and ask them why they wrote it that way or tell them that it broke these other things, they are likely to say, "What code?," "When?," or "Are you sure it was me?" If instead the system was able to provide good feedback to the developer within a few hours or less, they will more likely think about their coding approach and will learn from the mistake.

The objective here is to change the feedback process so that rather than beating up the developer for making mistakes they don't even remember, there is a real-time process that helps them improve. Additionally, you want to move this feedback from simply validating the code to making sure it will work efficiently in production so you can get everyone focused on delivering value all the way to the customer. Therefore, as much as possible you want to ensure the feedback is coming from testing in an environment that is as much like production as possible. This helps to start the cultural transformation across Development and Operations by aligning them on a common objective.

The Operations team can ensure their concerns are addressed by starting to add their release criteria to these test environments. The Development teams then start to learn about and correct issues that would occur in production because they are getting this feedback on a daily basis when it is easy to fix. **Executives must ensure that both Development and Operations make the cultural shift of using the same**

tools and automation to build, test, and deploy if the transformation is going to be successful.

2. Reduce the time and resources required to go from functionality complete or release branching to production

The next objective is reducing, as much as possible, the time and resources required to go from functionality complete or release branching to production. For large, traditional organizations, this can be a very lengthy and labor intensive process that doesn't add any value and makes it impossible to release code economically and on a more frequent basis. The work in this phase of the program is focused on finding and fixing defects to bring the code base up to release quality. Reducing this time requires automating your entire regression suite and implementing all-new testing so that it can be run every day during the development phase to provide rapid feedback to the developers. It also requires teaching your Development organization to add new code without breaking existing functionality, such that the main code branch is always much closer to release quality.

Once you have daily full-regression testing in place, the time from functionality complete or branch cut to production can go down dramatically because the historic effort of manually running the entire regression suite and finding the defects after development is complete has been eliminated. Ideally, for very mature organizations, this step enables you to keep trunk quality very close to production quality, such that you can use continuous deployment techniques to deploy into production with multiple check-ins a day.

This goal of a production-level trunk is pretty lofty for most traditional organizations, and lots of businesses customers would not accept overly frequent releases. Working towards this goal, though, enables you to support delivery of the highest-priority features on a regular cadence defined by the business instead of one defined by the development process capabilities. Additionally, if the developers are working on a development trunk that is very unstable and full of defects, the likely reaction to a test failure is “that's not my fault, I'm sure that defect was already there.” On the other hand, if the trunk is stable and of high quality, they are much more likely to realize that a new test failure may in fact be the result of the code they just checked in. With this realization you will see the Development community begin to take ownership for the quality of the code they commit each day.

3. Improve the repeatability of the build, deploy, and test process

In most large, traditional organizations, the repeatability of the entire build, test, and deploy process can be a huge source of inefficiencies. For small organizations with independent applications, a few small Scrum teams working together can easily accomplish this process improvement. For large organizations that have large groups of engineers working together on a leveraged code base or lots of different applications that need to work together, this is a very different story. Designing a deployment pipeline for how you build up and test these systems is important. It needs to be a structured, repeatable process, or you are going to end up wasting lots of time and resources chasing issues you can't find and/or trying to localize the offending code in a large, complex system. The objective here is to make sure you have a well-designed, repeatable process.

4. Develop an automated deployment process that will enable you to quickly and efficiently find any deployment or environment issues

Depending on the type of application, the deployment process may be as simple as FTPing a file to a printer or as complex as deploying and debugging code to hundreds or thousands of servers. If the application requires deploying to lots of servers, debugging the deployment process can be as complicated as finding code defects in a large system. Additionally, it can complicate the process of finding code issues because the system test failures can be either be code or deployment related. Therefore, it is important to create a deployment process that can quickly identify and isolate any deployment issues before starting system testing to find code issues.

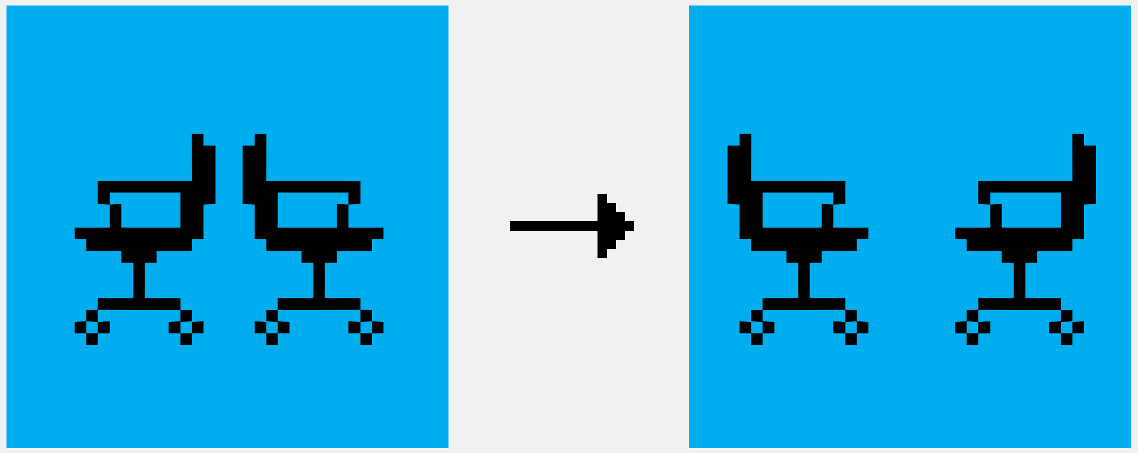
5. Remove the duplication of work that comes from supporting multiple branches of similar code

Another important objective at this stage of the transformation is taking duplication out of the process. The goal is to have your marginal costs for manufacturing and deploying software to be almost zero. This starts to break down when you are working with more than one branch of similar code. Every time you find a code issue, you need to make sure it is correctly ported and working on every associated branch. It breaks down even more when your qualification requires any manual testing, which is expensive and time-consuming on these different branches.

There are lots of different reasons you will hear for needing different branches. Some reasons are customer driven. Other reasons for different branches include the belief that you need branches at different levels of stability or that you will have to bring in architectural changes and different branches make that easier. All these issues have been solved by organizations that have learned how to develop on one trunk and have realized the efficiencies of this approach. It will take time and will be an organizational change management challenge, but the duplicate work associated with multiple branches is a huge inefficiency in most large software development organizations. Every time you see a branch that lasts more than a couple of days, you should think of it as duplicate costs in the system impacting the inherent economical benefits of your software. Repeat this phrase until you become comfortable with it: "Branches are evil; branches are evil; branches are evil." If you are working in a branch-heavy organization, this may take some time to address, but every time you see a branch you should ask why it's there and look for process changes that will address the same need without creating branches.

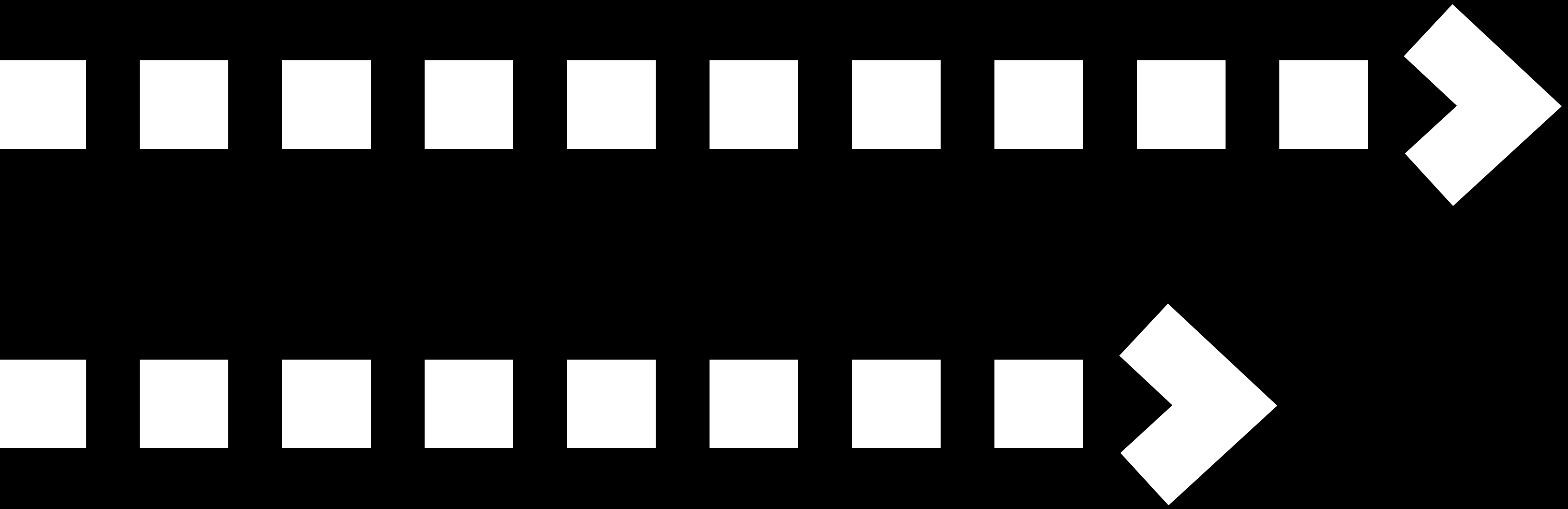
→ Order a copy of *Leading the Transformation*. To contact Gary Gruver about consulting, please visit practicallargescaleagile.com or find him on Twitter at @GRUVERGary

Win-Win Relationship between Dev and Ops



It's not Dev versus Ops, it's Dev plus Ops.

When the outcome of a Dev and Ops interaction is win-win, IT performance wins.



The First Way: System Flow from Left to Right

Bill Learns about Bottlenecks

The Phoenix Project
A Novel About IT, DevOps, and Helping Your Business Win
REVISED WITH NEW RESOURCE GUIDE
Gene Kim, Kevin Behr, and George Spafford

Excerpt from
The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win

Gene Kim, Kevin Behr, and George Spafford

IT Revolution, 2014

"I came as fast as I could." I say.

Erik merely grunts and gestures for me to follow him. Again, we climb the staircase and stand on the catwalk overlooking the plant floor.

"So tell me what you see," he says, gesturing toward the plant floor.

I look down, confused, not knowing what he wants to hear. Starting with the obvious, I say, "Like last time, I see raw materials coming in from the loading docks on the left. And on the right, I see finished goods leaving the other set of loading docks."

Surprisingly, Erik nods approvingly. "Good. And in between?"

I look down at the scene. Part of me feels foolish, afraid of looking like the Karate Kid being quizzed by Mr. Miyagi. But I asked for this meeting, so I just start talking. "I see materials and work in process, flowing from left to right—but, obviously, moving very slowly."

Erik peers over the catwalk, and says, "Oh, really? Like some sort of river?"

He turns to me, shaking his head with disgust, "What do you think this is, some sort of poetry reading class? Suddenly, WIP is like water running over smooth stones? Get serious. How would a plant manager answer the question? From where to where does the work go, and why?"

Trying again, I say, "Okay, okay. WIP goes from work center to work center, as dictated by the bill of materials and routings. And all that is in the job order, which was released at that desk over there."

"That's better," Erik says. "And can you find the work centers where the plant constraints are?"

I know that Erik had told me on that first odd trip to this plant.

"The heat treat ovens and paint curing booths," I say suddenly. "There," I say, after scanning the plant floor and finally spotting a set of large machines by the far wall. "And there," I say, pointing at the large rooms with signs saying, "Paint Booth #30-a" and "Paint Booth #30-b."

"What do you think this is, some sort of poetry reading class? Suddenly, WIP is like water running over smooth stones? Get serious. How would a plant manager answer the question? From where to where does the work go, and why?"

"Good. Understanding the flow of work is key to achieving the First Way," Erik says, nodding. More sternly, he asks, "So now, tell me again which work centers you've determined to be the constraints in your organization?"

I smile, answering easily, "Brent. We talked about that before." He scoffs, turning back to look at the plant floor.

"What?" I nearly shout. "How can it not be Brent? You even congratulated me when I told you it was Brent a couple of weeks ago!"

"Suddenly Brent is a robotic heat treat oven? You're telling me your equivalent of that paint curing booth down there is Brent?" he says with mock disbelief. "You know, that might be the dumbest thing I've ever heard."

He continues, "So, where would that leave your two managers, Chester and Penelope? Let me guess. Maybe they're equivalent to that drill press station and that stamping machine over there? Or maybe it's that metal grinder?"

Erik looks sternly at me, "Get serious. I asked you what work centers are your constraints. Think."

Completely confused, I look back down at the plant floor.

I know that part of the answer is Brent. But when I blurt it out so confidently, Erik all but smacks me on the head. Again.

Erik seems aggravated that I named an actual person, suggesting that Brent was a piece of equipment.

I look again at the heat treat oven. And then I see them. There are two people wearing coveralls, hard hats, and goggles. One is in front of a computer screen, punching in something, while the other is inspecting a pile of parts on a loading pallet, scanning something with his handheld computer.

"Oh," I say, thinking out loud. "The heat treat oven is a work center, which has workers associated with it. You asked what work centers are our constraints, and I told you that it was Brent, which can't be right, because Brent isn't a work center.

"Brent is a worker, not a work center," I say again. "And I'm betting that Brent is probably a worker supporting way too many work centers. Which is why he's a constraint."

"Now we're getting somewhere!" Erik says, smiling. Gesturing broadly at the plant floor below, he says, "Imagine if twenty-five percent of all the work centers down there could only be operated by one person named Brent. What would happen to the flow of work?"

I close my eyes to think.

"Work wouldn't complete on time, because Brent can only be at one work center at a time," I say. Enthusiastically, I continue, "That's exactly what's happening with us. I know that for a bunch of our planned changes, work can't even start if Brent isn't on hand. When that happens, we'll escalate to Brent, telling him to drop whatever he's doing, so some other work center can get going. We'll be lucky if he can stay there long enough for the change to be completely implemented before he's interrupted by someone else."

"Exactly!" he says.

I'm slightly dismayed at the warm feeling of approval that I feel in response.

"Obviously," he continues, "every work center is made up of four things: the machine, the man, the method, and the measures. Suppose for the machine, we select the heat treat oven. The men are the two people required to execute the predefined steps, and we obviously will need measures based on the outcomes of executing the steps in the method."

I frown. These factory terms are vaguely familiar from my MBA years. But I never thought they'd be relevant in the IT domain.

Looking for some way to write this down, I realize I left my clipboard in my car. I pat my pockets and find a small crumpled index card in my back pocket.

I hurriedly write down, "Work center: machine, man, method, measure."

Erik continues, "Of course, on this plant floor, you don't have one quarter of the work centers dependent upon one person. That would be absurd. Unfortunately for you, you do. That's why when Brent takes a vacation, all sorts of work will just grind to a halt, because only Brent knows how to complete certain steps—steps that probably only Brent even knew existed, right?"

I nod, unable to resist groaning. "You're right. I've heard my managers complain that if Brent were hit by the proverbial bus, we'd be completely up the creek. No one knows what's in Brent's head. Which is one of the reasons I've created the level 3 escalation pool."

I quickly explain what I did to prevent escalations to Brent during outages to keep him from being interrupted by unplanned work and how I've attempted to do the same thing for planned changes.

"Good," he says. "You're standardizing Brent's work so that other people can execute it. And because you're finally getting those steps documented, you're able to enforce some level of consistency and quality, as well. You're not only reducing the number of work centers where Brent is required, you're generating documentation that will enable you to automate some of them."

He continues, "Incidentally, until you do this, no matter how many more Brents you hire, Brent will always remain your constraint. Anyone you hire will just end up standing around."

I nod in understanding. This is exactly as Wes described it. Even though he got the additional headcount to hire more Brents, we never were able to actually increase our throughput.

I feel a sudden sense of exhilaration as the pieces fall into place in my head. He's confirming some of my deeply held intuitions and providing an underpinning theory for why I believe them.

My elation is short-lived. He looks me over disapprovingly, "You're asking about how to lift the project freeze. Your problem is that you keep confusing two things. Until you can separate them in your head, you'll just walk around in circles."

He starts walking and I hurry after him. Soon, we're standing over the middle of the plant floor.

"You see that work center over there, with the yellow blinking light?" he asks, pointing.

When I nod, he says, "Tell me what you see."

Wondering what it would take to have a normal conversation with him, I resume my dumb trainee role. "Some piece of machinery is apparently down—that's what I'm guessing the blinking light indicates. There are five people huddled off to the side, including what looks like two managers. They all look concerned. There are three more people crouched down, looking into what I'm guessing is the machine inspection panel. They have flashlights and—yeah—they're also holding screwdrivers—definitely a machine down..."

I feel a sudden sense of exhilaration as the pieces fall into place in my head. He's confirming some of my deeply held intuitions and providing an underpinning theory for why I believe them.

"Good guess," he says. "That's probably a computerized grinder that is out of commission, and the maintenance team is working on getting it back online. What would happen if every piece of equipment down there needs Brent to fix it?"

I laugh. "Every outage escalated immediately to Brent."

"Yes." He continues, "Let's start with your first question. Which projects are safe to release when the project freeze is lifted? Knowing how work flows through certain work centers and how some work centers require Brent and some do not, what do you think the answer is?"

I slowly repeat what Erik just recited, trying to piece together the answer.

"I got it," I say, smiling. "The candidate projects which are safe to release are those that don't require Brent."

I smile even wider when he says, "Bingo. Pretty simple, yes?"

My smile disappears as I think through the implications. "Wait, how do I know which projects don't require Brent? We never think we actually need Brent until we're half-way through the work!"

I immediately regret asking the question as Erik glares at me. "I'm supposed to give you the answer to everything that you're too disorganized to be able to figure out for yourself?"

"Sorry. I'll figure it out," I say quickly. "You know, I'll be so relieved when we finally know all the work that actually requires Brent."

"Damn right," he says. "What you're building is the bill of materials for all the work that you do in IT Operations. But instead of a list of parts and subassemblies, like moldings, screws, and casters, you're cataloging all the prerequisites of what you need before you can complete the work—like laptop model numbers, specifications of user information, the software and licenses needed, their configurations, version information, the security and capacity and continuity requirements, yada yada..."

He interrupts himself, saying, "Well, to be more accurate, you're actually building a bill of resources. That's the bill of materials along with the list of the required work centers and the routing. Once you have that, along with the work orders and your resources, you'll finally be able to get a handle on what your capacity and demand is. This is what will enable you to finally know whether you can accept new work and then actually be able to schedule the work."

Amazing. I think I almost get it.

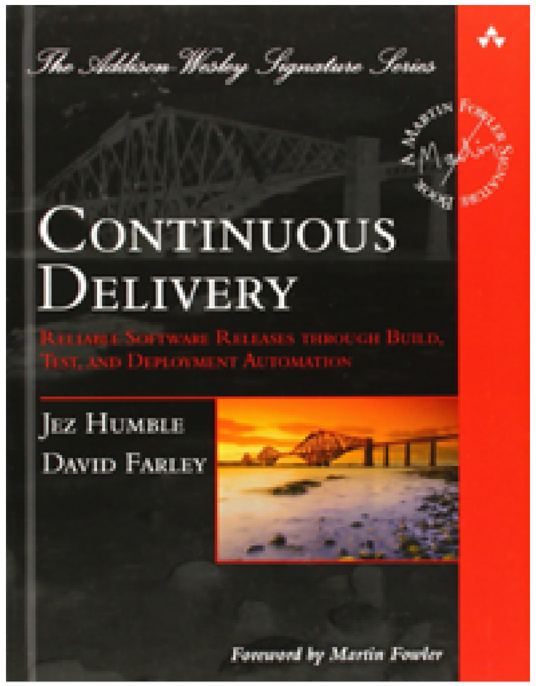
Peer-Reviewed Change Approval Process

We found that when external approval (e.g., change approval boards) was required in order to deploy to production, IT performance decreased. But

when the technical team held itself accountable for the quality of its code through peer review, performance increased.

Surprisingly, the use of external change approval processes had no impact on restore times and had only a negligible effect on reducing failed changes. In other words, external change approval boards had a big negative impact on throughput, with negligible impact on stability.

From "Top 5 Predictors of IT Performance," 2014 State of DevOps Report | © 2014 Puppet Labs | puppetlabs.com



Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation

Jez Humble and David Farley

Addison-Wesley Professional, 2010

Order a copy.

In the IT value stream, success is all about the left-to-right flow of work from Development into IT Operations. Probably the best embodiment of this work is Jez Humble and David Farley's seminal book *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.*

They codify many of the techniques required to replicate the famous 2009 Velocity Conference presentation, "10+ Deploys per Day: Dev and Ops Cooperation at Flickr," given by John Allspaw and Paul Hammond, as well as the Agile system administration movement.

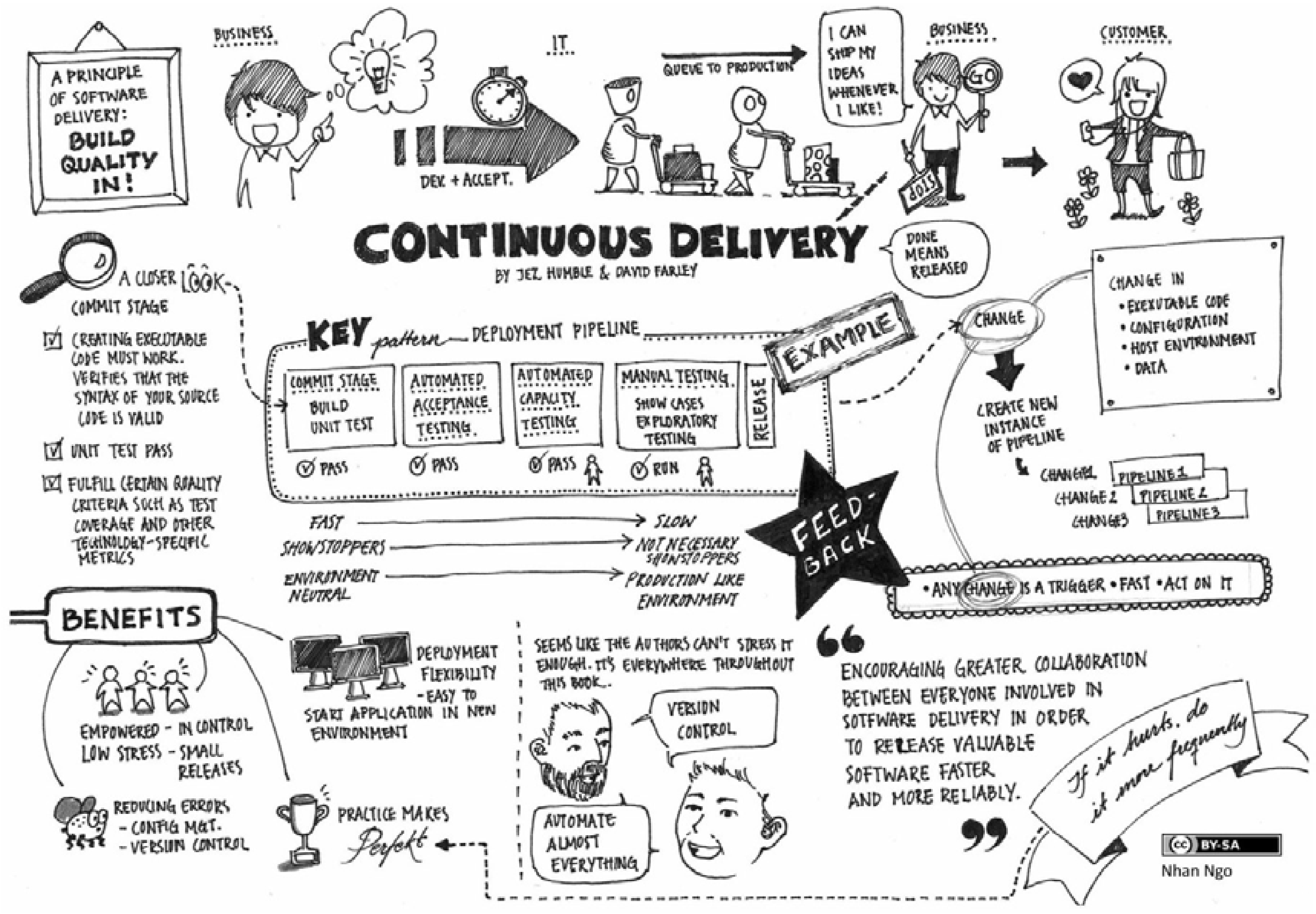
Continuous Delivery is the extension of continuous integration, which are the Development practices that include continuous builds, continuous testing, daily integration of branches back into trunk, testing in a clone of the production environment, etc. Continuous Delivery techniques extend these processes all the way into the production environment.

This makes continuous deployment a prerequisite for the high deploy rates characterized by DevOps and is therefore a needed skill set for the modern DevOps practitioner. It will also be the salvation for a generation of ITSM practitioners.

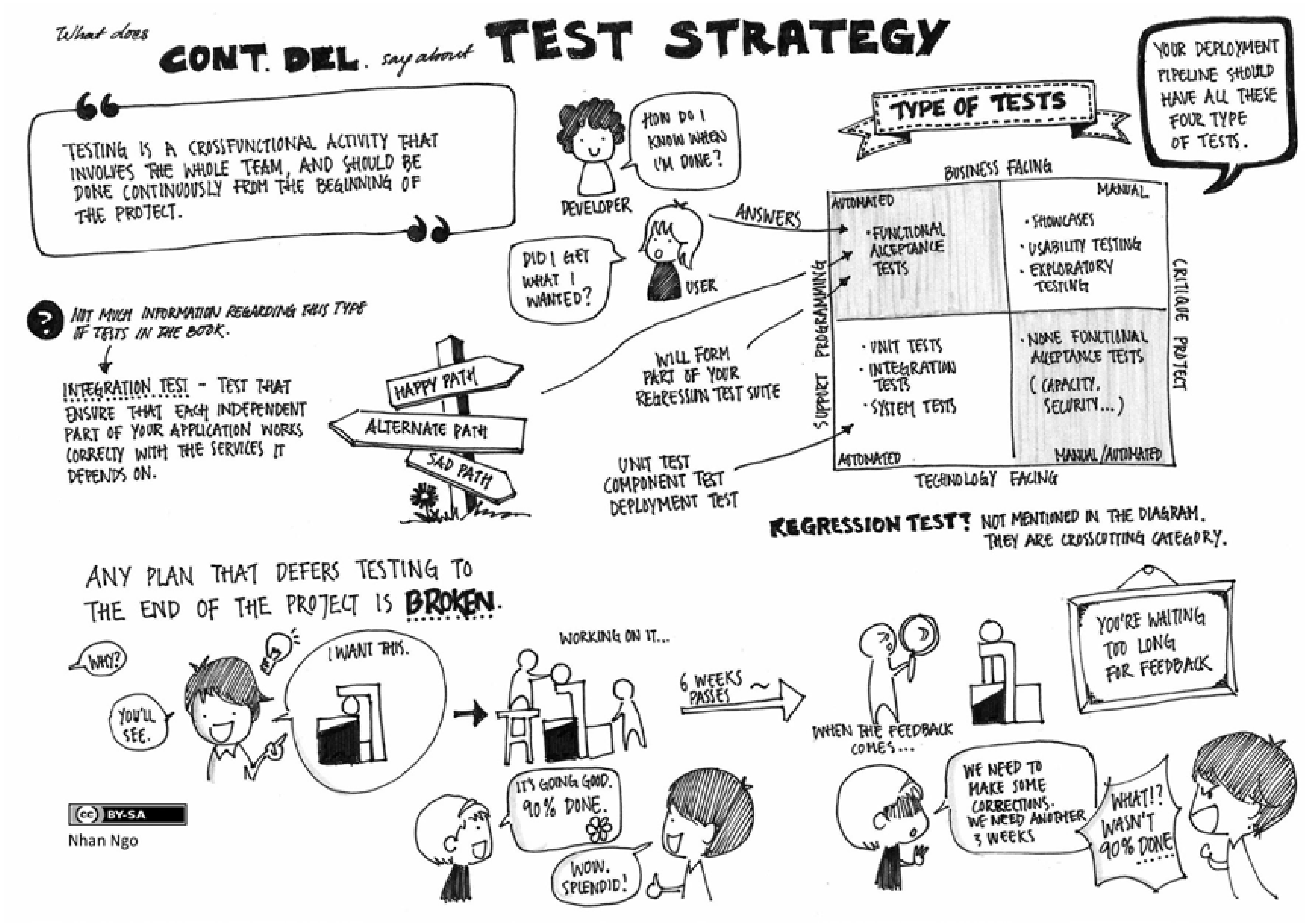
Continuous Delivery is the perfect embodiment of the First, Second, and Third Ways in The Phoenix Project, *as it emphasizes small batch sizes (e.g., check into trunk daily), stopping the line when problems occur (e.g., no new work allowed when builds, tests, or deployments fail; elevating the integrity of the system of work over the work itself), and the need to continually build the validation tests necessary to either prevent failures in production, or, at the very least, detect and correct them quickly (e.g., the transition from manual process reviews to automated tests, especially in the ITSM release, change, and configuration process areas).*

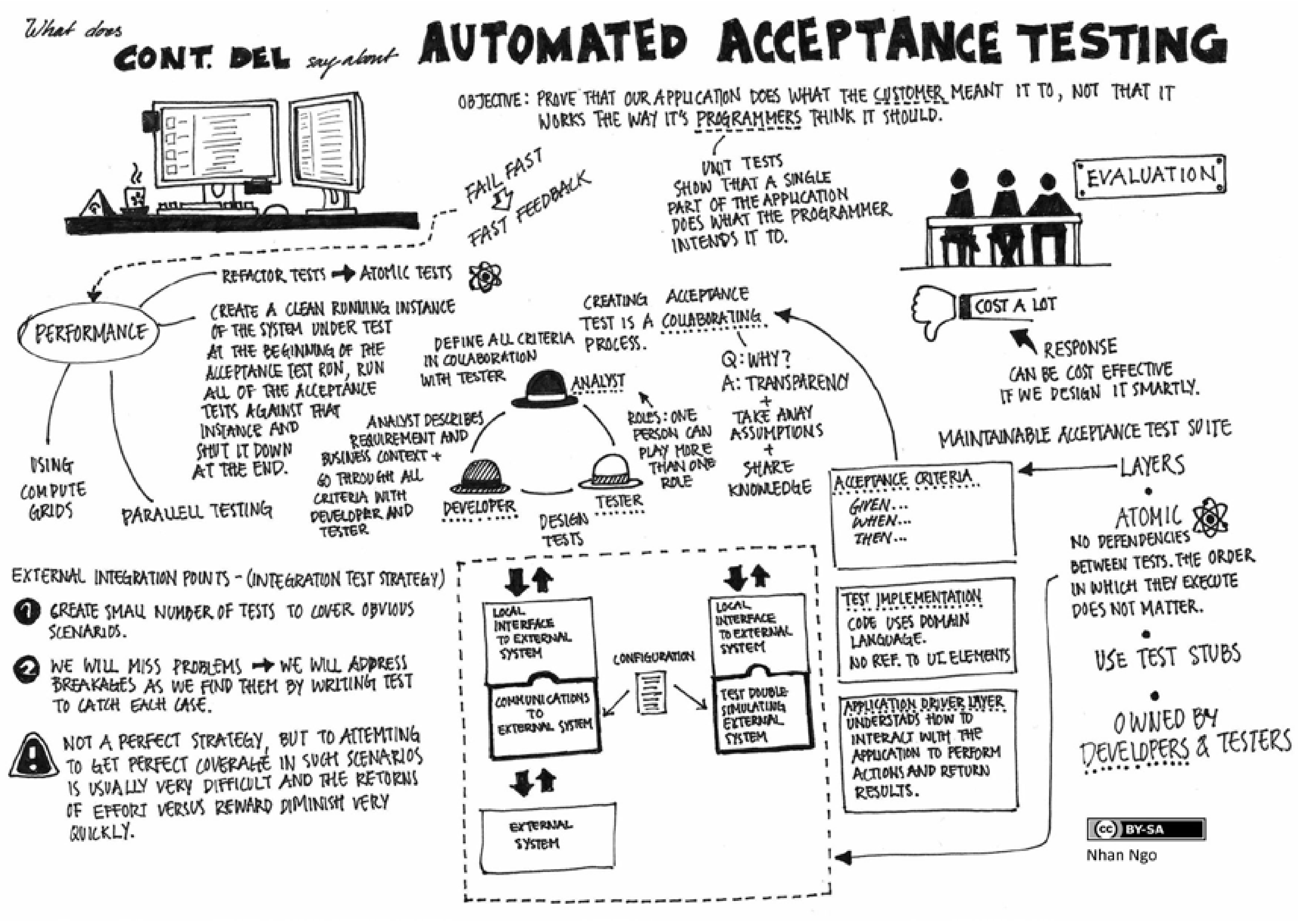
–GENE KIM

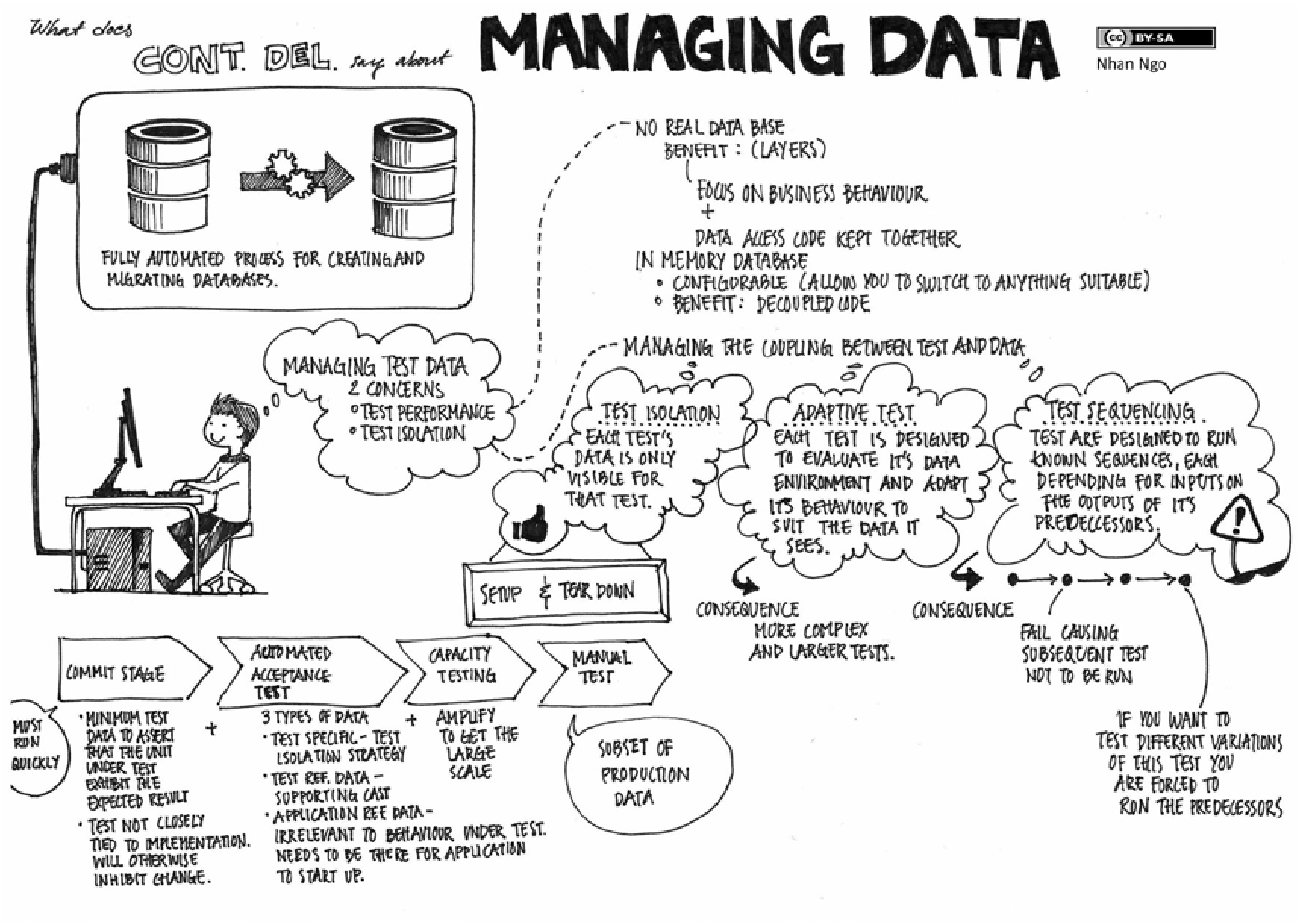
Visualizing *Continuous Delivery*: Illustrations by Nhan Ngo

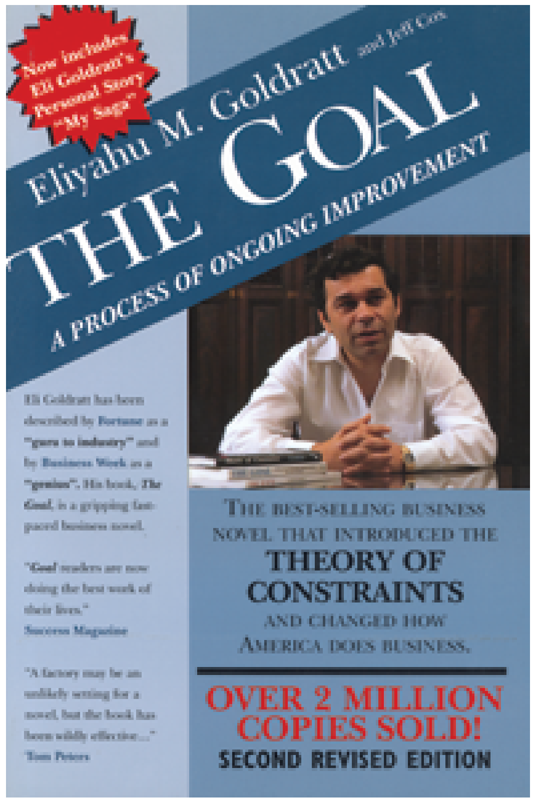

A PRINCIPLE OF SOFTWARE DELIVERY: BUILD QUALITY IN!
BUSINESS
IT
DEV + ACCEPT.
QUEUE TO PRODUCTION
I CAN SHIP MY IDEAS WHENEVER I LIKE!
BUSINESS
GO
STOP
CUSTOMER
CONTINUOUS DELIVERY
BY JEZ HUMBLE & DAVID FARLEY
DONE MEANS RELEASED
A CLOSER LOOK
COMMIT STAGE
☑ CREATING EXECUTABLE CODE MUST WORK. VERIFIES THAT THE SYNTAX OF YOUR SOURCE CODE IS VALID
☑ UNIT TEST PASS
☑ FULFILL CERTAIN QUALITY CRITERIA SUCH AS TEST COVERAGE AND OTHER TECHNOLOGY-SPECIFIC METRICS
KEY pattern — DEPLOYMENT PIPELINE
COMMIT STAGE BUILD UNIT TEST
AUTOMATED ACCEPTANCE TESTING
AUTOMATED CAPACITY TESTING
MANUAL TESTING SHOW CASES EXPLORATORY TESTING
RELEASE
✓ PASS
✓ PASS
✓ PASS
✓ RUN
EXAMPLE
FAST → SLOW
SHOWSTOPPERS → NOT NECESSARY SHOWSTOPPERS
ENVIRONMENT NEUTRAL → PRODUCTION LIKE ENVIRONMENT
FEED-BACK
CHANGE
CREATE NEW INSTANCE OF PIPELINE
CHANGE IN • EXECUTABLE CODE • CONFIGURATION • HOST ENVIRONMENT • DATA
CHANGE1 PIPELINE 1
CHANGE 2 PIPELINE 2
CHANGE3 PIPELINE 3
• ANY CHANGE IS A TRIGGER • FAST • ACT ON IT
BENEFITS
EMPOWERED - IN CONTROL
LOW STRESS - SMALL RELEASES
REDUCING ERRORS - CONFIG MGT. - VERSION CONTROL
DEPLOYMENT FLEXIBILITY - EASY TO START APPLICATION IN NEW ENVIRONMENT
PRACTICE MAKES Perfekt
SEEMS LIKE THE AUTHORS CAN'T STRESS IT ENOUGH. IT'S EVERYWHERE THROUGHOUT THIS BOOK.
VERSION CONTROL
AUTOMATE ALMOST EVERYTHING
“ENCOURAGING GREATER COLLABORATION BETWEEN EVERYONE INVOLVED IN SOTFWARE DELIVERY IN ORDER TO RELEASE VALUABLE SOFTWARE FASTER AND MORE RELIABLY.”
If it hurts, do it more frequently
(cc) BY-SA
Nhan Ngo

Nhan Ngo, a QA engineer at Spotify, made four fabulous sketchnote illustrations while reading *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* by Jez Humble and David Farley. She has very generously made them available for inclusion here under a Creative Commons license, and we very enthusiastically thank her for it.


What does CONT. DEL. say about TEST STRATEGY
"TESTING IS A CROSSFUNCTIONAL ACTIVITY THAT INVOLVES THE WHOLE TEAM, AND SHOULD BE DONE CONTINUOUSLY FROM THE BEGINNING OF THE PROJECT."
HOW DO I KNOW WHEN I'M DONE?
DEVELOPER
DID I GET WHAT I WANTED?
USER
ANSWERS
TYPE OF TESTS
YOUR DEPLOYMENT PIPELINE SHOULD HAVE ALL THESE FOUR TYPE OF TESTS.
BUSINESS FACING
AUTOMATED
· FUNCTIONAL ACCEPTANCE TESTS
MANUAL
· SHOWCASES
· USABILITY TESTING
· EXPLORATORY TESTING
SUPPORT PROGRAMMING
CRITIQUE PROJECT
· UNIT TESTS
· INTEGRATION TESTS
· SYSTEM TESTS
AUTOMATED
· NONE FUNCTIONAL ACCEPTANCE TESTS (CAPACITY, SECURITY...)
MANUAL/AUTOMATED
TECHNOLOGY FACING
NOT MUCH INFORMATION REGARDING THIS TYPE OF TESTS IN THE BOOK.
INTEGRATION TEST - TEST THAT ENSURE THAT EACH INDEPENDENT PART OF YOUR APPLICATION WORKS CORRECTLY WITH THE SERVICES IT DEPENDS ON.
HAPPY PATH
ALTERNATE PATH
SAD PATH
WILL FORM PART OF YOUR REGRESSION TEST SUITE
UNIT TEST
COMPONENT TEST
DEPLOYMENT TEST
REGRESSION TEST? NOT MENTIONED IN THE DIAGRAM. THEY ARE CROSSCUTTING CATEGORY.
ANY PLAN THAT DEFERS TESTING TO THE END OF THE PROJECT IS BROKEN.
WHY?
YOU'LL SEE.
I WANT THIS.
WORKING ON IT...
IT'S GOING GOOD. 90% DONE.
WOW. SPLENDID!
6 WEEKS PASSES ~
WHEN THE FEEDBACK COMES...
YOU'RE WAITING TOO LONG FOR FEEDBACK
WE NEED TO MAKE SOME CORRECTIONS. WE NEED ANOTHER 3 WEEKS
WHAT!? WASN'T 90% DONE
BY-SA
Nhan Ngo


What does CONT. DEL say about AUTOMATED ACCEPTANCE TESTING
OBJECTIVE: PROVE THAT OUR APPLICATION DOES WHAT THE CUSTOMER MEANT IT TO, NOT THAT IT WORKS THE WAY IT'S PROGRAMMERS THINK IT SHOULD.
FAIL FAST ⇩ FAST FEEDBACK
UNIT TESTS SHOW THAT A SINGLE PART OF THE APPLICATION DOES WHAT THE PROGRAMMER INTENDS IT TO.
EVALUATION
COST A LOT
RESPONSE CAN BE COST EFFECTIVE IF WE DESIGN IT SMARTLY.
REFACTOR TESTS → ATOMIC TESTS
PERFORMANCE
CREATE A CLEAN RUNNING INSTANCE OF THE SYSTEM UNDER TEST AT THE BEGINNING OF THE ACCEPTANCE TEST RUN, RUN ALL OF THE ACCEPTANCE TESTS AGAINST THAT INSTANCE AND SHUT IT DOWN AT THE END.
USING COMPUTE GRIDS
PARALLEL TESTING
CREATING ACCEPTANCE TEST IS A COLLABORATING PROCESS.
DEFINE ALL CRITERIA IN COLLABORATION WITH TESTER
ANALYST
ANALYST DESCRIBES REQUIREMENT AND BUSINESS CONTEXT + GO THROUGH ALL CRITERIA WITH DEVELOPER AND TESTER
DEVELOPER
TESTER
DESIGN TESTS
ROLES: ONE PERSON CAN PLAY MORE THAN ONE ROLE
Q: WHY?
A: TRANSPARENCY + TAKE AWAY ASSUMPTIONS + SHARE KNOWLEDGE
MAINTAINABLE ACCEPTANCE TEST SUITE
LAYERS
ACCEPTANCE CRITERIA
GIVEN...
WHEN...
THEN...
TEST IMPLEMENTATION
CODE USES DOMAIN LANGUAGE.
NO REF. TO UI ELEMENTS
APPLICATION DRIVER LAYER UNDERSTANDS HOW TO INTERACT WITH THE APPLICATION TO PERFORM ACTIONS AND RETURN RESULTS.
ATOMIC
NO DEPENDENCIES BETWEEN TESTS. THE ORDER IN WHICH THEY EXECUTE DOES NOT MATTER.
USE TEST STUBS
OWNED BY DEVELOPERS & TESTERS
EXTERNAL INTEGRATION POINTS - (INTEGRATION TEST STRATEGY)
1 CREATE SMALL NUMBER OF TESTS TO COVER OBVIOUS SCENARIOS.
2 WE WILL MISS PROBLEMS → WE WILL ADDRESS BREAKAGES AS WE FIND THEM BY WRITING TEST TO CATCH EACH CASE.
NOT A PERFECT STRATEGY, BUT TO ATTEMPTING TO GET PERFECT COVERAGE IN SUCH SCENARIOS IS USUALLY VERY DIFFICULT AND THE RETURNS OF EFFORT VERSUS REWARD DIMINISH VERY QUICKLY.
LOCAL INTERFACE TO EXTERNAL SYSTEM
COMMUNICATIONS TO EXTERNAL SYSTEM
CONFIGURATION
LOCAL INTERFACE TO EXTERNAL SYSTEM
TEST DOUBLE SIMULATING EXTERNAL SYSTEM
EXTERNAL SYSTEM
BY-SA
Nhan Ngo


What does CONT. DEL. say about
MANAGING DATA
(cc) BY-SA
Nhan Ngo
FULLY AUTOMATED PROCESS FOR CREATING AND MIGRATING DATABASES.
MANAGING TEST DATA
2 CONCERNS
• TEST PERFORMANCE
• TEST ISOLATION
NO REAL DATA BASE
BENEFIT : (LAYERS)
FOCUS ON BUSINESS BEHAVIOUR
+
DATA ACCESS CODE KEPT TOGETHER
IN MEMORY DATABASE
• CONFIGURABLE (ALLOW YOU TO SWITCH TO ANYTHING SUITABLE)
• BENEFIT: DECOUPLED CODE
MANAGING THE COUPLING BETWEEN TEST AND DATA
TEST ISOLATION
EACH TEST'S DATA IS ONLY VISIBLE FOR THAT TEST.
SETUP & TEAR DOWN
ADAPTIVE TEST
EACH TEST IS DESIGNED TO EVALUATE IT'S DATA ENVIRONMENT AND ADAPT ITS BEHAVIOUR TO SUIT THE DATA IT SEES.
CONSEQUENCE
MORE COMPLEX AND LARGER TESTS.
TEST SEQUENCING
TEST ARE DESIGNED TO RUN KNOWN SEQUENCES, EACH DEPENDING FOR INPUTS ON THE OUTPUTS OF IT'S PREDECESSORS.
CONSEQUENCE
FAIL CAUSING SUBSEQUENT TEST NOT TO BE RUN
IF YOU WANT TO TEST DIFFERENT VARIATIONS OF THIS TEST YOU ARE FORCED TO RUN THE PREDECESSORS
COMMIT STAGE
AUTOMATED ACCEPTANCE TEST
CAPACITY TESTING
MANUAL TEST
MUST RUN QUICKLY
• MINIMUM TEST DATA TO ASSERT THAT THE UNIT UNDER TEST EXHIBIT THE EXPECTED RESULT
• TEST NOT CLOSELY TIED TO IMPLEMENTATION. WILL OTHERWISE INHIBIT CHANGE.
+
3 TYPES OF DATA
• TEST SPECIFIC - TEST ISOLATION STRATEGY
• TEST REF. DATA - SUPPORTING CAST
• APPLICATION REF DATA - IRRELEVANT TO BEHAVIOUR UNDER TEST. NEEDS TO BE THERE FOR APPLICATION TO START UP.
+
AMPLIFY TO GET THE LARGE SCALE
SUBSET OF PRODUCTION DATA



The Goal: A Process of Ongoing Improvement

Eliyahu Goldratt and Jeff Cox

North River Press, 1992

Order a copy.

***The Goal* is a Socratic business novel about Alex Rogo,** a plant manager who must fix his productions cost and delivery issues in ninety days or his plant will be shut down. Unable to find a way forward, Alex recalls a chance meeting with a physicist named Jonah. Through a series of phone calls and meetings, Jonah teaches Alex about the Theory of Constraints and the steps to take to eliminate bottlenecks:

- Identify the constraint
- Exploit the constraint
- Subordinate all other activities to the constraint
- Elevate the constraint to new levels
- Find the next constraint

In *The Goal*, the constraints were initially the famous NCX-10 robot, then the heat treat ovens, and then the constraint became market demand. In *The Phoenix Project*, the constraint was initially Brent, because he was always dealing with unplanned work. Then it became the application deployment process, and then the constraint moved outside the organization because the needed MRP application support was outsourced.

My coauthors and I studied this book for nearly a decade, getting ready to write The Phoenix Project. *In many ways, I view our book as an homage to* The Goal. *We attempted to mirror most of the book structure and plot elements while making it contemporary, relevant, and, I hope, more dramatic.*

–GENE KIM

DOES14 - Scott Prugh - CSG - DevOps and Lean in Legacy Environments
0:00 / 25:38

DevOps & Lean In Legacy Environments

presentation by Scott Prugh at DevOps Enterprise Summit 2014

Startups are continually evangelizing DevOps to be able to reduce risk, hasten feedback, and deploy thousands of times a day. But what about the rest of the world that comes from Waterfall, Mainframes, Long Release Cycles, and Risk Aversion? Learn how one company went from 480-day lead times and six-month releases to three-month releases with high levels of automation and increased quality across disparate legacy environments. We will discuss how optimizing people and organizations, increasing the rate of learning, deploying innovative tools, and Lean system thinking can help large-scale enterprises increase throughput while decreasing cost and risk.

The Second Way: Amplify Feedback Loops

Proactive Monitoring

Teams that practice proactive monitoring are able to

diagnose and solve problems faster, and they have a high degree of accountability.

When failures are primarily reported by an external source, such as the network operations center (NOC)—or worse, by customers—IT performance suffers.

If You're Going for Continuous Delivery without Making Testing Your #1, You're Doing It Wrong

Best Practices for Test Automation to Enable Continuous Delivery

XebiaLabs
Deliver Faster

XebiaLabs is a pioneer of automation software for DevOps and Continuous Delivery that helps companies accelerate the delivery of new software. Viktor Clerc is product manager for testing applications at XebiaLabs. You may reach him at vclerc@xebialabs.com.

Release cycles are accelerating as more businesses commit to Agile Development methodologies and adopt DevOps and Continuous Delivery. But, for all the advancement and modernization in development processes and tooling, testing is often forgotten.

The fastest delivery pipeline in the world isn't very useful if most of what you ship is broken. There must be some balance between quality and speed. You need a real-time measure of risk and an accurate overview of the quality of the features going through the pipeline. These are prerequisites for making the right decision on when and what to release—a decision you can't make without putting testing first.

How do you accelerate testing, manage test sets, and make sense of the results in a rapidly shifting landscape? Here are four best practices for putting quality at the heart of your Continuous Delivery initiative.

1. Test automation, yes…but *also* test analysis!

As your software progresses, the number of test jobs grows, and manual testing soon becomes impractical. You also need to be able to run them quickly, or testing becomes a bottleneck. How do you manage unit tests, component tests, system tests, end-to-end tests, browser tests, cross-browser tests, performance tests, usability tests, regression and stability tests?

Automation is the obvious answer, and it's a vital component in any Continuous Delivery setup whose initial overhead is far outweighed by the savings you'll make in the mid- to long term. It enables you to run tests quickly and efficiently. But it's important to continue to allow for manual testing where necessary. There's no substitute for human exploratory testing, for example.

Building test automation into your pipeline is a process that's never finished and that requires constant fine-tuning if you want to extract maximum value. There's always a need to balance available resources and the pressure for greater speed with the right standard of quality and an acceptable level of risk.

Even if you're fully automated, it won't be practical to run every test for every release—you won't have enough resources. How do you decide which test sets to run? To make the right decisions, you need to be able to accurately analyze your existing test results to figure out which tests are required to give you the necessary level of insight into quality and risk for the features currently going through the pipeline.

As your software progresses, the number of test jobs grows, and manual testing soon becomes impractical. You also need to be able to run them quickly, or testing becomes a bottleneck. How do you manage unit tests, component tests, system tests, end-to-end tests, browser tests, cross-browser tests, performance tests, usability tests, regression and stability tests?

2. Understand your test results

Can you say exactly what you've tested in any given moment? Do you have an analysis of what that means for your product quality, or the overall health of your pipeline? Risk, and even quality, can be subjective, as developers, business stakeholders, and testers may not have the same priorities. That's why you want tooling that allows you to change your definitions of quality and risk depending on the decision-maker or the situation.

This is where a new generation of tools needs to come in to combine and analyze complex sets of test results so you can easily visualize the quality level of the features in your delivery pipeline, and make accurate and effective go/no-go decisions. The value of each test you run is dramatically diminished if you aren't doing something with the data it produces. Without a broad overview that encompasses all of your different test sets and test tools, it's impossible to get the big picture.

3. Bring it all together

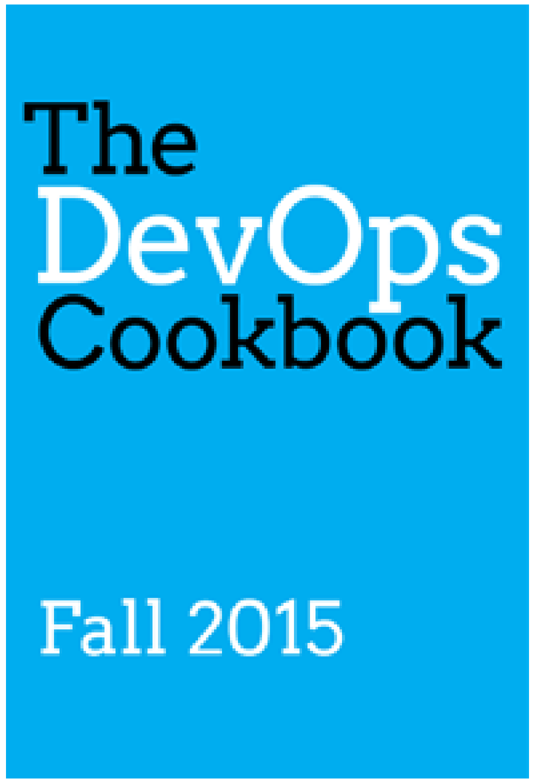
If you want to reap the rewards of Continuous Delivery, then you need to build comprehensive testing into your pipeline from the beginning. This is the only way to manage risk and achieve an acceptable level of quality as you are building your product or service.

The test management tooling you put in place for this needs to be flexible to support a wide variety of different types of testing. It must record all the relevant data from automated and manual testing sessions, and combine that with contextual information on the systems under test. It needs to be able to collate test results from different tools and sources to provide a simple real-time overview, and it should allow you to draw on historical performance to identify trends.

In order to prevent your Continuous Delivery initiative from turning into a high-speed train wreck, you need to put testing at the center of your plans. Automating your tests, and then managing them as a coherent unit, enables your team to meet customer demands faster...with higher-quality software.

→ **Join others who have experienced the benefits of an automated testing solution built for DevOps and Continuous Delivery. Try out the free community edition of XL Test.**

Conduct Blameless Postmortems



Excerpt from
The DevOps Cookbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations

Gene Kim, Patrick Debois, Jez Humble, and John Willis

IT Revolution, 2015

Operating within (and attempting to control) complex systems requires a different mode of management. When failures occur, our goal is not to "name, blame, and shame" the person who caused the failure. Instead, our goal is to maximize our organizational learnings from it: understand how the accident occurred, equip ourselves to create the countermeasures required to prevent it from happening again, and enable quicker detection and recovery.

One of the ways we institutionalize this is to conduct "blameless postmortems," a term coined by John Allspaw. The goal is to create feedback into the system, implementing the goals of a Just Culture, where we are balancing safety and accountability as well as enabling and cultivating high-trust relationships.

In his book *Just Culture: Balancing Safety and Accountability*, Sydney Dekker writes,

Responses to incidents and accidents that are seen as unjust can impede safety investigations, promote fear rather than mindfulness in people who do safety-critical work, make organizations more bureaucratic rather than more careful, and cultivate professional secrecy, evasion, and self-protection . . . A just culture is critical for the creation of a safety culture. Without reporting of failures

> and problems, without openness and information sharing, a safety culture cannot flourish.

Allspaw suggests that "by investigating mistakes in a way that focuses on the situational aspects of a failure's mechanism and the decision-making process of individuals proximate to the failure, an organization can come out safer than it would normally be if it had simply punished the actors involved as a remediation." He continues:

> Anyone who's worked with technology at any scale is familiar with failure. Failure cares not about the architecture designs you slave over, the code you write and review, or the alerts and metrics you meticulously pore through.
>
> So when failures happen, what do we do with those careless humans who caused everyone to have a bad day? Maybe they should be fired. Or maybe they need to be prevented from touching the dangerous bits again. Or maybe they need more training.
>
> This is the traditional view of "human error," which focuses on the characteristics of the individuals involved. It's what Sidney Dekker calls the "Bad Apple Theory"—get rid of the bad apples, and you'll get rid of the human error. Seems simple, right?
>
> We don't take this traditional view at Etsy. We instead want to view mistakes, errors, slips, lapses, etc. with a perspective of learning. Having blameless postmortems on outages and accidents are part of that.

Allspaw isn't just talking about a theory of high learning or abstract cultural changes. He isn't suggesting that everyone is off the hook or accountability is cast aside. He's seen the power of the blameless postmortem in action, and it's about balancing both safety and accountability.

When beginning a program of blameless postmortems, first consider some basic guidelines, who will be included, and an agenda. Bethany Macri, software engineer at Etsy, explains their process: "Anyone interested in issue can attend postmortems. In fact, this was how I learned about what the Payments and Search team was doing, which is mostly decoupled from the rest of the Etsy environment. In 2013, Etsy had over thirty teams, and postmortems are a great way to disseminate knowledge."

We don't just start out feeling safe sharing the details of our mistakes. Those who have made mistakes must understand that by sharing, they will not be exposing themselves to punishment. This is an ongoing process that can unfold based on a set of basic guidelines for our blameless postmortems.

- Gather details from multiple perspectives on failures to ensure we don't single people out
- Empower employees with authority to improve safety based on their detailed accounts of their contributions to failures
- Enable and encourage people who do make mistakes to be the experts on educating the rest of the organization how not to make them in the future
- Accept that humans can decide to make actions or not, and that the judgement of those decisions lies in hindsight

We schedule our postmortems for after the incident has been satisfactorily resolved to focus everyone's energy on the task at hand, not putting out an remaining embers or ongoing flare-ups from the mistake. Alternately, the postmortem should occur in a timely manner so all details, analysis, and hindsight are still fresh and relevant, maximizing learning potential for everyone involved.

While Etsy allows anyone who is interested to attend blameless postmortems, there are certainly other criteria for defining who should be in attendance. Consider inviting the stakeholders who

- introduced the problem
- identified the problem
- responded to the problem
- diagnosed the problem

We will hold postmortems for all significant issues (i.e., P3 or more), which will guide the inclusiveness and frequency of the meetings.

Just as important as who is attending the meeting is the agenda for the meeting. It should be clear so as to keep everyone on topic and to capture as much relevant and detailed information as possible. The atmosphere should be without fear and communication should be without threat of punishment or retribution. Seeking the following information from attending stakeholders should guide the agenda:

- actions taken and at what time
- observed effects of the mistake (ideally, in the form of metrics from any production telemetry)
- expectations
- assumptions
- everyone's understanding of the timeline of events as they occurred
- investigation paths used
- resolutions considered

The most important output of the meeting is to propose effective countermeasures, which should result in corrective work that is prioritized at the highest level (i.e., more important than daily work is the improvement of daily work). This should ideally take the form of work assigned, an owner, and a deadline.

Case Study: Blameless Postmortems

One of the best glimpses into the Etsy culture and how it has shaped their tooling is outlined in Bethany Macri's 2014 DevOpsDays NYC talk, where she describes their internally developed Morgue tool, used to help facilitate the efficient recording and sharing of post-mortem meetings.

At Etsy in 2013, they were averaging sixty million unique monthly visitors, with one deployment happening every twenty minutes. When anything went wrong, they routinely held a postmortem meeting, developing countermeasures at each one. They were even holding postmortems for periods where nothing terrible went wrong (e.g., after the 2012 holiday season, they held one to review what problematic issues could have been done better).

With over four years of operations, they were running into issues with their previous means of recording post-mortems, which was in an internal Wiki page. The problems included that the Wiki had become unsearchable, unsaveable, and with only one person able to enter information, it didn't foster collaboration.

The result was they built Morgue, a tool that enabled them to record for each issue the MTTR, severity (especially whether customers affected), time zones (to make life easier for remote employees), Markdown formatting for the "what happened" field, embedded images, tags, and history. Morgue also records:

- whether the problem was scheduled or unscheduled
- the postmortem owner
- relevant IRC lots (especially important for 3 a.m. issues)
- JIRA tickets and relevant due dates (especially important for management)
- links to customer forums (where customers complain about issues)

Macri remembers how John Allspaw wouldn't let anyone leave the meeting until "there were countermeasures in JIRA, and with an owner and due dates assigned."

At the time of this writing, they were even considering flagging the use of "could" or "should" in the "what happened" field, as counterfactual language is not conducive to blameless postmortems.

As a result of developing and using Morgue, she reports that the number of recorded postmortems has gone up significantly compared to Wiki, especially for P2, P3, and P4. The conjecture is that because the process of documenting postmortems is easier, more people are doing it.

As Macri states, "This is great, because the more postmortems we hold, the more we are learning; and the more we are learning, the more effective an organization we are."

"This is great, because the more postmortems we hold, the more we are learning; and the more we are learning, the more effective an organization we are."

→ Sign up to receive updates about *The DevOps Cookbook* release.

Why Test Data Management Is Broken

Delphix is the market leader in Data as a Service, helping enterprises accelerate application development and achieve DevOps goals. The Delphix DaaS Platform software installs on-premises or in the cloud and automatically provides the right data to the right team at the right time, breaking the key development bottleneck. Over 20% of Fortune 100 companies use Delphix to deliver data 99% faster across development, testing, and reporting environments, driving 50% increases in productivity while improving data security.

With almost every industry becoming part of the software industry—from stock trading to booksellers and from taxi companies to hotels—the ability to quickly develop innovative business applications is driving competitive advantage. And with so much depending on their applications, businesses must look for new ways to improve the testing that ensures software quality.

In particular, today's data-driven applications demand better ways to collect, manage, and deliver the test data that so often determines the success of quality assurance efforts. Test data management is hard, and it continues to confound even progressive IT organizations that have adopted Agile and DevOps practices. In fact, data-related tasks consume up to 60%[1] of application development schedules, shifting attention away from other value-added activities. Three main factors contribute to this dynamic:

- The data that feeds business applications is growing exponentially, both in volume and complexity

- Slow processes and inflexible infrastructure increase the time and cost it takes to deliver test data to QA teams
- Existing tools trade quality and completeness of test data for convenience

In light of these challenges and limitations, organizations struggle to implement truly dependable test data management practices. Too often, testing is pushed to late in the software development life cycle (SDLC), and testers are forced to work with incomplete or compromised data. The end result is rework, delayed releases, and costly bugs that cripple production systems: current industry estimates place the cost of system downtime at $100,000[2] per hour for mission-critical applications.

The Four S's of Test Data Management: How Organizations Manage Their Data Today

Four solutions typify test data management today: *Subsetting*, *Synthetic Data*, *Shared Environments*, and *Standalone Masking*. These approaches have reached widespread adoption and are often used in combination with one another.

Subsetting Strategies

Data subsetting technologies emerged to overcome limitations in copying and moving full, production-sized datasets. In theory, smaller, more portable datasets can serve as substitutes for complete ones as long as they constitute representative samples. In practice, subsets fail to adequately embody the breadth of real-world conditions, leading all too often to errors caught late in testing or slipping through to production.

Synthetic Data

Synthetic data represents an alternative approach in which algorithmically generated test data substitutes for data derived from production sources. On one hand, synthetic data circumvents the security issues involved in distributing "real data" containing potentially sensitive information. On the other hand, it suffers from some of the same shortcomings as subsetting approaches: even large, intelligently generated datasets fail to adequately cover the data permutations attendant in production sources.

Shared Test Data Environments

The inability to provision dedicated QA environments to individual testers means that teams and projects often share test datasets. Theoretically, sharing provides efficiency benefits by giving multiple teams immediate, concurrent access to a common data environment. But in practice, conflicts occur when more than one stakeholder contends for the same resources at the same time. The result is often a low-quality, chaotic test environment in which data changes from test runs collide with each other, yielding inconsistent and untrustworthy test results.

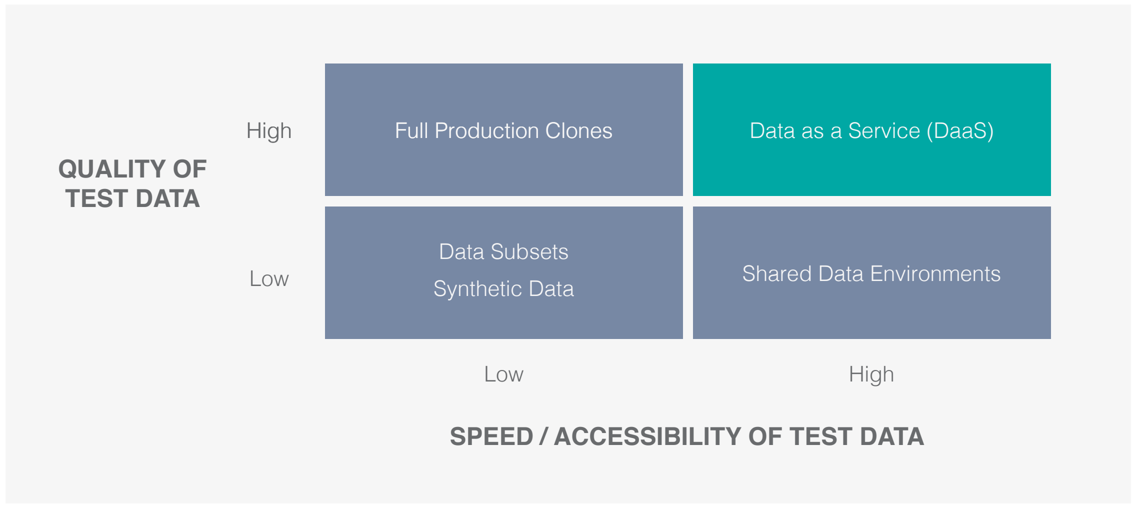
Standalone Masking Solutions

Further complicating test data management is the need to ensure that sensitive information is protected when delivered to non-production environments, including those used for testing. In fact, for most organizations over 80% of all sensitive data resides in non-production environments. Given the shortcomings of synthetic data, as well as those of measures including access control and encryption, data masking has become a de facto standard for securing test data.

Unfortunately, the dynamic nature of the application lifecycle renders standalone-masking solutions impractical for many organizations. Test data is constantly changing and growing, introducing the need for frequent data movement and updates across environments. As a result, organizations largely avoid employing rigorous masking procedures in an effort to avoid the associated overhead.

Test Data Management Redefined: A Model for Testing Solutions that Accelerates Application Projects

The prevailing test data management approaches attempt to ameliorate the time and expense burden of creating full copies of production data. However, the alternatives offer flawed solutions that fail across one or more key dimensions. Shared environments offer the potential for concurrent access, but collisions between testers erode data quality and consistency. Synthetic and subsetted data creation requires time and effort that slows delivery and yields non-representative or incomplete datasets.



Any effective test data management solution needs to deliver high-quality test environments, as well as have the ability to deliver those environments with speed and efficiency. In addition, they must accomplish these goals while protecting sensitive data.

Data Quality: Better Test Data, Better Applications

QA efforts depend on how closely the testing environment reproduces the production environment. Reproducing the production environment, in turn, demands a solution that delivers full copies of data instead of subsets.

In addition, testers require data that is temporally relevant. This often means test data needs to be “fresh,” but it can also mean that data needs to be set to a specific point in time. For example, data for integration testing must be sourced across multiple repositories and synchronized to the same instant.

To maximize the productivity of individual test cycles, test data management solutions should have the ability to:

- Non-disruptively deliver fresh data sourced from operational production systems
- Provision data to a specific point in time
- Deliver complete, real data that allows for thorough testing of edge and corner cases

Data Speed: Shift Testing to the Left

Equally important as the quality of test data is the speed at which it can be delivered. Fast data delivery allows for concentrated testing cycles earlier in the SDLC. By shifting testing to the left, teams can identify bugs when they are less expensive to remedy. It also eliminates future rework, which accounts for 20% of software development budgets for most organizations.[3]

Ideally, test data management solutions should have the ability to:

- Eliminate bottlenecks in the data delivery process
- Deliver test data in minutes instead of hours, days, or weeks
- Give developers and testers self-service control over data sourcing and sharing

Data Security: Mask + Deliver Data

Finally, test data needs to be securely managed and delivered. Any mechanism used to secure data must be operationally simple enough to ensure that data in volatile environments is continuously protected. Test data management solutions must:

- Dynamically secure data as it changes
- Reduce the surface area of risk
- Eliminate privileged access to sensitive data

Better Test Data Management with Data as a Service: Bring Quality, Speed, and Security to Test Data

Technologies in the emerging category of Data as a Service (DaaS) promise to deliver against key requirements for test data management. DaaS platforms bring the benefits of virtualization to application data by:

- Capturing data—including ongoing changes—in production systems
- Versioning and managing data across the full application lifecycle
- Delivering data to non-production systems, including test environments

With intelligent block sharing and advanced compression, DaaS platforms can provision complete, high-quality datasets to multiple test environments while reducing infrastructure requirements. DaaS also brings a new level of agility to test data management by providing access to data in minutes, instead of days or weeks. Moreover, DaaS allows application teams to treat databases like codebases. Since DaaS systems capture and version production data over a window of time, testers can:

- Bookmark and share test data with teammates
- Refresh test data from the latest version production
- Reset test data back to prior point in time
- Branch data for performance and A/B testing

Application teams already have these capabilities for managing source code through the test-fix-test cycle. With DaaS, they can leverage the same capabilities for managing the data that the code talks to.

While test data management will always present organizations with vexing challenges, DaaS represents an enormous opportunity to allow testers to wrestle back control over data. And with a quickly growing number of adopters, technology integrations, and mature best practices, the value of DaaS-based tools will only increase. Organizations will soon be faced with the decision to continue using legacy tools, or transition to a new breed of DaaS solutions that transform testing data management.

1 Infosys Ltd., Test Data Management: Enabling reliable testing through realistic test data
2 Cognizant, Transforming Test Data Management for Increased Business Value
3 IDG Enterprise, *Top CIO Challenges that Contribute to Enterprise Application Failure*

→ **To learn more about Delphix, or to try our free download, visit our website at www.delphix.com.
For any other questions, please contact us at info@delphix.com.**

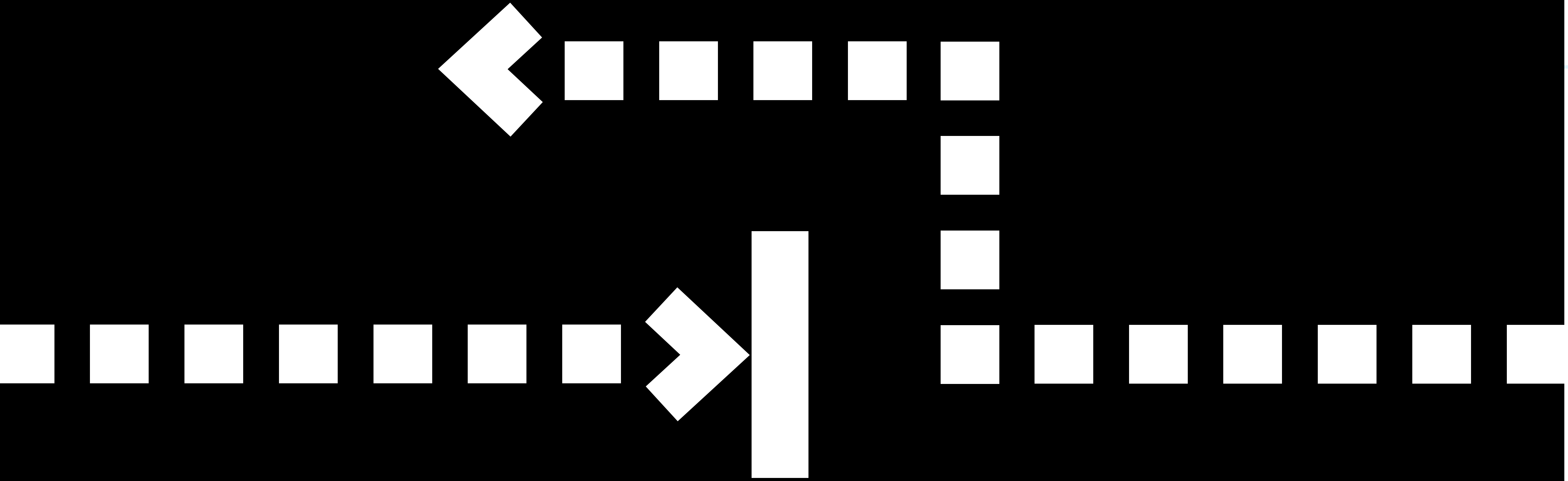
DOES14 - Elisabeth Hendrickson - Pivotal - On the Care and Feeding of Feedback Cycles
DevOps Enterprise Summit
#DOES14
DevOps Enterprise Summit
Presented by: Electric Cloud
IT Revolution Press
www.devopsenterprise.io
0:00 / 31:56

On the Care and Feeding of Feedback Cycles
presentation by Elisabeth Hendrickson
at DevOps Enterprise 2014

Nothing interrupts the continuous flow of value like bad surprises that require immediate attention: major defects, service outages, support escalations, or even scrapping just-completed capabilities that don't actually meet business needs.

You already know that the sooner you can discover a problem, the sooner and more smoothly you can remedy it. Agile practices involve testing early and often. However, feedback comes in many forms, only some of which are traditionally considered testing. Continuous integration, acceptance testing with users, and even cohort analysis to validate business hypotheses are all examples of feedback cycles.

This talk examines the many forms of feedback, the questions each can answer, and the risks each can mitigate. We'll take a fresh look at the churn and disruption created by having high feedback latency, when the time between taking an action and discovering its effect is too long. We'll also consider how addressing "bugs" that may not be detracting from the actual business value can distract us from addressing real risks. Along the way we'll consider fundamental principles that you can apply immediately to keep your feedback cycles healthy and happy.



The Third Way: Culture Experimentation and Mastery

From Agile to DevOps at Microsoft Developer Division

Microsoft

This is an excerpt from the ebook *Our Journey to Cloud Cadence, Lessons Learned at Microsoft Developer Division* by Sam Guckenheimer. The broader book tells how a "box" software company, delivering on-premises software releases on a multiyear cadence, became an SaaS provider as well, with Continuous Delivery from the public cloud. It covers the DevOps engineering practices and tools that the organization needed to evolve and the lessons learned.

How We Moved from Agile to DevOps

Over seven years, Microsoft Developer Division (DevDiv) embraced Agile. We had achieved a 15x reduction in technical debt through solid engineering practices, drawn heavily from XP. We trained everyone on Scrum, multidisciplinary teams, and product ownership across the division. We significantly focused on the flow of value to our customers. By the time we shipped Visual Studio 2010, the product line achieved a level of customer recognition that was unparalleled.

After we shipped VS2010, we knew that we needed to begin to work on converting Team Foundation Server into a Software as a Service (SaaS) offering. The SaaS version, now called Visual Studio Online (VSO), would be hosted on Microsoft Azure, and to succeed with that we needed to begin adopting DevOps practices. That meant we needed to expand our practices from Agile to DevOps. What's the difference?

Part of a DevOps culture is learning from usage. A tacit assumption of Agile was that the Product Owner was omniscient and could groom the backlog correctly. In contrast, when you run a high-reliability service, you can observe how customers are actually using its capabilities in near real-time. You can release frequently, experiment with improvements, measure, and ask customers how they perceive the changes. The data you collect becomes the basis for the next set of improvements you do. In this way, a DevOps product backlog is really a set of hypotheses that become experiments in the running software and allow a cycle of continuous feedback.

As shown in Figure 1, DevOps grew from Agile based on four trends:

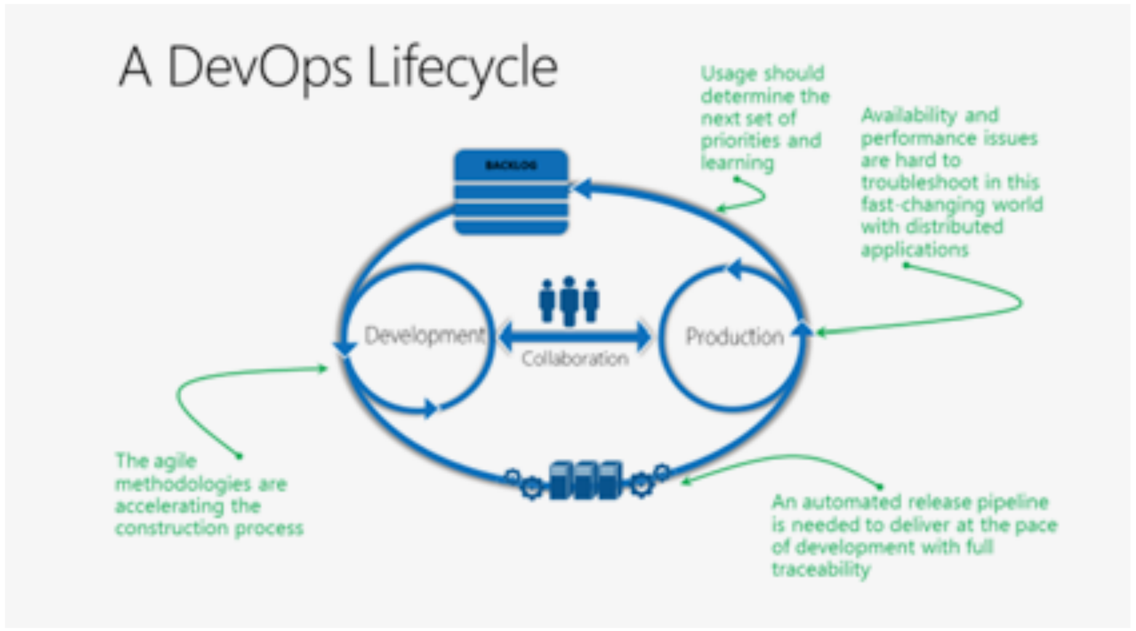

A DevOps Lifecycle
Usage should determine the next set of priorities and learning
Availability and performance issues are hard to troubleshoot in this fast-changing world with distributed applications
BACKLOG
Development
Collaboration
Production
The agile methodologies are accelerating the construction process
An automated release pipeline is needed to deliver at the pace of development with full traceability

Figure 1

Unlike many “born-in–the-cloud” companies, we did not start with a SaaS offering. Most of our customers are using the on-premises version of our software (Team Foundation Server, originally released in 2005 and now available in Version 2015). When we started VSO, we determined that would we maintain a single code base for both the SaaS and “box” versions of our product, developing cloud-first. When an engineer pushes code, it triggers a continuous integration pipeline. At the end of every three-weekly sprint, we release to the cloud, and after 4–5 sprints, we release a quarterly update for the on-premises product, as shown in Figure 2.

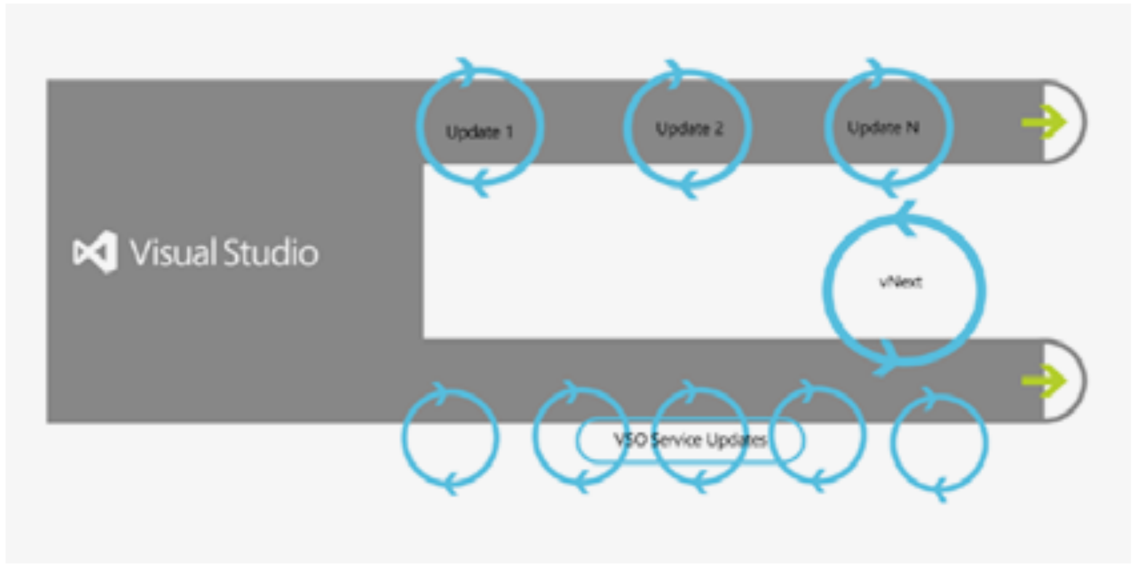


Figure 2

Exposure Control

When you are working on a service, you have the blessing of frequent releases, in our case at the end of every three-weekly sprint. This creates a great opportunity to expose work and a need to control when it is exposed. Some of the issues that arise are:

- How do you work on features that span sprints?
- How do you experiment with features in order to get usage and feedback when you know they are likely to change?
- How do you do “dark launches” that introduce services or capabilities before you are ready to expose or market them?

In all of these cases, we have started to use the feature flag pattern. A feature flag is a mechanism to control *production exposure* of any feature to any user or group of users. As a team working on the new feature, you can register a flag with the feature flag service, and it will default down. When you are ready to have someone try your work, you can raise the flag for that identity in production as long as you need. If you want to modify the feature, you can lower the flag with no redeployment and the feature is no longer exposed.

By allowing progressive exposure control, feature flags also provide one form of testing in production. We will typically expose new capabilities initially to ourselves, then to our early adopters, and then to increasingly larger circles of customers. Monitoring the performance and usage allows us to ensure that there is no issue at scale in the new service components.

Code Velocity and Branching

When we first moved to Agile in 2008, we believed that we would enforce code quality with the right quality gates and branching structure. In the early days, developers worked in a fairly elaborate branch structure and could only promote code that satisfied a stringent definition of done, including a gated check-in that effectively did a “get latest” from the trunk and built the system with the new changesets and ran the build policies.

The unforeseen consequence of that branch structure was many days—sometimes months—of impedance in the flow of code from the leaf nodes to the trunk, and long periods of code sitting in branches unmerged. This created significant merge debt. When work was ready to merge, the trunk had moved considerably, and merge conflicts abounded, leading to a long reconciliation process and lots of waste.

The first step we made, by 2010, was to significantly flatten the branch structure so that there are now very few branches, and they are usually quite temporary. We created an explicit goal to optimize code flow: in other words, to minimize the time between a check-in and that changeset becoming available to every other dev working.

The next step was to move to distributed version control, using Git, which is now supported under VSO and TFS. Most of our customers and colleagues continue to use centralized version control and VSO and TFS support both models. Git has the advantage of allowing very lightweight, temporary branches. A topic branch might be created when deork item, and cleaned up when the changes are merged into the mainline.

All the code lives in Master (the trunk) when committed, and the pull-request workflow combined both code review and the policy gates. This makes merging continuous, easy, and in tiny batches, while the code is fresh in everyone's mind.

This process isolates the developers' work for the short period it is separate and then integrates it continuously. The branches have no bookkeeping overhead and shrivel when they are no longer needed.

We created an explicit goal to optimize code flow: in other words, to minimize the time between a check-in and that changeset becoming available to every other dev working.

Agile on Steroids

We continue to follow Scrum, but stripped to its essentials for easy communication and scaling across geographies. For example, the primary work unit is a feature crew, equivalent to a Scrum team, with the product owner sitting inside the team and participating day in, day out. The Product Owner and Engineering Lead jointly speak for the team.

We apply the principle of team autonomy and organizational alignment. There is a certain chemistry that emerges in a feature crew. Teams are multidisciplinary, and we have collapsed the career ladders for developers and testers into a common engineering role. (This has been a big morale booster.)

Teams are cohesive. There are 8–12 engineers in a crew. We try to keep them together for 12–18 months minimum, and many stay together much longer. If there is a question of balancing work, we will ask a second crew to pull more backlog items, rather than try to move engineers across crews.

The feature crew sits together in a team room. This is not part of a large open space, but a dedicated room where people who are working on a common area share space and can talk freely. Around the team room are small focus rooms for breakouts, but free-form conversations happen in the team room. When it's time for a daily standup, everyone stands up.

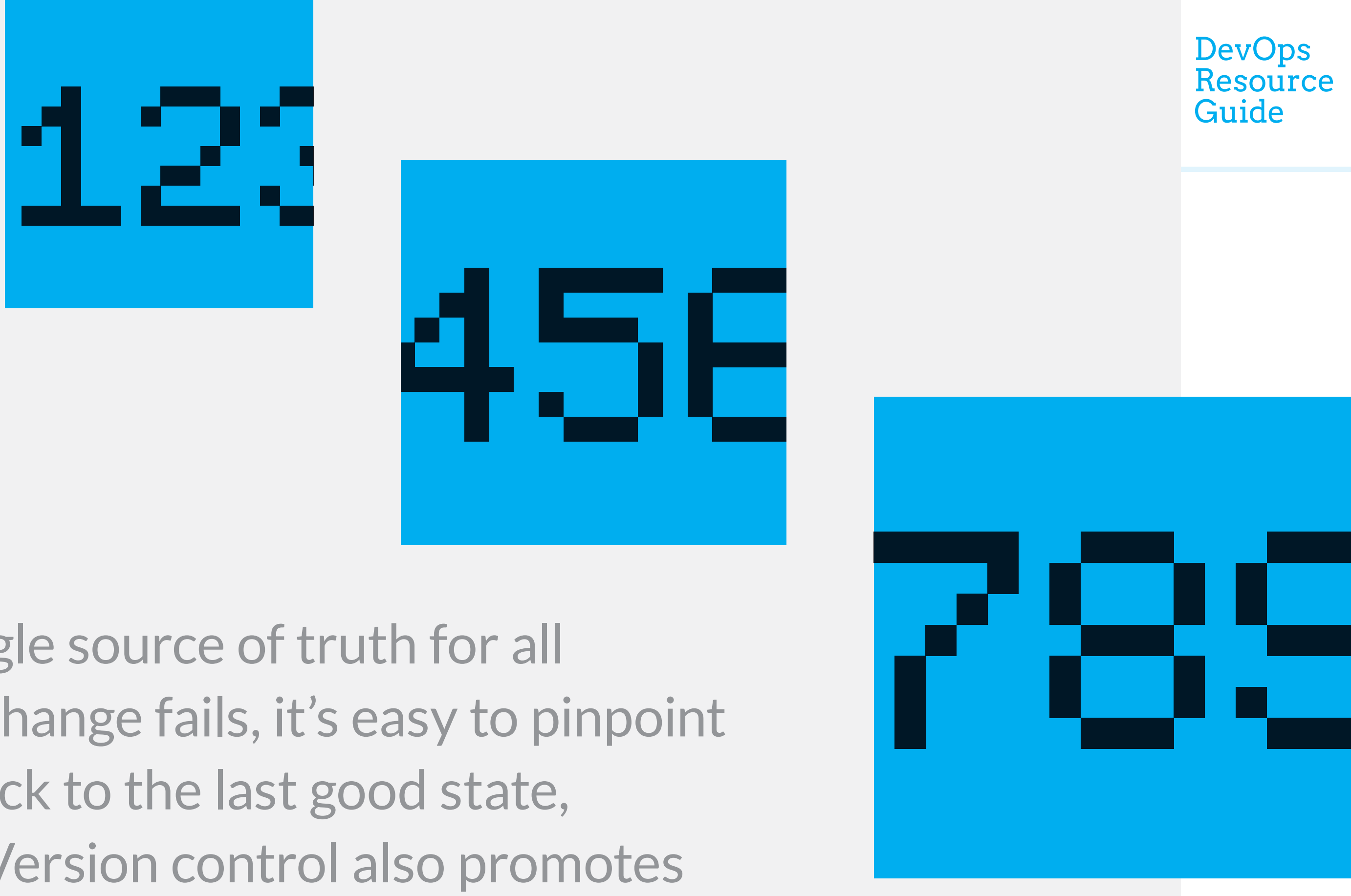
We have settled on three-weekly sprints, empirically. It's proven difficult for us to deliver enough value in shorter periods and coordinate across worldwide sites. Longer periods have left too much work dark. Some groups in Microsoft use two-weekly sprints, while others effectively flight experiments of new work many times per day without the time boxing of sprints at all.

The definition of done is conceptually very simple. You build it, you run it. Your code will be deployed live to millions of users at the end of the sprint, and if there are live-site issues, you (and everyone else) will know immediately. You will remediate to root cause.

We rotate the role of Scrum master, so that everyone experiences the responsibility of running the Scrum. We keep the sprint ceremonies as lightweight as possible for outward communication. At the beginning of the sprint, the crew pulls its work from the product backlog and communicates its sprint plan in a one page email, hyperlinked to the product backlog items in VSO. The sprint review is distributed as an update to this mail, with a roughly three-minute video added. The video shows a demo, in customer terms, of what you can now do as a result of the team's accomplishments in the sprint.

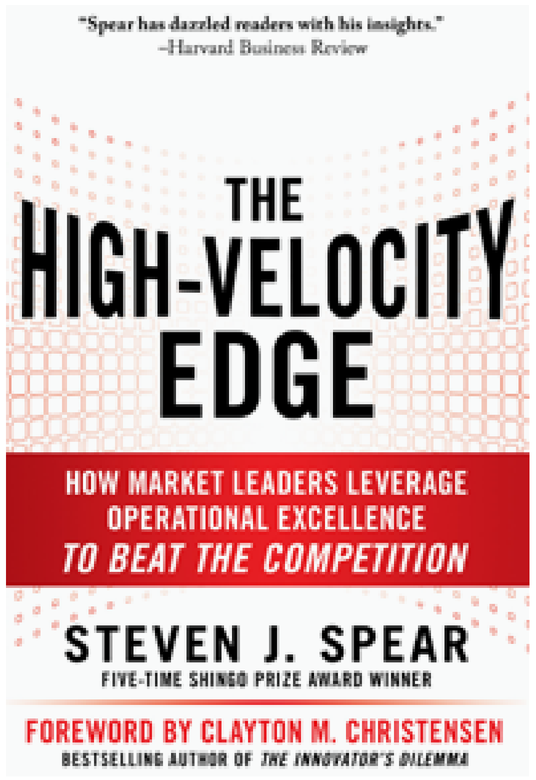
We also keep planning lightweight. We will generally work toward an 18-month vision, which acts as a true north. This may be captured in some technology spikes, storyboards, conceptual videos, and brief documents. Every six months we think of as a season, "spring" or "fall," and during this period we'll be firmer about commitments and dependencies across teams. Every three sprints, the feature crew leads will get together for a "feature chat," in which they share their intended stack ranks, check for course correction, ask for news, and synchronize against other teams.

→ Download the full ebook, *Our Journey to Cloud Cadence, Lessons Learned at Microsoft Developer Division.*



Version Control for *All* Production Artifacts

Version control provides a single source of truth for all changes. That means when a change fails, it's easy to pinpoint the cause of failure and roll back to the last good state, reducing the time to recover. Version control also promotes greater collaboration between teams. The benefits of version control shouldn't be limited to application code; in fact, our analysis shows that organizations using version control for both system and application configurations have higher IT performance.


"Spear has dazzled readers with his insights." –Harvard Business Review
THE HIGH-VELOCITY EDGE
HOW MARKET LEADERS LEVERAGE OPERATIONAL EXCELLENCE TO BEAT THE COMPETITION
STEVEN J. SPEAR
FIVE-TIME SHINGO PRIZE AWARD WINNER
FOREWORD BY CLAYTON M. CHRISTENSEN
BESTSELLING AUTHOR OF THE INNOVATOR'S DILEMMA

The High-Velocity Edge: How Market Leaders Leverage Operational Excellence to Beat the Competition

Dr. Steven Spear

McGraw-Hill, 2010

Order a copy.

Decades of research have shown us that failure is inherent in complex systems. When failures occur, we all too often mistakenly attribute the root cause as "human error." This is an especially unhelpful conclusion when we have created a system that is too complex to understand, let alone operate and safely and repeatedly change.

Complex systems are not just the domain of the largest computing systems, such as Google and Amazon. If one of the primary attributes of a complex system is that it defies any single person's ability to see the whole system and understand how all the pieces fit together, then virtually any significant technology-enabled service we build qualifies as such.

To mitigate complexity, our goal is to create a safe system of work, where we can operate and make changes to our applications and environments without constant fear that small changes will have catastrophic failures.

Furthermore, when something goes wrong, we must be capable of detecting and correcting the problem early, ideally long before a customer is impacted.

While designing perfectly safe systems is likely beyond our abilities, Dr. Steven Spear (credited for "decoding the Toyota Production System" as part of his doctoral thesis at Harvard Business School) has shown that "safe systems are close to achievable when (a) complex work is managed so that problems in design are revealed, (b) problems that are seen are solved so that new knowledge is built quickly, and (c) the new knowledge, although discovered locally, is shared throughout the organization."

In *The High-Velocity Edge*, Dr. Spear's model describes the causal mechanism that explains the long-lasting success of the Toyota Production System, Alcoa, and many others. Among them is the US Navy's Nuclear

Power Propulsion Program, which has provided over 5,700 reactor-years of operation without a single reactor-related casualty or escape of radiation. They achieved this by integrating into their daily work of design and operations a need to continually test their assumptions of reality, enabling a culture of learning, and turning their learnings into systemic improvements that prevent future failures. Spears writes,

> Whatever knowledge the group had, it was assumed to be inadequate. There was no room for guessing; learning had to be constant and fast, not only experiential but experimental. They made explicit their best understanding and expectation of what actions would lead to what outcomes, which created constant opportunities to learn and improve.
>
> Their intense commitment to scripted procedures, incident reports about even seemingly minor departures from or failures of procedure, and the rapid update of procedures and of system designs enable a young crew and their officers setting out for their first cruise to have over 5,700 reactor-years of experience underpinning their individual expertise.

Not surprisingly, these principles and behaviors can be seen in all high-performing DevOps organizations, as well.

–**GENE KIM**

Continuous Discussions (#c9d9)

Electric Cloud

Electric Cloud is the leader in enterprise Continuous Delivery and DevOps automation, helping organizations deliver better software faster by automating and accelerating build, test and deployment processes at scale. Industry leaders like Qualcomm, SpaceX, Cisco, GE, Gap and E*TRADE use Electric Cloud's solutions to boost DevOps productivity and Agile throughput.

Powered by the DevOps community

"DevOps and Continuous Delivery are as much about people and process as technology. We think connecting with peers and sharing experiences, lessons learned, tips and best practices is the best way to accelerate our mutual success."

–ANDERS WALLGREN, CTO AT ELECTRIC CLOUD

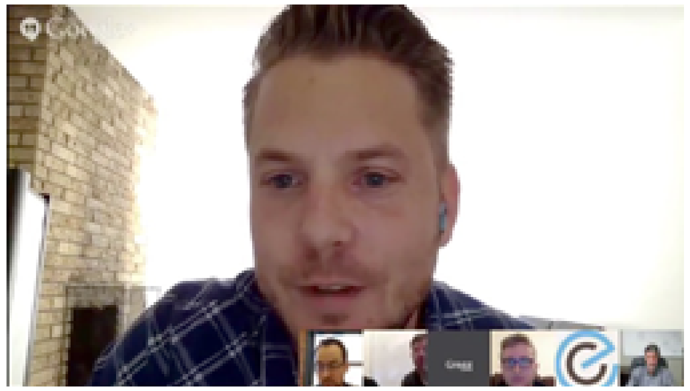
DevOps is a team sport. It's all about collaboration: bringing people and teams together to ask interesting questions, experiment, and continuously improve.

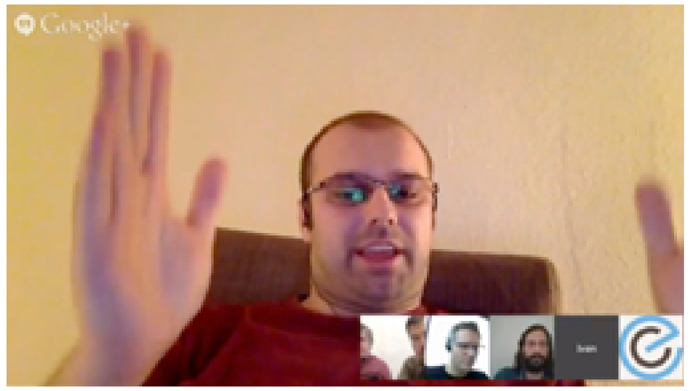
It is in this spirit of collaboration that Electric Cloud hosts "Continuous Discussions" (#c9d9), **an open forum to discuss Agile, DevOps and Continuous Delivery.** Each episode focuses on a different software delivery use case and features DevOps practitioners, who join us as panelists over Hangout, to discuss their views.

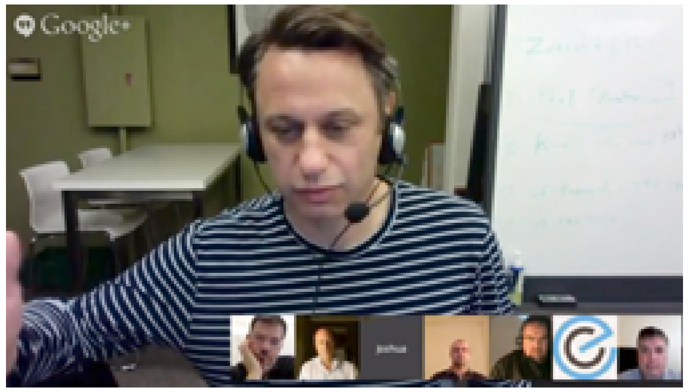
c9d9 exposes DevOps as it exists in the real world, with panelists who are passionate about what they do and eager to share what they know. Anyone can attend or sign up as a panelist to discuss the ins and outs of delivering better software, faster—not just as a slogan, but as a daily practice.

Check out some of the recent episodes:

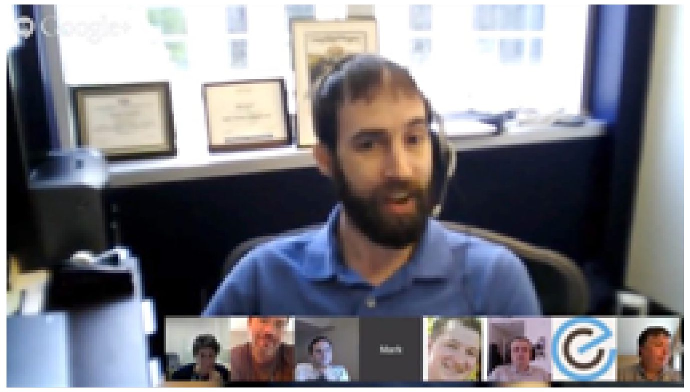
Deployment Automation


Continuous Testing and Test Acceleration

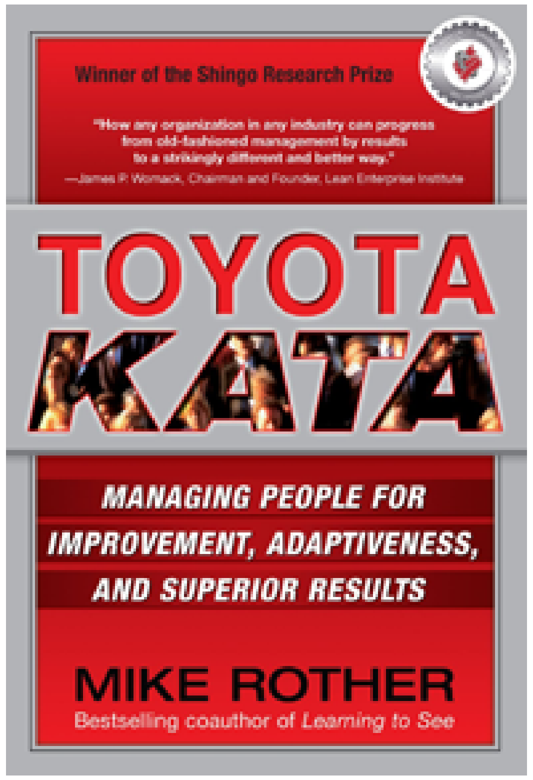

CI Best Practices


Continuous Delivery

DevOps and Lean in Legacy Environments


Consistent Deployments Across Dev, Test and Prod

→ **To watch previous episodes of Continuous Discussions (#c9d9) and to join the next live episode, visit electric-cloud/c9d9. Be sure to chime-in on Twitter using #c9d9.**



Toyota Kata: Managing People for Improvement, Adaptiveness, and Superior Results

Mike Rother

McGraw-Hill, 2009

Order a copy.

Twenty years ago, Mike Rother visited Toyota plants with a team of researchers and American car manufacturing executives. He observed and codified the Toyota practices that led to their extraordinary and market-leading performance. These processes and culture must exist to enable the Lean Plan, Do, Check, Act (PDCA) cycle.

The most obvious manifestation of the Toyota Kata is the two-week improvement cycle, in which every work center supervisor must improve something (anything!) every two weeks. To quote Mr. Rother, "The practice of kata is the act of practicing a pattern so it becomes second nature. In its day-to-day management, Toyota teaches a way of working—a kata—that has helped make it so successful over the last six decades."

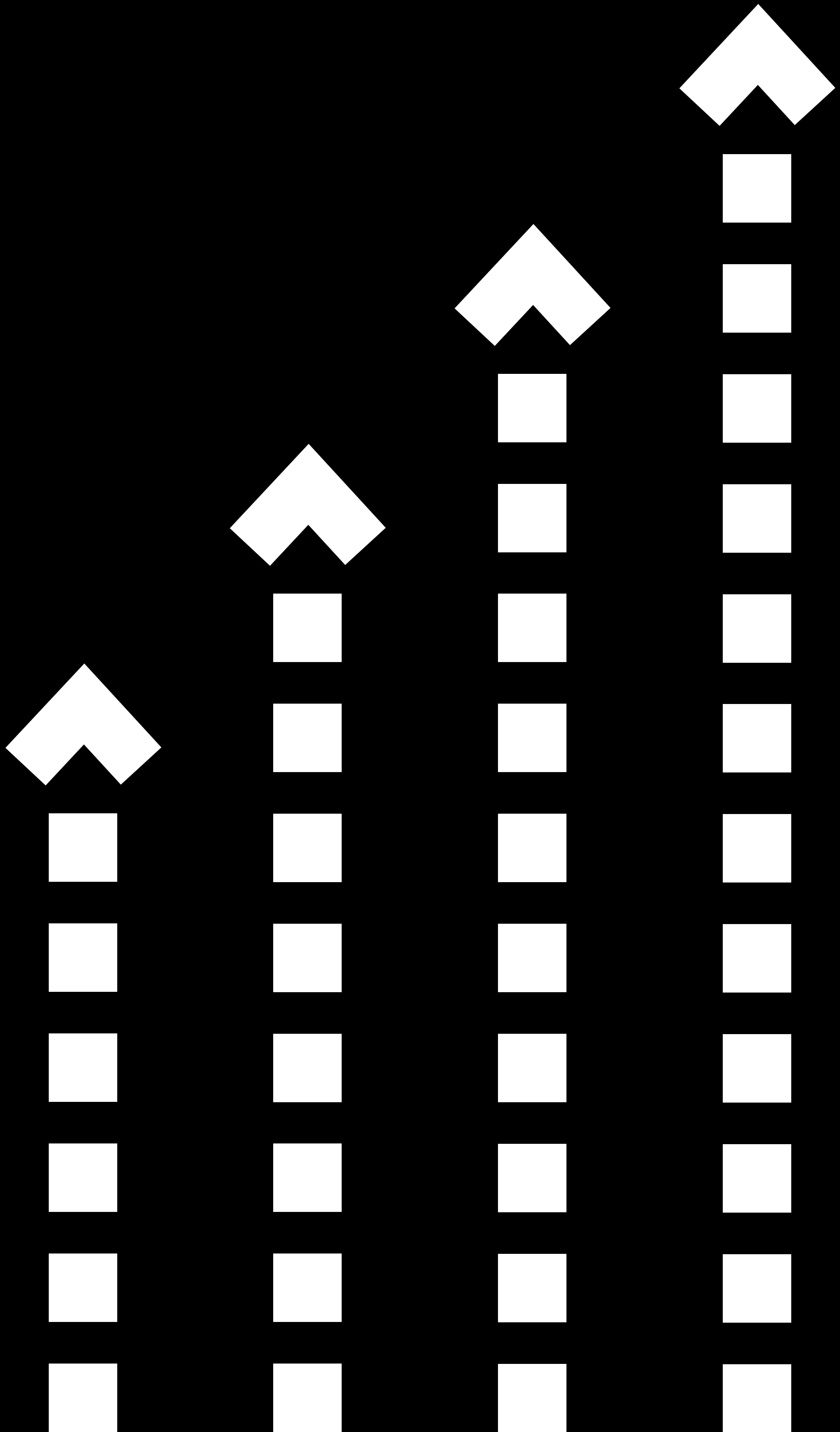
These two-week improvement cycles put constant pressure into the system, forcing it to improve by providing a systematic, scientific routine that can be applied to any problem or challenge, commonizing how the members of an organization develop solutions, migrating managers toward a role of coach and mentor by having them practice coaching cycles, and framing PDCA in a way that has people taking small steps every day.

The need for daily repetition to create habits and change outcomes is well established in the domains of sports, music, the military, and now modern manufacturing. This forms the basis of Erik's Third Way in *The Phoenix Project*. He explains, "It's about how to create a culture that simultaneously fosters experimentation, learning from failure, and understanding that repetition and practice are the prerequisites to mastery."

In *The Phoenix Project*, Patty's ITIL/ITSM crusade is very much like Lean practitioners who were never able to replicate the performance of Toyota. Why? They'd do a Lean Kaizen event once per year but then get marginalized from daily operations the remainder of the year. Mr. Rother asserts that if a system is not improving, the result is not a steady state, but instead, because of entropy, organizational performance declines.

–GENE KIM

Growth and Change



DOES14 - Mark Schwartz - U.S. Citizenship and Immigration Services
IT REV
PRESS
0:00 / 28:59

How DevOps Can Fix Federal Government
presentation by Mark Schwartz
at DevOps Enterprise Summit 2014

The federal government spends more than $80 billion each year on information technology. As the fiasco with healthcare.gov demonstrates, the results are not always good. Government IT programs are expensive and monolithic, and the lead time from a "mission need" to a deployed capability is often measured in years (in one of our agency's programs, about 12 years!). IT systems are often difficult to use, and the US government's online service offerings to citizens are far from meeting the expectations of a public that is used to Google, Facebook, and Twitter.

The US government has only recently begun to adopt Agile approaches, and only in a few agencies. But the results have been encouraging, and they show that it is possible for the bureaucracy to be Agile. DevOps, however is a game changer. At USCIS we have moved to a continuous integration and Continuous Delivery approach, and we have begun experimenting with a DevOps model tailored to the needs of the government.

By combining DevOps with some ideas taken from the Lean Startup movement, I believe we can cause a radical change in how the government does IT. We can dramatically reduce lead times and costs, improve the usability of systems, provide more transparency, create citizen-centric online services, and—importantly—significantly improve the government's security posture.

The Secret to Scaling DevOps

In this excerpt from "The Secret to Scaling DevOps," we share actionable steps to successfully implement and scale DevOps in the enterprise. The secret is to lead with Agile operations and let DevOps happen naturally. Rally provides software and services that drive agility. Learn more.

One of the top indicators of IT performance is a "high-trust" culture, and in practice, that culture is almost impossible to distinguish from an Agile culture. DevOps is really just an example of Agile culture and process: getting previously siloed organizations collaborating, side-by-side, working together towards the common goals of delivering customer value faster and getting feedback at regular intervals.

When you hear about large companies succeeding with DevOps, it's often because they've implemented continuous integration and deployment in a new line of business or a progressive IT project. But what does it look like to implement DevOps in a more traditional operations organization? What does it look like to introduce DevOps into organizations that aren't high-trust environments? It looks like, and is, Agile.

Leaders and change agents who've successfully tackled both DevOps and Agile revolutions within large enterprises can tell courageous stories about embracing disruption inside their organizations. However, the fundamental challenge for DevOps champions is that they too often focus on software solutions for what are inherently cultural problems: lack of trust, collaboration, and common practices.

Through hundreds of large-scale Agile development and IT transformations in this kind of organization, we've found that implementing and scaling DevOps requires not only culture change but end-to-end adoption of Agile methodologies. Leading with the tooling, as so many companies do when they try to do DevOps, leaves out the benefits that silo-busting agility delivers. Scalable DevOps begins with Agile, not the other way around.

Architecting the (Business) System

Businesses must take a systems approach to how their engineering and operations teams work together. It requires viewing the entire technology organization as a single Agile delivery team, with shared success measures and outcomes based on the value that's delivered to customers. This takes effort and time, but by breaking the work down into smaller, actionable steps you can get going immediately:

1. Create an Agile development/engineering/IT organization
2. Create an Agile operations organization
3. Make sure a feature can flow from one organization to the other seamlessly

Agile Engineering & IT

After more than a decade of scaling Agile development in software engineering and IT, we know what works. We've found that when your engineering and IT teams are collaborating to deliver smaller batches of work more frequently, you've set the stage for operations to follow suit. When leadership teams are acting as servant leaders, trusting their teams to raise issues and risks, prioritizing work based on customer value, and continuously improving, you have a high-trust culture.

The business benefits of Agile in software engineering alone are extraordinary. Cut time to market from 12–18 months to three. Cut defects in half. Raise customer satisfaction. All of this is necessary for being successful with Agile operations and DevOps.

Agile Operations

Creating an Agile operations group presents many of the same challenges as creating an Agile development or IT organization, but it is equally necessary for successful DevOps. Here are some common objections to overcome:

- Agile practices and culture viewed as "not possible in this organization"
- Systems engineering seen as a lesser skill / expertise set (note: QA and testing suffered the same perception before Agile)

Businesses must take a systems approach to how their engineering and operations teams work together. It requires viewing the entire technology organization as a single Agile delivery team, with shared success measures and outcomes based on the value that's delivered to customers.

Operations teams may also feel like their core values are at risk: after all, you've chartered these teams with preserving the stability and reliability of customer-facing systems. It can seem counter-intuitive that delivering more frequently reduces risk and increases quality; until quality and performance become personal and team responsibilities, it's hard to imagine this change.

Every operations group struggles with balancing planned and unplanned work, and interrupts are the mortal enemy of having reliable velocity. Agile operations helps to buffer unplanned work as much as possible, allowing some part of the group to stay focused on delivering planned customer value without risking stability to everyday operations.

Ops teams with focused, planned work can successfully adopt engineering best practices like using version control, configuration management tools, test automation, pairing, and integrated testing.

In an Agile operations organization, you begin seeing steel thread or minimum viable versions of infrastructure that avoid the all-or-nothing waterfall approach that was previously the norm. And, you see empathy and understanding flourish between systems engineers and software engineers as their worlds begin looking quite a bit alike.

Seamless Flow of a Feature

At this point in your journey, DevOps, as many envision is largely done. But this last step is the crux of scaling DevOps. The seamless flow of a feature refers to much more than code delivered through a traditional CI/CD (continuous integration / continuous deployment) pipeline. A feature in an Agile operations group could include infrastructure, disaster recovery, compliance, analytics, and many other traditional IT core competencies.

The way the work is prioritized, iterated on, and implemented is exactly the same, regardless of the value being delivered—this is the value of having Agile underpinnings throughout the delivery organization. Without it, DevOps tends to too narrowly define what can and can't be delivered seamlessly since it relies on tooling scoped to specific tasks.

Results

If you take these three steps, how are your people? Thinking Lean, behaving with agility. Feeling valued, empowered, and trusted. Communicating and collaborating.

Your processes? Agile across the whole technology delivery group—delivering highest customer value first, in small batches, at frequent intervals, with regular customer feedback.

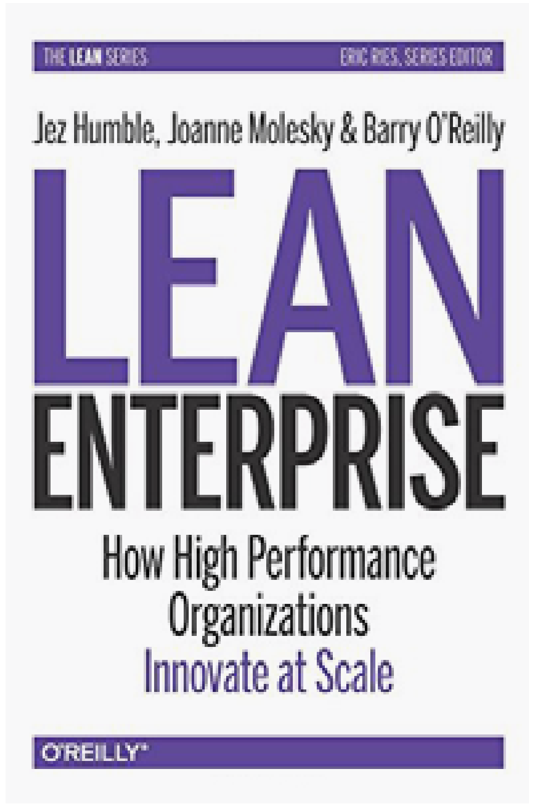
Technology and tools? Supporting the processes and people. If Agile lifecycle management software is used to track all of the work, and teams share a common chat/collaboration tool, then you're able to send any feature seamlessly between the two groups, now one Agile delivery organization.

The business value delivered to your customers? You're delivering high-quality value with fewer defects and higher customer satisfaction than ever before. You're delivering at speeds you never thought possible for a large enterprise.

That's what it means to scale DevOps.

→ **Attend RallyON! for more information on "The Secret to Scaling DevOps," and to learn from your peers (who are doing this every day). For more information about how Rally can help you scale DevOps visit www.rallydev.com/devops.**

Amazon's Approach to Growth


THE LEAN SERIES
ERIC RIES, SERIES EDITOR
Jez Humble, Joanne Molesky & Barry O'Reilly
LEAN ENTERPRISE
How High Performance Organizations Innovate at Scale
O'REILLY

Excerpt from
Lean Enterprise: How High Performance Organizations Innovate at Scale

Jez Humble, Joanne Molesky, Barry O'Reilly

O'Reilly Media, 2015

In 2001, Amazon had a problem: the huge, monolithic "big ball of mud" that ran their website, a system called Obidos, was unable to scale. The limiting factor was the databases. CEO Jeff Bezos turned this problem into an opportunity. He wanted Amazon to become a platform that other businesses could leverage, with the ultimate goal of better meeting customer needs. With this in mind, he sent a memo to technical staff directing them to create a service-oriented architecture, which Steve Yegge summarizes thus:

1. All teams will henceforth expose their data and functionality through service interfaces.
2. Teams must communicate with each other through these interfaces.
3. There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.
4. It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols—doesn't matter. Bezos doesn't care.

5. All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.
6. Anyone who doesn't do this will be fired.

Bezos hired West Point Academy graduate and ex-Army Ranger Rick Dalzell to enforce these rules. Bezos mandated another important change along with these rules: each service would be owned by a cross-functional team that would build and run the service throughout its lifecycle. As Werner Vogels, CTO of Amazon, says, "You build it, you run it." This, along with the rule that all service interfaces are designed to be externalizable, has some important consequences. As Vogels points out, this way of organizing teams "brings developers into contact with the day-to-day operation of their software. It also brings them into day-to-day contact with the customer. This customer feedback loop is essential for improving the quality of the service."

Each team is thus effectively engaged in product development—even the people working on the infrastructural components that comprise Amazon Web Services, such as EC2. It's hard to overemphasize the importance of this transition from a project-based funding and delivery paradigm to one based on product development.

One of the biggest problems as organizations grow is maintaining effective communication between people and between teams. Once you move people to a different floor, a different building, or a different timezone, communication bandwidth becomes drastically limited and it becomes very hard to maintain shared understanding, trust, and effective collaboration. To control this problem, Amazon stipulated that all teams must conform to the "two pizza" rule: they should be small enough that two pizzas can feed the whole team—usually about 5 to 10 people.

This limit on size has four important effects:

1. It ensures the team has a clear, shared understanding of the system they are working on. As teams get larger, the amount of communication required for everybody to know what's going on scales in a combinatorial fashion.
2. It limits the growth rate of the product or service being worked on. By limiting the size of the team, we limit the rate at which their system can evolve. This also helps to ensure the team maintains a shared understanding of the system.
3. Perhaps most importantly, it decentralizes power and creates autonomy, following the Principle of Mission. Each two-pizza team (2PT) is as autonomous as possible. The team's lead, working with the executive team, would decide upon the key business metric that the team is responsible for, known as the fitness function, that becomes the overall evaluation criteria for the team's experiments. The team is then able to act autonomously to maximize that metric, using the techniques we describe in Chapter 9.
4. Leading a 2PT is a way for employees to gain some leadership experience in an environment where failure does not have catastrophic consequences—which "helped the company attract and retain entrepreneurial talent." An essential element of Amazon's strategy was the link between the organizational structure of a 2PT and the architectural approach of a service-oriented architecture.

To avoid the communication overhead that can kill productivity as we scale software development, Amazon leveraged one of the most important laws of software development—Conway's Law: "Organizations which design systems...are constrained to produce designs which are copies of the communication structures of these organizations." One way to apply Conway's Law is to align API boundaries with team boundaries. In this way we can distribute teams all across the world. So long as we have each service developed and run by a single, co-located, autonomous cross-functional team, rich communication between teams is no longer necessary.

Organizations often try to fight Conway's Law. A common example is splitting teams by function, e.g., by putting engineers and testers in different locations (or, even worse, by outsourcing testers). Another example is when the front end for a product is developed by one team, the business logic by a second, and the database by a third. Since any new feature requires changes to all three, we require a great deal of communication between these teams, which is severely impacted if they are in separate locations. Splitting teams by function or architectural layer typically leads to a great deal of rework, disagreements over specifications, poor handoffs, and people sitting idle waiting for somebody else.

Amazon's approach is certainly not the only way to create velocity at scale, but it illustrates the important connection between communication structures, leadership, and systems architecture.

To avoid the communication overhead that can kill productivity as we scale software development, Amazon leveraged one of the most important laws of software development—Conway's Law: "Organizations which design systems... are constrained to produce designs which are copies of the communication structures of these organizations."

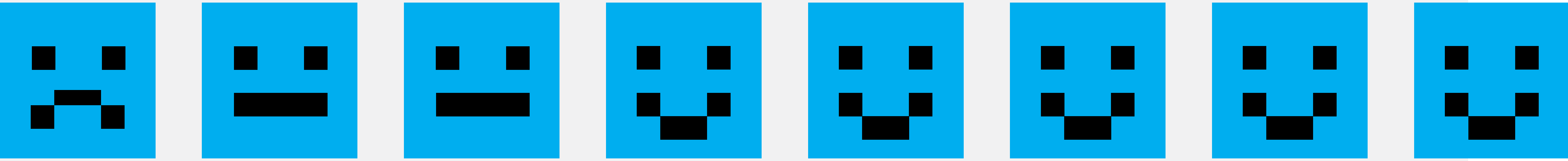
→ Order a copy of *Lean Enterprise*: *How High Performance Organizations Innovate at Scale.*

High-Trust Organizational Culture

One of the pillars of DevOps is culture, and we were pleased to prove what we already knew anecdotally: culture matters. In fact,

organizational culture was highly predictive of both IT performance and overall organizational performance.

No one should be surprised to hear that high-trust cultures lead to greater performance, while bureaucratic and fear-based cultures are destructive to performance.



Learnings:
Practices Where We Gauge Our Excellence

This is an excerpt from the ebook *Our Journey to Cloud Cadence, Lessons Learned at Microsoft Developer Division* by Sam Guckenheimer. As we have moved to DevOps, we have come to assess our growth in seven practice areas, which we collectively think of as the Second Decade of Agile.

Agile scheduling and teams. This is consistent with Agile, but more lightweight. Feature crews are multidisciplinary, pull from a common product-backlog, minimize work in process, and deliver work ready to deploy live at the end of each sprint.

Management of technical debt. Any technical debt you carry is a risk, which will generate unplanned work, such as Live Site Incidents, that will interfere with your intended delivery. We are very careful to be conscious of any debt items and to schedule paying them off before they can interfere with the quality of service we deliver. (Occasionally, we have misjudged, as in the VS 2013 launch story above, but we are always transparent in our communication.)

Flow of value. This means keeping our backlog ranked according to what matters to the customers and focusing on the delivery of value for them. We always spoke of this during the first decade of Agile, but now with DevOps telemetry, we can measure how much we are succeeding and whether we need to correct our course.

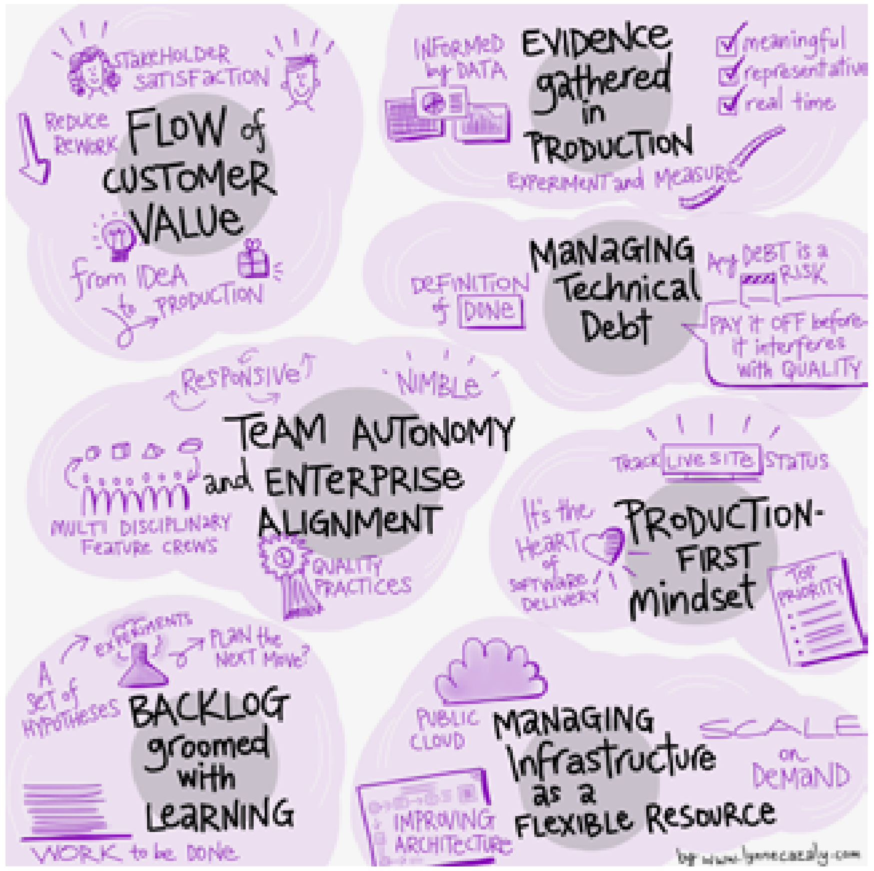


Figure 3

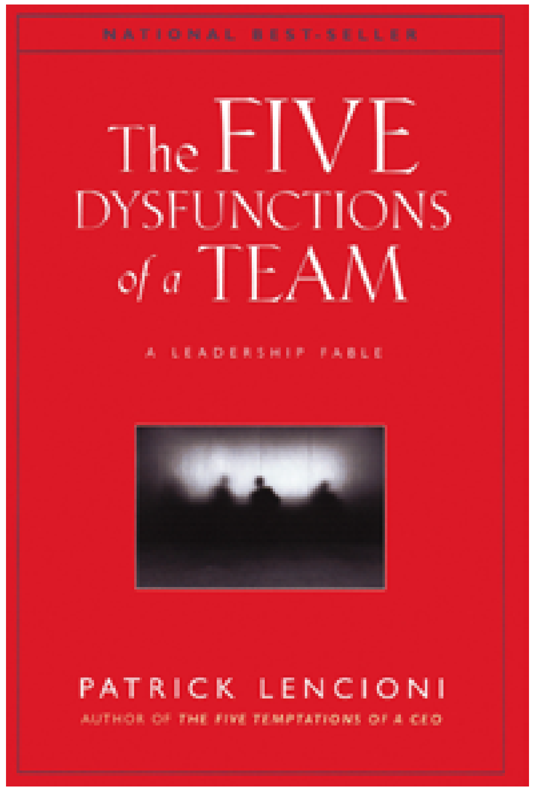
Hypothesis-based backlog. Before DevOps, the product owner groomed the backlog based on the best input from stakeholders. Nowadays, we treat the backlog as a set of hypotheses, which we need to turn into experiments, and for which we need to collect data that supports or diminishes the hypothesis. Based on that evidence we can determine the next move in the backlog and persevere (do more) or pivot (do something different).

Evidence and data. We instrument everything, not just for health, availability, performance, and other qualities of service, but to understand usage and to collect evidence relative to the backlog hypotheses. For example, we will experiment with changes to user experience and measure the impact on conversion rates in the funnel. We will contrast usage data among cohorts, such as weekday and weekend users, to hypothesize ways of improving the experience for each.

Production first mindset. That data is reliable only if the quality of service is consistently excellent. We always track the live site status, remediate any live site issues at root cause, and proactively identify any outliers in performance to see why they are experiencing slowdowns.

Cloud ready. We can only deliver a 24 × 7 × 365 service by continually improving our architecture to refactor into more independent, discrete services and by using the flexible infrastructure of the public cloud. When we need more capacity, the cloud (in our case Azure) provides it. Every new capability we develop cloud-first and then move into our on-premises product, with a few very intentional exceptions, knowing that it has been hardened at scale and that we have received continuous feedback from constant usage.

→ Download the full ebook, *Our Journey to Cloud Cadence, Lessons Learned at Microsoft Developer Division.*


NATIONAL BEST-SELLER
The FIVE DYSFUNCTIONS of a TEAM
A LEADERSHIP FABLE
PATRICK LENCIONI
AUTHOR OF THE FIVE TEMPTATIONS OF A CEO

The Five Dysfunctions of a Team: A Leadership Fable

Patrick Lencioni

Jossey-Bass, 2011

Order a copy.

Patrick Lencioni posits that one of the core contributors to a team's inability to achieve goals is due to lack of trust. In his model, the five dysfunctions are

- Absence of trust—unwilling to be vulnerable within the group
- Fear of conflict—seeking artificial harmony over constructive passionate debate
- Lack of commitment—feigning buy-in for group decisions creates ambiguity throughout the organization
- Avoidance of accountability—ducking the responsibility to call peers on counterproductive behavior, which sets low standards
- Inattention to results—focusing on personal success, status, and ego before team success

With the bitter intertribal warfare that has existed between Development and Operations, as well as between IT and "the business," we very much need the lessons of Mr. Lencioni to achieve the DevOps ideal. Often, the first step in using Mr. Lencioni's methodology is for leaders to become more vulnerable (or, at the very least, start by modeling vulnerable behaviors). In *The Phoenix Project*, Steve has already internalized these practices for decades and leads what is called a personal history exercise.

I was fortunate enough to have personally observed and benefited from Mr. Lencioni's techniques when my former boss, Jim B. Johnson, first joined as CEO of Tripwire, Inc. He shared his own story, which was so personal and touching it left the rest of us on the executive team emotionally raw, with tears in (almost) everyone's eyes.

In turn, we all had to share some elements of our own stories, showing vulnerability to each other and enabling the next step, which is to stop fearing conflict. Jim set the tone of the honesty and candor he demanded from everyone. It changed us, and we started acting more like a team.

This was probably one of the most important lessons in my life. It is now my aspiration in every domain of my life to never fear conflict, telling the truth, or saying what I really think. I'd be delusional to think I can fully achieve this, but it's still a worthy goal.

—GENE KIM

Next

Here are a few more resources to further your DevOps research and discovery. We've gathered a sampling of podcasts, news-letters, reports, videos, and opportunities to connect with the wider DevOps community.

State of DevOps Report

With combined responses of over 18,000, this annual report is the largest and most comprehensive DevOps study to date. You can read the full 2013 and 2014 reports. The 2015 report, a collaboration by Puppet Labs and IT Revolution and sponsored by PWC, is due to be released in summer 2015.

DevOps Cafe

Hosted by Damon Edwards and John Willis, this series of mostly monthly podcasts spans a wide range of DevOps topics—both philosophical and practical. You'll find book reviews, interviews, and how-tos.

DevOps Meetups

Never underestimate the power of the Meetup. This is a thriving community comprising over 1,000 groups and 210,000 members in 64 countries.

The Ship Show

The Ship Show is a twice-monthly podcast, featuring discussion on everything from build engineering to DevOps to release manage-ment, plus interviews, new tools and techniques, and reviews.

DevOps Enterprise Summit on YouTube

This YouTube channel has 30+ recordings of presenta-tions from the inaugural DevOps Enterprise Summit, held October 21–23, 2014.

DevOps Weekly

Described as "a weekly slice of DevOps news," this newsletter by Gareth Rushgrove has approximately 15,000 subscribers and an issue archive extending back to 2010.

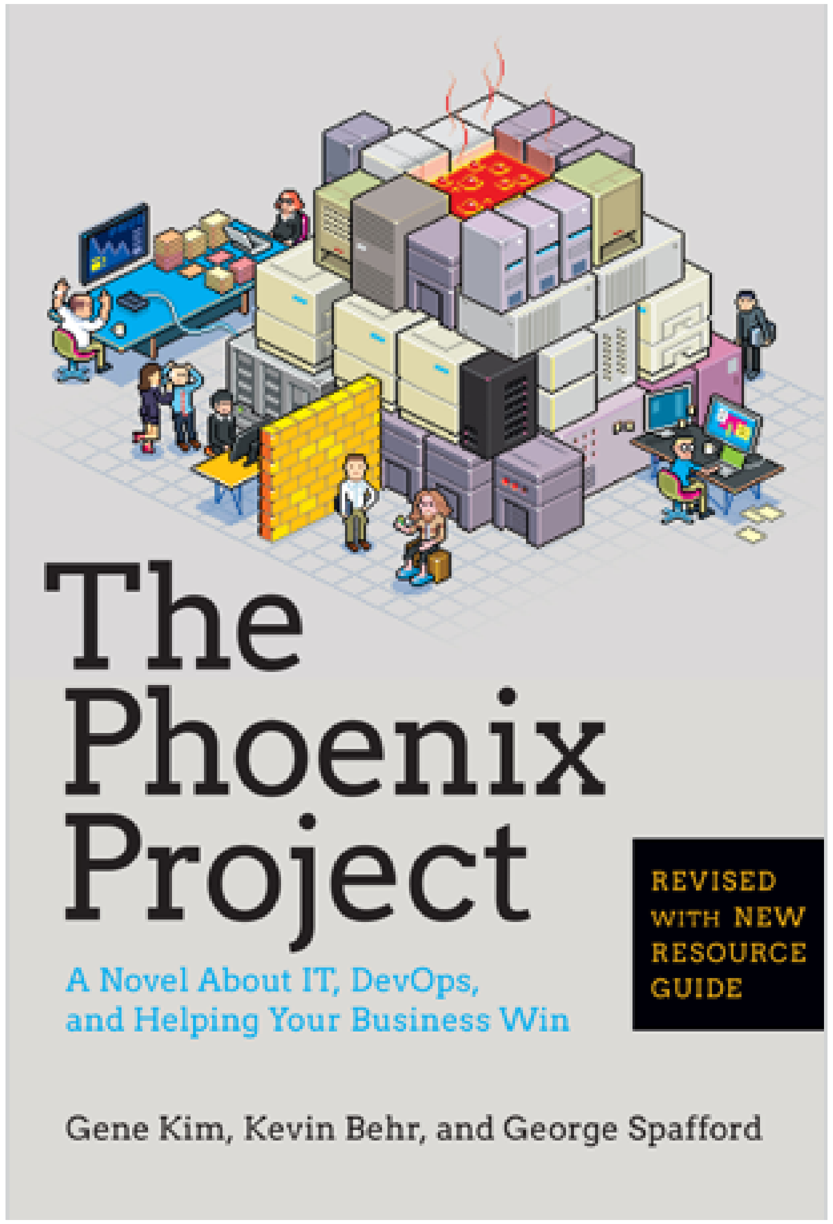
DevOps Days

Known as "the conference that brings development and operations together," this series of ongoing events happen in 12–15 locations a year. You can probably find one near you! And if not, there's information on planning your very own DevOps Days.

IT Revolution assembles technology leaders and practitioners through publishing, events, and research.

Our goal is to elevate the state of technology work, quantify the economic and human costs associated with suboptimal IT performance, and improve the lives of one million IT professionals by 2017.

To stay updated on IT Revolution's publishing, events, and research opportunities, please subscribe to our newsletter.


The Phoenix Project
A Novel About IT, DevOps, and Helping Your Business Win
REVISED WITH NEW RESOURCE GUIDE
Gene Kim, Kevin Behr, and George Spafford

The Phoenix Project:

A Novel About IT, DevOps, and Helping Your Business Win

Gene Kim, Kevin Behr, George Spafford

Bill is an IT manager at Parts Unlimited. It's Tuesday morning and on his drive into the office, Bill gets a call from the CEO.

The company's new IT initiative, code named Phoenix Project, is critical to the future of Parts Unlimited, but the project is massively over budget and very late. The CEO wants Bill to report directly to him and fix the mess in 90 days, or else Bill's entire department will be outsourced.

With the help of a prospective board member and his mysterious philosophy of The Three Ways, Bill starts to see that IT work has more in common with manufacturing plant work than he ever imagined. With the clock ticking, Bill must organize work flow, streamline interdepartmental communications, and effectively serve the other business functions at Parts Unlimited.

In a fast-paced and entertaining style, three luminaries of the DevOps movement deliver a story that anyone who works in IT will recognize. Readers will not only learn how to improve their own IT organizations, they'll never view IT the same way again.

Buy the book.

For bulk orders, please e-mail:
orders@itrevolution.net

Narrated by Chris Ruen
Listen to the first 16 chapters FREE on SOUNDCLOUD

Purchase the audiobook on amazon.com audible an amazon company Available on iTunes

DevOps Enterprise Summit

DevOps Enterprise is *the* event for people who are bringing Lean principles into the IT value stream while building DevOps and Continuous Delivery into their organization.

Join us for an incredible three-day event with the best practitioners from large and complex organizations, across all industry verticals. Lineup to include keynotes from industry luminaries and speakers from well-known enterprises who will share their enterprise DevOps initiatives. This event is a joint partnership between Electric Cloud and IT Revolution.

For more information on registering for the conference, submitting a proposal to present, or becoming a sponsor, please visit the **conference website.**

The DevOps Cookbook:

How to Create World-Class Agility, Reliability, and Security in Technology Organizations

We've been quiet on *The DevOps Cookbook*, but we're still at it! After receiving some incredible feedback in early 2015, we've been hard at work making revisions.

If you'd like to stay informed on the book and what's to come, sign up to receive **updates.**

Acknowledgments

We’d like to thank Delphix, Electric Cloud, Microsoft, New Relic, Rally, and XebiaLabs for contributing their outstanding content to our efforts to accomplish our mission.

And this excellent resource would never have come to be without the contributions of time, talent, and tenacity by Todd Sattersten, Gene Kim, Robyn Crummer-Olson, Alex Broderick-Forster, and Stauber Design Studio.