O'REILLY®

Compliments of Pivotal®

Monolithic Transformation

Using DevOps, Agile, and Cloud Platforms to Execute a Digital Transformation Strategy

Michael Coté

Monolithic Transformation

Using DevOps, Agile, and Cloud Platforms to Execute a Digital Transformation Strategy

Michael Coté

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Monolithic Transformation
by Michael Coté

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Alicia Young and Melissa Duffield
Production Editor: Nan Barber
Copyeditor: Octal Publishing, LLC
Proofreader: Nan Barber
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

March 2019: First Edition

Revision History for the First Edition
2019-02-15: First Release

This work is part of a collaboration between O'Reilly and Pivotal. See our statement of editorial independence.

978-1-492-04977-7

[LSI]

Table of Contents

Introduction: Why Change? v

1. Fostering Change 1
Small-Batch Thinking 1
Shift to User-Centric Design 4
From Functional Teams to Product Teams 5
Case Study: No One Wants to Call the IRS 12
Transforming Is Easy...Right? 13

2. Leadership's Cloud-Native Cookbook 15
Establishing a Vision and Strategy 17
Communicating the Vision and Strategy 22
Creating a Culture of Change, Continuous Learning, and Comfort 26
Building Your Business Case 38
Considering the Enterprise Architect 42
Tackling a Series of Small Projects 47
Assemble the Right Team 54
Building Trust with Internal Marketing, Large and Small 58
Tracking Your Improvement with Metrics 61
Tending to Compliance and Regulation 71
Building Your Pipeline and Choosing Your Platform 77
Own Your Role 89

Introduction: Why Change?

> *"We're in the technology business. Our product happens to be banking, but largely that's delivered through technology."*
>
> —Brian Porter, CEO, Scotiabank

The phrase "digital transformation" is mostly useless, but, then again, it's perfect. By my reckoning, the phrase has come to mean doing anything with new technologies, especially emerging technologies. When those new technologies span marketing efforts in Instagram to replatforming mainframe applications to containerized, 12-factor applications, it's not precise enough to be useful. But, there's utility in the phrase if it's narrowed. To me, the phrase means innovating new business models and fixing under-performing ones using rapidly delivered and well-designed software. For many businesses, fixing their long-dormant, lame software capabilities is an urgent need: companies like Amazon loom as overpowering competitors in most every industry. Competitive threats from these so-called "tech companies" are real, but some traditional enterprises are becoming even fiercer competitors.

Liberty Mutual, for example, entered a new insurance market on the other side of the world in six months, doubling the average close rate. Home Depot grew its online business by around $1 billion each of the past four years, is the number two-ranked digital retailer, and is adding more than 1,000 technical hires in 2018. The US Air Force created an air tanker scheduling app in 120 days, driving $1 million in fuel savings each week, and canceled the traditional, $745 million contract that hadn't delivered a single line of code in five years.

Though there are some exceptions, many organizations do not have the software capabilities needed to survive in this environment.

They've let their IT capabilities wither, too comfortable with low expectations and defensible business differentiation that ensured steady cash flows. Now they're trapped with slow, often calcified software. This makes the overall organization a sitting duck for merciless business predators. Organizations like those just mentioned show, however, that such fat ducks can become lean, quick, and profitable businesses if they transform how they do software.

I spend most of my time studying how large organizations plan for, initially fail at, and then succeed at this kind of transformation is—I'm super-fun at parties! This report documents what I've found so far, drawing on the experiences of companies that are suffering through the long journey to success. The ideas in this report are centered on the principles of small-batch thinking, user-centric design, and moving from functional teams to product teams, which we cover in Chapter 1. Chapter 2 then provides you with a playbook for applying these principles across your organization, from the first steps of creating and communicating your strategy all the way through building your pipeline and choosing the right cloud platform.

As you'll see, this report is built from the direct experience and research of others. I've tried to maximally cite these sources, primarily through linking to the source, presentation, book, article, or just the person. This means many citations are not available in print and only in the electronic version. If you find I'm missing a citation or a link, please send it along to me so that I can correct it in future versions and the errata.

CHAPTER 1
Fostering Change

> *"If you aren't embarrassed by the first version of your product, you shipped too late."*
>
> —Reid Hoffman, LinkedIn cofounder and former PayPal COO

Over the past 20 years, I've seen successful organizations use the same general process: shipping small batches of software in short iterations, using the resulting feedback loop to drive improvements to their software. These teams continuously learn and change their software to match user needs.

IT organizations that follow this process are delivering a different type of outcome rather than implementing a set of requirements. They're giving their organization the ability to adapt and change monthly, weekly, even daily. This is the outcome of Agile software development, and it's at the core of how IT can drive innovation for their organization.

Small-Batch Thinking

By "small batches," I mean identifying the problem to solve, formulating a theory of how to solve it, creating a hypothesis that can prove or disprove the theory, doing the smallest amount of coding necessary to test your hypothesis, deploying the new code to production, observing how users interact with your software, and then using those observations to improve your software. The cycle, of course, repeats itself, as illustrated in Figure 1-1.

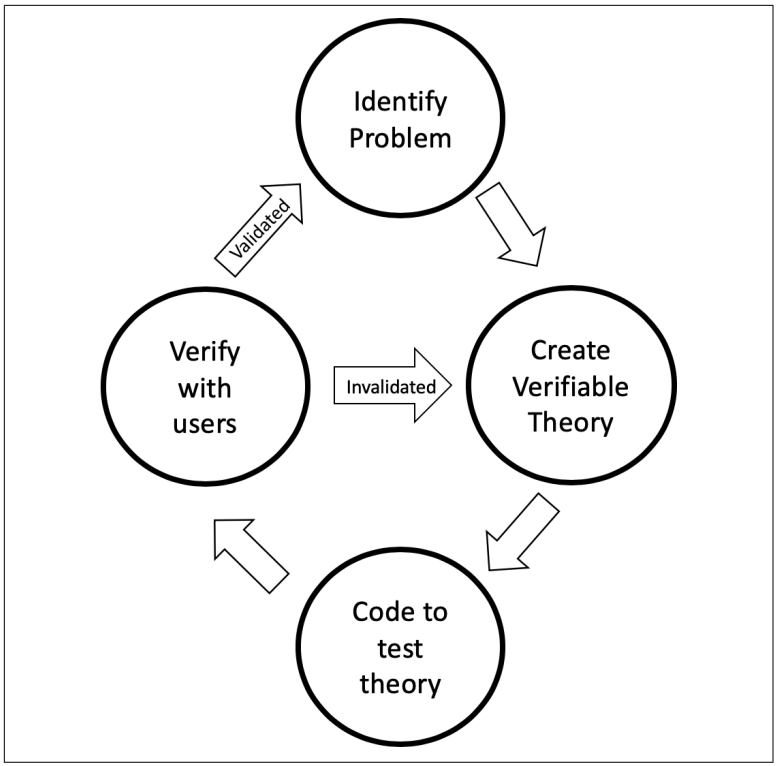

Figure 1-1. Small-batch thinking is a cyclical process.

This entire process should take at most a week—and hopefully just a day. All of these small batches, of course, add up over time to large pieces of software, but in contrast to a "large batch" approach, each small batch of code that survives the loop has been rigorously validated with actual users.

Schools of thought such as *Lean Startup* reduce this practice to helpfully simple sayings like "think, make, check." People like to prepend "lean" to various software roles, as well: lean production management, lean designer, or just lean on its own. What you call the process is up to you, as long as you actually follow it.

This discipline gives you a tremendous amount of insight into decisions about your software. A small-batch process gives you a much richer, fact-based ability to drive decisions about features to add, remove, or modify. In turn, this creates much better software. No

matter how many tests run green or schedule milestones are cleared, your software isn't done until a user validates your assumptions.

Liberty Mutual's Chris Bartlow describes the core benefit of small batches:

> When you get to the stoplight on the circle [the end of a small-batch loop] and you're ready to make a decision on whether or not you want to continue, or whether or not you want to abandon the project, or experiment [more], or whether you want to pivot, I think [being hypothesis driven] gives you something to look back on and say, "okay, did my hypothesis come true at all? Is it right on or is it just not true at all?"

The record of your experiments also serves as an ongoing report of your progress. Auditors will cherish this log of decision making and validation. These well-documented and tracked records are also your ongoing design history. This makes even your failures valuable because you've proven what *doesn't* work and, thus, what you should avoid in the future. You avoid the cost and risk of repeating bad decisions; these are savings that can be passed on to other teams who can now avoid those invalidated decisions, as well.

In contrast, a large-batch approach follows a different process. Teams document a pile of requirements up front, developers code away at implementing those features, perhaps creating "golden builds" each week or two (but not deploying those builds to production!), and only after all of the requirements are implemented and QA'ed is the code finally deployed to production. With the large-batch approach, this pile of unvalidated code creates a huge amount of risk.[1]

This is the realm of multiyear projects that either underwhelm or are consistently late. As one manager put it, "[w]e did an analysis of hundreds of projects over a multiyear period. The ones that delivered in less than a quarter succeeded about 80% of the time, while the ones that lasted more than a year failed at about the same rate."

No stranger to lengthy projects with big upfront analyses, the US Air Force is beginning to think in terms of small batches for its software, as well. "A [waterfall] mistake could cost $100 million, likely ending the career of anyone associated with that decision. A smaller mistake

1 In Lean terms, you can think of this as Work In Process (WIP), the unfinished work or just idle inventory sitting in your warehouse, wasting time and money.

is less often a career-ender and thus encourages smart and informed risk-taking," said M. Wes Haga in *Defense One: Future of the Navy*.

Shift to User-Centric Design

The small-batch loop enables a highly effective, user-centric approach to software design. The simplest, most accurate definition I've seen is from Pivotal Labs:

> User-centric design ensures that the software solves a real problem for real users in a desirable and usable product.

There's little new about taking a user-centric approach to software. What's different now is how much more efficient and fast it is thanks to highly networked applications and cloud-automated platforms.

In the past, studying and refining desktop software was difficult, costly, and slow. Outside of their labs, designers knew very little about how people used their software. Well, they knew when there were errors because users fumed about bugs. But, users rarely reported how well things were going when software functioned as planned. Worse, users didn't report when things were just barely good enough and could be improved. Without this type of knowledge, designers were left to, more or less, just make it up as they went along. Slow, multimonth, if not year, release cycles made this situation even worse. Even if designers knew how to structure the software better, they would need to wait through each release cycle to test their theories.

Starting in the mid-2000s, networked applications, released frequently, finally gave designers a rich trove of data to continually analyze and use to improve their software design quality. For example, a 2009 Microsoft study found that only about a third of its applications' features achieved the team's original goals—that is, were useful and considered successful. This allowed the team to adjust its focus accordingly, improving the software but also removing time wasted working on and supporting underused features.

Good design is worth spending time on. Maxie Schmidt-Subramanian and Laura Garvin Tramm at Forrester, for example, estimate that car manufacturers improving customer experience could bring $49 to $100 uplift in per customer revenue, and $8 to $10 in the banking industry. Those might seem small per person,

but multiplied across millions of customers, it adds up: $879 million for car manufacturers, and $123 million for banking.[2] Without a doubt, good design creates differentiation and drives revenue.

From Functional Teams to Product Teams

Traditional staffing and organization models are a poor fit for the small-batch process. Most organizations are grouped into functional silos: developers are in one bucket and project managers are in another bucket. The same goes for QA, security staff, database administrators, architects, and so on.

A small-batch approach, however, moves too fast for this type of organization. Communication and handoffs between each silo takes time: business analysts must communicate and explain requirements to developers; developers need to document features for QA and get the approval of architects; project managers need to spelunk in email and interview people to update project status; and DBAs, well, they're a deep, deep silo to cross.

When you're making weekly, if not daily, decisions about the product, all of this coordination also loses fidelity between each silo. Additionally, as Lean Manufacturing has discovered, those closest to the work are the best informed to make decisions about the product. Business analysts, enterprise architects, and, worst of all, executives simply don't have the week-to-week knowledge about the application and the people who use it.

Functional organizations also encourage local optimization: each group is focused on doing its part instead ensuring that the overall outcome is good.

To address these problems, you need to rearrange your organization chart to favor product teams. Let's look inside these teams and see what they're made of.

2 The exact numbers from the 2018 model are: $48.82 per customer for "mass market" auto manufactures, and $104.54 for luxury, spread across the average number of customers per company, resulting in $879 million and $37 million in revenue increase per company. For banking: multichannel at $8.19 per customer and $9.82 for direct, resulting in $123 million and $29 million per company.

Product Teams[3]

Each team is dedicated to one application, service, or sizable component of the overall system. Unless you use the First National Bank of Mattresses, you likely understand a retail banking system and its subcomponents: a ledger, a transfer service, a bill-paying service, new service sign-ups, loan applications, and so on. The backend will have numerous services like single sign-on, security checks, and so forth. Each of these areas maps to a product, as shown in Figure 1-2.

Enterprise architects are well positioned to define these groups. One method is to use domain-driven design (DDD) to discover the "bounded contexts" that define these teams. The entity and event trails described by a DDD analysis also help define the team's data models, dependencies, and methods of interacting with other teams (e.g., APIs and events). "Forming those teams allows us to distill to the heart of the problem we're focused on in order to gain competitive advantage," says Jakub Pilimon. As we discuss later, this intersection of business strategy and technical acuity is exactly the role of the enterprise architect.

After a product team is formed, its goal is to focus on the end-to-end process of delivering quality software. Instead of spreading that responsibility across many different groups, all of the roles needed to "own" the software are put on one team. That team is dedicated to the product full-time and given the trust and authority to decide on features, schedules, and most everything related to that product. In turn, they're responsible for running the software and its outcomes. This is analogous to how a software vendor works: the vendor's core business is delivering a product, not the implementation of a bunch of requirements from a distant third party.

3 Much of the role descriptions in this section are taken from and based on Pivotal Labs' guides and experience.

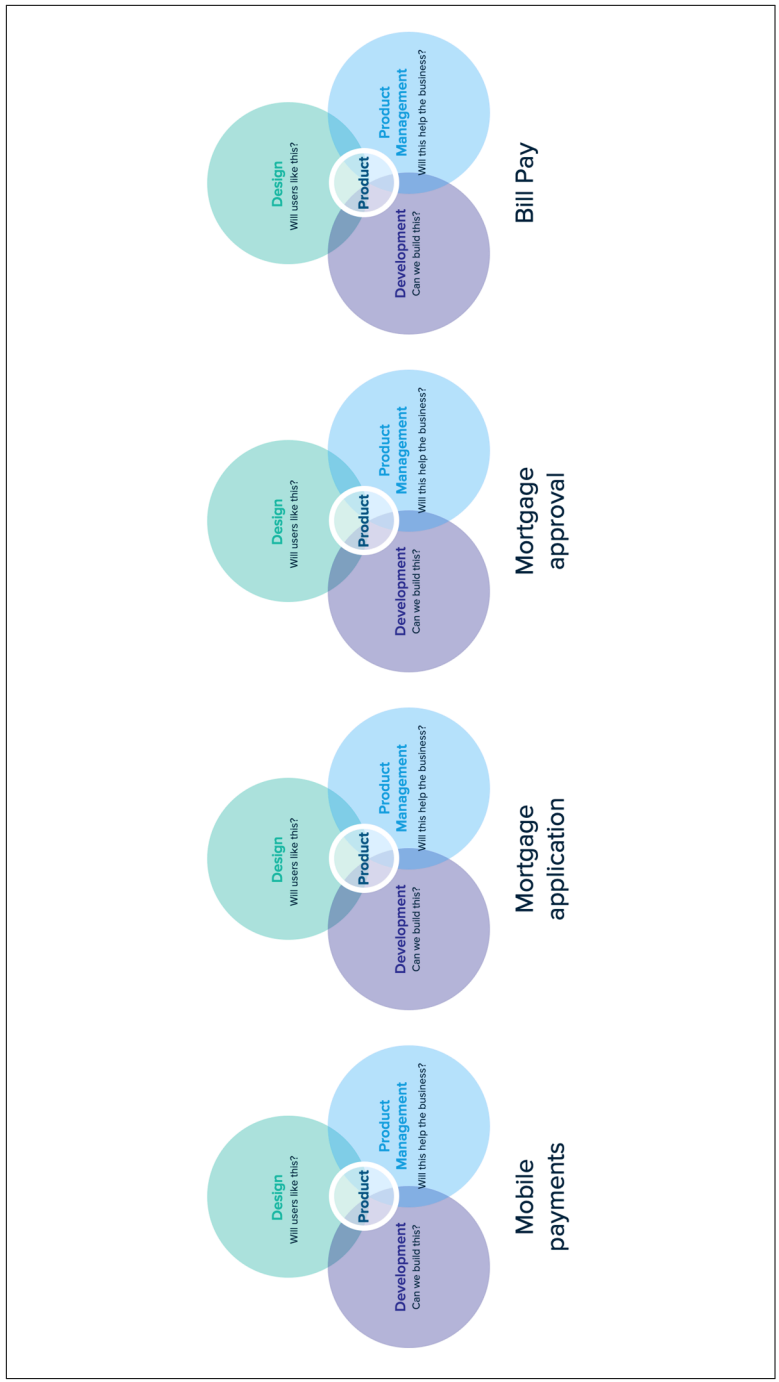

Figure 1-2. Examples of product teams by application and service.

As illustrated in Figure 1-3, each product team has three key roles: developers, designers, and product managers. In addition to these might be supporting roles that come and go as needed—testers, architects, DBAs, security staff, data scientists, and other specialists.

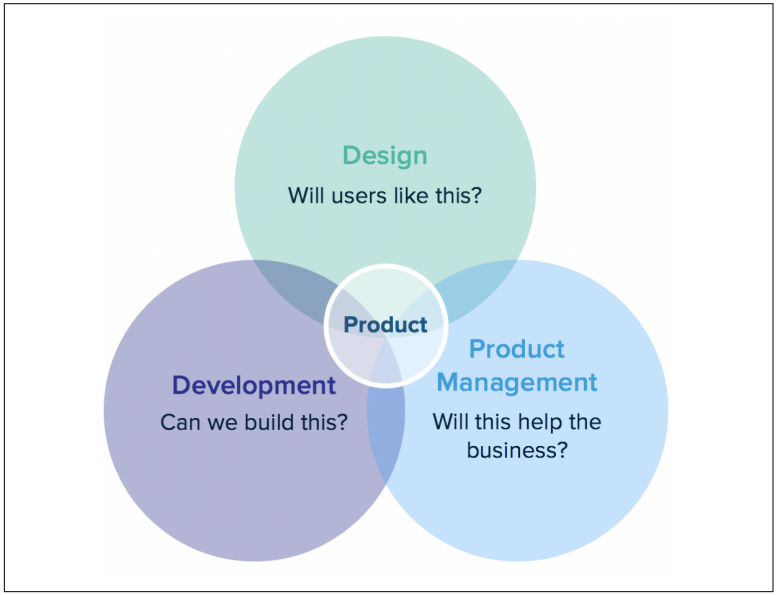

Figure 1-3. Any given product team has three major roles: developers, designers, and product managers.

Developers

These are the programmers. Developers also give technical input to the team, directing which technologies and architectural models to follow and build. They also determine the feasibility of various solutions: can we really do instant dog-breed recognition in real time on a PalmPilot?

Through the practice of *pairing*, knowledge is quickly spread among developers, ensuring that there are no empires built and addressing the risks of a low bus factor. Developers are encouraged to rotate through various roles, from frontend to backend, to get good exposure to all parts of a project. By using a cloud platform, like Pivotal Cloud Foundry, developers also use continuous integration and continuous delivery (CI/CD) tools to deploy code on their own, removing release management wait times.

Developers are closely involved with estimating the length of time that stories in your backlog will take to implement as well as in scoping stories. Similarly, developers can help product managers and designers walk through the technical trade-offs involved in deciding how to implement and design features.

Applying the principle of owning the product end to end, developers also ensure that their software runs well and can be managed in production. Some very advanced teams even have developers manage and remediate applications in product—a practice of orthodox DevOps. Having that kind of skin in the game motivates developers to write production-ready code.

You shouldn't expect developers to be mythical "full-stack developers"; that is, experts in all programming and operational concerns. Instead, they rely on a cloud platform's self-service and automation capabilities for most operations needs. This means, for example, that developers don't need to wait for operations staff to requisition development environments, set up middleware, deploy releases, or perform other configuration and release management tasks.

There will, of course, be operations knowledge that developers need to learn, especially when it comes to designing highly networked, distributed applications. Initially, prescriptive patterns help here as well as embedded operations staff. By relying on their organization's cloud platform to automate and enforce routine operations tasks, over time developers often gain enough operations knowledge to work without dedicated operations support.

The number of developers on each team is variable, but so far, you'll typically see anywhere from two to six developers, and sometimes more. Too many more and you risk reintroducing communications overhead; this is also often the sign of decadent scoping.

Product manager

At its core, the product manager is the "owner" of the team's software, the product. More accurately, product managers set the vision week-to-week, pointing the team in the right direction.

It's best to approach the product owner role as a breadth-first role: these individuals must understand the business, the customer, and the technical capabilities. This broad knowledge helps product managers make the right prioritization decisions. It also gives them the

knowledge needed to work with the rest of the organization and relevant stakeholders like auditors and "the business."

In organizations that are transitioning, this role also serves as a barrier between the all-too-fragile new teams and the existing, legacy teams. The product owner becomes the gatekeeper that keeps all of the "helpful" interest and requests at bay so that the teams can focus on their work.

Designer

One of the major lessons of contemporary software is that design matters a tremendous amount more than previously believed...or, at least, budgeted for. Although nice-looking UIs are all well and good, effective software design is much more than looks. The designer takes responsibility for deeply understanding the user's needs and challenges and how to create solutions to overcome these challenges. Designers are the architects of the conceptual workflows that make users productive.

The designer focuses on identifying the feature set for the application and translating that to a user experience for the development team. As some put it, design is how it works, not (just) how it looks. Activities might include completing the information architecture, user flows, wireframes, visual design, and high-fidelity mock-ups and style guides.

Most important, designers need to get out of the building and not only see what users are doing with the software, they need to become intimately familiar with those users and their needs. Peripateticism is helpful for all roles, of course, but vital for designers.

As discussed earlier, supported by platforms like Pivotal Cloud Foundry, the small-batch process gives the designers more feedback on their design theories than ever. This makes it feasible and worthwhile to embed designers on product teams.

Design has long been neglected by most organizations, which often find design skill severely lacking. The design role is critical to good software, however, so leaders need to quickly budget for and find designers.

Pairing Roles

Many of the high-performing teams I've seen *pair* across each role. There are always two people working on the same thing from product managers, designers, and developers. This practice increases work quality, educates and level-sets knowledge across the team, and builds trust and collaboration. Later on, pairing is used to spread your transformation throughout the rest of the organization by seeding new teams with people from paired teams.

Pairing might seem ridiculous at first: you've just halved production and doubled costs. However, almost 20 years of study and results from enterprises that practice pairing prove that it's an effective technique. For example, DBS Bank's CEO, Piyush Gupta, describes the bank's pairing policy, motivation, and strategic benefits:

> Mostly, we believe in pairing employees who need to be trained with others who are native to those capabilities; we have found this to be one of the most effective methods of transforming people. We hire people with the required capabilities and pair them up with the incumbents, and the result is that those capabilities are multiplied across the organization. This is rooted in the simple belief that in order to learn something, you must experience and practice it rather than hear about it in a classroom setting.

To take a look at pairing, let's use developers as an example. With rare exception, when programming, two developers always work together, following the maxim that two heads are better than one. Developers regularly rotate to a new partner, usually at least once a day. With two sets of eyes, quality rises and writing unit tests goes faster: one developer might write a unit test first followed by the other writing the corresponding code.

Knowledge spreads quickly and developers end up teaching each other, not only senior to junior, but from junior to senior, as well. In fact, after a while, the team's skills tend to level. Spreading knowledge across the team also improves risk management. A lone developer can't build specializations, becoming the only one who knows how "their" part of the system works. This means that staff churn has less negative effect. It also reduces the type of "hero culture" that can lead to staff burnout and mistakes.

"Part of the goal is to not have siloed knowledge," says Matt Curry (then at Allstate), "So we're kind of getting away from this world of optimizing for one person to show up on the scene and make every-

thing okay." Instead, he goes on, "Anybody on the team can make it okay or can get us over this barrier." That's a much more reliable team than a set of heroes with unique knowledge that only they can deploy to solve problems.

As you begin to spread change across your organization, you rotate key developers out of existing teams into new teams. These developers seed the new team with knowledge, skills, and trust in the new process. As we discuss later, this is one of the key methods for scaling change beyond the initial teams.

Case Study: No One Wants to Call the IRS

You wouldn't think a big government agency, particularly a tax-collecting organization, would be a treasure trove of good design stories, but the IRS provides a great example of how organizations are reviving their approach to software.

The IRS historically used call centers to provide basic account information and tax payment services. Call centers are expensive and error prone: one study found that only 37% of calls were answered. That is, the IRS hung up on more than 60% of people calling for help! With the need to continually control costs and deliver good service, the IRS had to do something.

In the consumer space, solving this type of account management problem has long been taken care of. It's pretty easy in fact. Just think of all the online banking systems you use and how you pay your monthly phone bills. But at the IRS, paying delinquent taxes had yet to be digitized.

When putting software around this, the IRS first thought that it should show you your complete history with the agency, over multiple years and types of transactions. This confused users and most of them still wanted to pick up the phone. Think about what a perfect failure that is: the software worked exactly as designed and intended, it was just the wrong way to solve the problem.

Thankfully, because the IRS was following a small-batch process, it caught this very quickly and iterated through different hypotheses of how to solve the problem. Finally, developers hit on a simple finding: when people want to know how much money they owe the IRS, they want to know *only* how much money they owe the IRS. The team removed all the lavish extra history from the UI. When this

version of the software was tested, most people didn't want to use the phone.

Now, if the IRS was on a traditional 12- to 18-month cycle (or longer!), think of how poorly this would have gone. The business case would have failed, and you would probably have a dim view of IT and the IRS. But, by thinking about software in an Agile, small-batch way, the IRS did the right thing, not only saving money, but also solving people's actual problems.

This project had great results: after some onerous upfront red-tape transformation, the IRS put an app in place that allows people to look up their account information, check payments due, and pay them. As of October 2017, there have been more than 2 million users, and the app has processed more than $440 million in payments. Clearly, a small-batch success.

Transforming Is Easy...Right?

Putting small-batch thinking in place is no easy task: how long would it take you, currently, to deploy a single line of code, from a whiteboard to actually running in production? If you're like most people, following the official process, it might take weeks—just getting on the change review board's schedule would take a week or more, and hopefully key approvers aren't on vacation. This thought experiment will start to flesh out what you need to do—or, more accurately, fix—to switch over to doing small batches.

Transforming one team amd one piece of software isn't easy, but it's often very possible. Improving two applications usually works. How do you go about switching 10 applications over to a small batch process? How about 500?

Supporting hundreds of applications and teams, plus the backing services that support these applications, is a horse of a different color, rather, a drove of horses of many different colors. There's no comprehensive manual for doing small batches at large scale, but in recent years several large organizations have been stampeding through the thicket. Thankfully, many of them have shared their successes, failures, and, most important, lessons learned. We look at their learnings next, with an eye, of course, at taming your organization's big-batch bucking.

CHAPTER 2

Leadership's Cloud-Native Cookbook

> *"[T]he role of leadership on technology transformation has been one of the more overlooked topics in DevOps, despite the fact that transformational leadership is essential"*
>
> —Accelerate: The Science of Lean Software and DevOps

In large organizations, higher-level executives are the only ones given the authority to make organizational change at a scale that matters and are, thus, key to any meaningful transformation. These executives must do more than update the "the slides" as so many corporate strategy groups seem to do. They must actually change the organization's structure, processes, norms, and rules—the organization's "culture."

The work won't be easy. "I've found that changing culture is by far more complicated than any software project I've worked on," Great American Insurance Company's Jon Osborn says, "partly because you have to deal with people and personalities." Managers rarely focus on "transformation," spending most of their time enforcing the status quo. "Management's mandate is to minimize risk and to keep the current system operating," as John Kotter summarized more than 20 years ago. This situation hasn't changed over the past two decades.

Changing how one team, or even five teams work is a neat trick and can even be done "bottoms up." But, changing, for example, JP Morgan Chase's 19,000 developers work requires big-time leadership

support and effort. This kind of challenge is something that only the leadership team can address. They're the ones responsible for the organizations' architecture and daily operation. Leading digital transformation is, to put it in developer terms, programming the organization.

In large organizations, the most sustainable, scalable change begins and ends with management. As the name would suggest, "digital transformation" requires transforming, which includes how the organization functions, its structure, and also how "things are done around here." Gardening all these are management's job. Just as IT and product teams must go through the failing-as-learning trials to transform and then iteratively create better software, management needs to continuously run through the small-batch loop, as well, to systematically transform their organization.

Next, let's look at some of the leadership and management tactics I've seen from successful organizations. These cover most major aspects of implementing a successful cloud-native leadership strategy, including the following:

- Establishing and communicating your vision and strategy
- Fostering a culture of change and continuous learning
- Planning your finances
- Valuing the role of enterprise architects in your transformation
- Creating successful projects
- Building strong teams
- Addressing change-averse leadership
- Creating trust with teams through internal marketing
- Tracking improvement with the right metrics
- Addressing compliance in every level of your business
- How to choose a strong platform and create efficient build pipelines

Establishing a Vision and Strategy

> *Start your project on Monday and ship it on Friday. It's no longer that it's going to take nine months.*
>
> —Andy Zitney, Allstate (at the time), and now McKesson

When you're changing, you need to know what you're changing to. It's also handy to know how you're going to change and, equally, how you're not going to change. In organizations, vision and strategy are the tools management uses to define why and how change happens.

Use Vision to Set Your Goals and Inspiration

"Vision" can be a bit slippery. Often it means a concise phrase of hope that can actually happen, if only after a lot of work. Andy Zitney's vision of starting on Monday and shipping on Friday is a classic example of vision. Vision statements are often little more than a sentence, but they give the organization a goal and the inspiration needed to get there. Everyone wants to know "why I'm here," which the vision should provide, helping stave off any corporate malaise and complacency.

Kotter has an excellent description of vision as ever divided into a list:

> Vision refers to a picture of the future with some implicit or explicit commentary on why people should strive to create that future. In a change process, a good vision serves three important purposes. First, by clarifying the general direction for change, by saying the corporate equivalent of "we need to be south of here in a few years instead of where we are today," it simplifies hundreds or thousands of more detailed decisions. Second, it motivates people to take action in the right direction, even if the initial steps are personally painful. Third, it helps coordinate the actions of different people, even thousands and thousands of individuals, in a remarkably fast and efficient way.

Creating and describing this vision is one of the first tasks a leader, and then their team, needs to do. Otherwise, your staff will just keep muddling through yesterday's success, unsure of what to change, let alone why to change. In IT, a snappy vision also keeps people focused on the right things instead of focusing on IT for IT's sake. "Our core competency is 'fly, fight, win' in air and space," says the US

Air Force's Bill Marion, for example, "It is not to run email servers or configure desktop devices."

The best visions are simple, even quippy sentences. "Live more, bank less" is a great example from DBS Bank. "[W]e believe that our biggest competitors are not the other banks," DBS's Siew Choo Soh says. Instead, she continues, competitive threats are coming from new financial tech companies "who are increasingly coming into the payment space as well as the loan space."

DBS Bank's leadership believes that focusing on the best customer experience in banking will fend off these competitors and, better, help DBS become one of the leading banks in the world. This isn't just based on rainbow whimsey, but strategic data: in 2017, 63% of total income and 72% of profits came from digital customers. Focusing on that customer set and spreading whatever magic brought in that much profit to the "analog customers" is clearly a profitable course of action.

"We believe that we need to reimagine banking to make banking simple, seamless, as well as invisible to allow our customers to live more and bank less," Soh says. A simple vision like that is just the tip of the iceberg, but it can easily be expanded into strategy and specific, detailed actions that will benefit DBS Bank for years to come. Indeed, DBS has already won several awards, including *Global Finance* magazine's best bank in the world for 2018.

Instilling "a sense of urgency," as Kotter describes it, is also very useful. Put cynically, you need people to be sufficiently freaked out to willingly suffer through the confusing and awkward feels of change. With companies like Amazon entering new markets right and left, urgency is easy to find nowadays. Taking six months to release competitive features isn't much use if Amazon can release them in two months.

Be judicious with this sense of urgency, however, lest you become chicken little. Executives and boards seem to be most susceptible to industry freaking out, but middle management and staff have grown wary to that tool over the years.

Create an Actionable Strategy

"Strategy" has many, adorably nuanced and debated definitions. Like enterprise architecture, it's a term that at first seems easily

knowable but becomes more obtuse as you stare into the abyss. A corporate strategy defines how a company will create, maintain, and grow business value. At the highest level, the strategy is usually increasing investor returns, generally through increasing the company's stock price (via revenue, profits, or investor's hopes and dreams thereof), paying out dividends, or engineering the acquisition of the company at a premium. In not-for-profit organizations, "value" often means how effectively and efficiently the organization can execute its mission, be that providing clean water, collecting taxes, or defending a country. The pragmatic part of strategy is cataloging the tools that the organization has at its disposal to achieve, maintain, and grow that value. More than specifying which tools to use, strategy also says what the company will not do.

People often fail at writing down useful strategy and vision. They want to serve their customers, be the best in their industry, and other such thin bluster. The authors of *Winning Through Innovation* provide a more practical recipe for defining your strategy:

1. Who are your customers and what are their needs?
2. Which market segments are you targeting?
3. How broad or narrow is your product or service offering?
4. Why should customers prefer your product or service to a competitor's?
5. What are the competencies you possess that others can't easily imitate?
6. How do you make money in these segments?

Strategy should explain how to deliver on the vision with your organization's capabilities, new capabilities enabled by technologies, customers' needs and jobs to be done, your market, and your competitors. "This is where strategy plays an important role," Kotter says, "Strategy provides both a logic and a first level of detail to show how a vision can be accomplished."

There are endless tools for creating your strategy, including hiring management consulting firms, focusing on cost or better mouse traps, eating nothing but ramen noodles, drawing on napkins, and playing the boardroom version of The Oregon Trail. If you don't already have a strategy definition method, it doesn't really matter

which one you choose. They're all equally terrible if you do nothing and lack an actionable strategy.

Case study: a strategy for the next 10 years of growth at Dick's Sporting Goods

Dick's Sporting Goods, the largest sporting goods retailer in the US, provides a recent example of putting a higher-level vision and strategy into action. As described by Jason Williams, over the past 10 years Dick's rapidly built out its ecommerce and omni-channel capabilities—an enviable feat for any retailer. As always, success created a new set of problems, especially for IT. It's worth reading Williams's detailed explanation of these challenges:

> With this rapid technological growth, we've created disconnects in our overall enterprise view. There were a significant number of store technologies that we've optimized or added on to support our ecommerce initiatives. We've created an overly complex technology landscape with pockets of technical debt, we've invested heavily in on premise hardware—in the case of ecommerce, you have to plan for double peak, that's a lot of hardware just for one or two days of peak volume. Naturally, this resulted in a number of redundant services and applications; specifically, we have six address verification services that do the same thing. And not just technical issues, we often had individuals and groups that have driven for performance, but it doesn't align to our corporate strategy. So why did we start this journey? Because of our disconnect in enterprise view, we lack that intense product orientation that a lot of our competitors already had.

These types of "disconnects" and "pockets of technical debt" are universal problems in enterprises. Just as with Dick's, these problems are usually not the result of negligence and misfeasance, but of the actions needed to achieve and maintain rapid growth.

To clear the way for the next 10 years of success, Dick's put in place a new IT strategy, represented by four pillars:

Product architecture
: Creating an enterprise architecture based around the business; for example, pricing, catalog, inventory, and other business

functions.[1] This focus helps shift from a function- and service-centric mindset to a product-centric mindset.

Modern software development practices
: Using practices like test-driven development (TDD), pairing, continuous integration and continuous delivery (CI/CD), lean design, and all the proven, Agile best practices.

Software architecture
: Using a microservices architecture, open source, following 12-factor principles to build cloud-native applications on top of Pivotal Cloud Foundry. This defines how software will be created, reducing the team's toil so that they can focus on product design and development.

Balanced teams
: Finally, as Williams describes it, having a unified, product-centric team is the "the most critical part" of Dick's strategy. The preceding three provide the architectural and infrastructural girding to shift IT from service delivery over to product delivery.

Focusing on these four areas gives staff very clear goals that easily translate into next steps and day-to-day work. Nine months after executing this strategy, Dick's achieved tangible success: the company created 31 product teams, increased developer productivity by 25%, ramped up its testing to 70% coverage, and improved the customer experience by increasing page load time and delivering more features, more frequently.

Keep Your Strategy Agile

Finally, keep your strategy Agile. Even though your vision is likely to remain more stable year to year, how you implement it might need to change. External forces will put pressure on a perfectly sound strategy: new government regulations or laws could change your organization's needs, Amazon might finally decide to bottom out your market. You need to establish a strategy review cycle to check

1 Domain Driven Design (DDD) is often used to define these "bounded contexts." Similarly, on the product side, the jobs to be done (JTBD) methodology can define the business architecture.

your assumptions and make course corrections to your strategy as needed. That is, apply a small-batch approach to strategy.

Organizations usually review and change strategy on an annual basis as part of corporate planning, which is usually little more than a well-orchestrated fight between business units for budget. Although this is an opportunity to review and adjust strategy, it's at the whim of finance's schedule and the mercurial tactics of other business units.

Annual planning is also an unhelpfully Waterfall-centric process, as pointed out by Mark Schwartz in *The Art of Business Value* (IT Revolution). "The investment decision is fixed," he writes, but "the product owner or other decision maker then works with that investment and takes advantage of learnings to make the best use possible of the investment within the scope of the program. We learn on the scale of single requirements, but make investment decisions on the scale of programs or investment themes—thus the impedance mismatch."

A product approach doesn't thrive in that annual, fixed mindset. Do at least an additional strategy review each year and many more in the first few years as you're learning about your customers and product with each release. Don't let your strategy become hobbled by the fetters of the annual planning and budget cycle.

Communicating the Vision and Strategy

If a strategy is presented in the boardroom but employees never see it, is it really a strategy? Obviously not. Leadership too often believes that the strategy is crystal clear, but staff usually disagree. For example, in a survey of 1,700 leaders and staff, 69% of leaders said their vision was "pragmatic and could be easily translated into concrete projects and initiatives." Employees had a glummer picture: only 36% agreed.

Your staff likely doesn't know the company's vision and strategy. More than just understanding it, they rarely know how they can help. As Boeing's Nikki Allen put it:

> In order to get people to scale, they have to understand how to connect the dots. They have to see it themselves in what they do—whether it's developing software, or protecting and securing the network, or provisioning infrastructure—they have to see how the work they do every day connects back to enabling the business to either be productive or generate revenue.

Use Internal Channels

There's a little wizardry to communicating strategy. First, it must be compressible. But you already did that when you established your vision and strategy (see "Establishing a Vision and Strategy" on page 17...right? Next, you push it through all the mediums and channels at your disposal to tell people over and over again. Chances are, you have "town hall" meetings, email lists, and team meetings up and down your organization. Recording videos and podcasts of you explaining the vision and strategy is helpful. Include strategy overviews in your public speaking because staff often scrutinizes these recordings. Even though "Enterprise 2.0" fizzled out several years ago, Facebook has trained us all to follow activity streams and other social flotsam. Use those habits and the internal channels that you have to spread your communication.

Show Your Strategy in Action

You also need to include examples of the strategy in action: what worked and what didn't work. As with any type of persuasion, getting people's peers to tell their stories are the best. Google and others find that celebrating failure with company-wide post mortems is instructive, career-ending crazy as that might sound. Stories of success and failure are valuable because you can draw a direct line between high-level vision to fingers on keyboard. If you're afraid of sharing too much failure, try just opening up status metrics to staff. Leadership usually underestimates the value of organization-wide information radiators, but staff usually wants that information to stop prairie dogging through their 9 to 5.[2]

2 "Just 16% of all respondents say regular access to information on the effort's progress is an effective way to engage the front line. But nearly twice the share of frontline respondents say the same." —Dana Maor, Angelika Reich, Lara Yocarini, *McKinsey Global Survey,* June, 2016.

Gather Feedback

As you're progressing, getting feedback is key: do people understand it? Do people know what to do to help? If not, it's time to tune your messages and mediums. Again, you can apply a small batch process to test out new methods of communicating. Although I find them tedious, staff surveys help: ask people whether they understand your strategy. Be sure to also ask whether they know how to help execute the strategy.

Create a Manifesto

Manifestos can help decompose a strategy into tangible goals and tactics. The insurance industry is on the cusp of a turbulent competitive landscape. To call it "disruptive" would be too narrow. To pick one sea of chop, autonomous vehicles are "changing everything about our personal auto line and we have to change ourselves," says Liberty Mutual's Chris Bartlow. New technologies are only one of many fronts in Liberty's new competitive landscape. All existing insurance companies and cut-throat competitors like Amazon are using new technologies to optimize existing business models and introduce new ones.

"We have to think about what that's going to mean to our products and services as we move forward," Bartlow says. Getting there required reengineering Liberty's software capabilities. Like most insurance companies, mainframes and monoliths drove its success over past decades. That approach worked in calmer times, but now Liberty is refocusing its software capability around innovation more than optimization. Liberty is using a stripped-down set of three goals to make this urgency and vision tangible.

"The idea was to really change how we're developing software. To make that real for people we identified these bold, audacious moves, or 'BAMS,'" says Liberty Mutual's John Heveran:

- 60% of computing workloads to the cloud
- 75% of technology staff writing code
- 50% of code releasing to production in a day

These BAMs grounded Liberty's strategy, giving staff very tangible, if audacious, goals. With these in mind, staff could begin thinking about *how* they'd achieve those goals. This kind of manifesto makes strategy actionable.

So far, it's working. "We're just about across the chasm on our DevOps and CI/CD journey," says Liberty's Miranda LeBlanc. "I can say that because we're doing about 2,500 daily builds, with over 1,000 production deployments per day," she adds. These numbers are tracers of putting a small-batch process in place that's used to improve the business. They now support around 10,000 internal users at Liberty and are better provisioned for the long ship ride into insurance's future.

Choosing the right language is important for managing IT transformation. For example, most change leaders suggest dumping the term "Agile." At this point, near 25 years into "Agile," everyone feels like they're Agile experts. Whether that's true is irrelevant. You'll faceplam your way through transformation if you're pitching switching to a methodology people believe they've long mastered.

It's better to pick your own branding for this new methodology. If it works, steal the buzzwords du jour, such as "cloud native," DevOps, or serverless. Creating your own brand is even better. As we discuss later, Allstate created a new name, CompoZed Labs, for its transformation effort. Using your own language and branding can help bring smug staff onboard and involved. "Oh, we've always done that, we just didn't call it 'Agile,'" sticks-in-the-mud are fond of saying as they go off to update their Gantt charts.

Make sure people understand why they're going through all of this "digital transformation." And make even more sure that they know how to implement the vision and strategy, or, as you start thinking, our strategy.

Creating a Culture of Change, Continuous Learning, and Comfort

> *In banking, you don't often get a clean slate like you would at some of the new tech companies. To transform banking, you not only need to be equipped with the latest technology skills, you also need to transform the culture and skill sets of existing teams, and deal with legacy infrastructure.*
>
> —Siew Choo Soh, Managing Director, DBS Bank

Most organizations have a damaging mismatch between the culture of service management and the strategic need to become a product organization. In a product culture, you need the team to take on more responsibility, essentially all of the responsibility, for the full life cycle of the product. Week to week, they need to experiment with new features and interpret feedback from users. In short, they need to become innovators.

Service delivery cultures, in contrast, tend more toward a culture of following upfront specification, process, and verification. Too often when put into practice, IT Service Management (ITSM) becomes a governance bureaucracy that drives project decision. This governance-driven culture tends to be much slower at releasing software than a product culture.

The sadly maligned architectural change advisory boards (CABs) are an example, well characterized by Jon Hall:

> [A] key goal for DevOps teams is the establishment of a high cadence of trusted, incremental production releases. The CAB meeting is often seen as the antithesis of this: a cumbersome and infrequent process, sucking a large number of people into a room to discuss whether a change is allowed to go ahead in a week or two, without in reality doing much to ensure the safe implementation of that change.[3]

Recent studies have even suggested that too much of this process, in the form of CABs, actually damages the business. Most ITSM experts don't so much disagree as suggest that these governance

3 I should note that Jon, far from being yet another cackling ITSM embalmer, goes on to explain how ITSM has adapted and changed in response to DevOps.

bureaucracies are doing it wrong. ITSM has been evolving and can evolve more to fit all this new-fangled product think, they add.

Despite the best intentions of ITSM adherents, IT organizations that put service management into practice tend to become slow and ineffective, at least when it comes to change and innovation.

The most difficult challenge for leaders is changing this culture.

What Even Is Culture?

"Culture" is a funny word in the DevOps, Agile, and digital transformation world. I don't particularly like it, but it's the word we have. Being soft and squishy, it's easy to cargo cult culture by providing 10 ways to make coffee, ping-pong tables, and allowing people to wear open-toed shoes in the office. These things are, for better or worse, often side effects of a good software culture, but they won't do anything to actually change culture.

Mainstream organizational management work has helpful definitions of culture: "Culture can be seen in the norms and values that characterize a group or organization," O'Reilly and Tushman write, "that is, organizational culture is a system of shared values and norms that define appropriate attitudes and behaviors for its members."

Jez Humble points out another definition, from Edgar Schein:

> [Culture is] a pattern of shared tacit assumptions that was learned by a group as it solved its problems of external adaptation and internal integration, that has worked well enough to be considered valid and, therefore, to be taught to new members as the correct way to perceive, think, and feel in relation to those problems.

We should take "culture," then, to mean the mindset used by people in the organization to make day-to-day decisions, policy, and best practices. I'm as guilty as anyone else of dismissing "culture" as simple, hollow acts like allowing dogs under desks and ensuring that there's six different ways to make coffee in the office. Beyond trivial pot-shots, paying attention to culture is important because it drives how people work and, therefore, the business outcomes they achieve.

For many years, the DevOps community has used the Westrum spectrum to describe three types of organizational culture, the worst of which ring too true with most people, as shown in Table 2-1.

Table 2-1. The three major kinds of organizational cultures: pathological, bureaucratic, and generative

Pathological	Bureaucratic	Generative
Power-oriented	**Rule-oriented**	**Performance-oriented**
Low cooperation	Modest cooperation	High cooperation
Messengers shot	Messengers neglected	Messengers trained
Responsibilities shirked	Narrow responsibilities	Risks are shared
Bridging discouraged	Bridging tolerated	Bridging encouraged
Failure leads to scapegoating	Failure leads to justice	Failure leads to enquiry
Novelty crushed	Novelty leads to problems	Novelty implemented

Year after year, the DevOps reports show that "high performing" organizations are much more generative than pathological—as you would suspect from the less-than-rosy words chosen to describe "power-oriented" cultures. It's easy to identify your organization as pathological and equally easy to realize that doing that alone is unhelpful. Moving from the bureaucratic column to the generative column, however, is where most IT organizations struggle.[4]

The Core Values of Product Culture

There are two layers of product culture, at least that I've seen and boiled down over the years. The first layer describes the attitudes of product people, the second the management tactics you put in place to get them to thrive.

Regarding the first layer, product people should be the following:

Innovative
: They're interested in discovering problems, solving problems, and coming up with new ways to accomplish inefficient tasks. These kinds of people also value continuous learning, without which innovation can't happen except by accident: you don't want to depend on accidentally dropping a burrito into a deep fryer to launch your restaurant chain.

Risk takers
: I don't much like this term, but it means something very helpful and precise in the corporate world; namely, that people are will-

4 Simon Wardley's Pioneers, Settlers, and Town Planners categories are another, useful cultural descriptor.

ing to do something that has a high chance of failing. The side that isn't covered enough is that they're also focused on safety. "Don't surf if you can't swim," as Andrew Clay Shafer summed it up. Risk takers ensure they know how to "swim" and they build safety nets into their process. They follow a disciplined approach that minimizes the negative consequences of failure. The small-batch process, for example, with its focus on a small unit of work (a minimal amount of damage if things go wrong and an easier time diagnosing what caused the error) and studying the results, good and bad, creates a safe, disciplined method for taking risks.

People focused

Products are meant to be used by people, whether as "customers" or "employees." The point of this report is to make software that better helps people, be that delivering a product they like using or one that allows them to be productive—getting banking done as quickly as possible so that they can get back to living their life, to lengthen DBS Bank's vision. Focusing on people, then, is what's needed. Too often, some people are focused on process and original thinking, sticking to those precepts even if they prove to be ineffective. People-focused staff will instead be pragmatic, looking to observe how their software is helping or hindering the people we call "users." They'll focus on making people's lives better, not achieving process excellence, making schedules and dates, or filling out request tickets correctly.

Finding people like this can seem like winning the lottery. Product-focused people certainly are difficult to find and valuable, but they're a lot less rare than you'd think. More important, you can create them by putting the right kind of management policy and nudges in place. A famous quip by Adrian Cockcroft (then at Netflix, now at Amazon) illustrates this. As he recounts:

> [A]t a CIO summit I got the comment, "we don't have these Netflix superstar engineers to do the things you're talking about," and when I looked around the room at the company names my response was "we hired them from you and got out of their way."

There is no talent shortage, just a shortage of management imagination and gumption.

As for the second, management-focused layer of product culture, over and over again, research[5] finds that the following gumptions give you the best shot at creating a thriving, product-centric culture: autonomy, trust, and voice. Each of these three support and feed into one another, as you'll see.

Autonomy

People who are told exactly what to do tend not to innovate. Their job is not to think of new ways to solve problems more efficiently and quickly, or solve them at all. Instead, their job is to follow the instructions. This works extremely well when you're building IKEA furniture, but following instructions is a poor fit when the problem set is unknown, when you don't even know whether you know that you don't know.

Your people and the product teams need autonomy to study their users, theorize how to solve their problems, and fail their way to success. Pour on too much command-and-control, and they'll do exactly what you don't want: they'll follow your orders perfectly. A large part of a product-centric organization's ability to innovate is admitting that the people closest to the users—the product team—are the most informed about what features to put into the software and even what the user's problems are. You, the manager, should be overseeing multiple teams and supporting them by working with the rest of the organization. You'll lack the intimate, day-to-day knowledge of the users and their problems. Just as the business analysts and architects in a Waterfall process are too distant from the actual work, you will be too and will make the same errors.

The 2018 DORA DevOps report suggests a few techniques for helping product teams gain autonomy:

- Establishing and communicating goals, but letting the team decide how the work will be done
- Removing roadblocks by keeping rules simple
- Allowing the team to change rules if the rules are obstacles to achieving the goals

5 Most recently as studied and described in DORA's 2018 *Accelerate*: "State of DevOps" report.

- Letting the team prioritize good outcomes for customers, even if it means bending the rules

This list is a good start. As ever, apply a small-batch mentality to how you're managing this change and adapt according to your findings.

There are some direct governance and technology changes needed to give teams this autonomy. The product teams need a platform and production tools that allow them to actually manage the full life cycle of their product. "[I]f you say to your team that 'when you build it you also run it,'" says Rabobanks' Vincent Oostindië, "you cannot do that with a consolidated environment. You cannot say to a team, 'you own that stuff, and by the way somebody else can also break it.'"

While it might seem trivial, even the simple act of adding a new field in an address form can limit the team's autonomy if they're reliant on a database administrator to add the field...whenever the DBA has time to get around it. This type of autonomy relies on management giving the team autonomy, but it also requires the transformational benefits of using a cloud platform to give teams self-service access to everything they need and, then, monitor and manage their applications in production as much as possible.

Trust

We often celebrate the idea of "risk takers," but we rarely celebrate "failures." Confidently, we forget that a high chance of failure is exactly what risk taking is: often, an assurance of failure. Meaningful innovation requires a seemingly endless amount of failure. This means taking risks trying new features, resolving problems in production, and, especially, changing "how things are done around here." Leadership is the only part of the organization that creates effective, long-term trust, every other part of the organization will follow their management's lead. The DORA report defines trust in this context as "how much a person believes their leader or manager is honest, has good motives and intentions, and treats them fairly."

To succeed at digital transformation, the people in the product teams must trust management. Changing from a services-driven organization of a product organization requires a great deal of upheaval and discomfort. Staff are being asked to behave much differently than they've been accustomed to in the past. The new orga-

nization can seem threatening to careers, and why should we trust what management is saying, anyway? It almost sounds too good to be true.

Management needs to first demonstrate that its desire to change can be trusted. Doing things like celebrating failures, rewarding people for using the new methods, and spending money on the trappings of the new organization (like free breakfast, training, and, indeed, 10 different ways of making coffee) will demonstrate management's commitment.

The "blameless postmortem" is the most popular representation of this trust. In this approach, after a failure in production, in design, or maybe even in coffee grinding is resolved, the team that committed the failure and fixed it gives its story to the wider organization. First, the goal is to build up trust that failure is OK. Of course, as one former boss used to say, "It's OK to screw up. But please screw up in only *new* ways next time."[6] Second, a blameless postmortem is actually educational and will allow you to put in place practices and safeguards to better prevent another, or even similar failures. Pretty quickly, you want to start "blaming" the process and policy that you have in place, not the people working in that context. This will force you to fix the process and create the right kind of work context people need to succeed at innovation and problem solving.

Just as staff must trust management, managers must trust the product teams to be responsible and independent. This means managers can't constantly check in on and meddle in the day-to-day affairs of product teams. Successful managers will find it all too tempting to get their hands dirty and volunteer to help out with problems. Getting too involved on a day-to-day basis is likely to hurt more than help, however.

Felten Buma suggests an exercise to help transform "helicopter managers." In a closed meeting of managers, ask them to each share one of their recent corporate failures. Whether you discuss how it was fixed is immaterial to the exercise, the point is to have the managers practice being vulnerable and then show them that their career doesn't end. Then, to practice giving up control, ask them to delegate an important task of theirs to someone else. Buma says that,

6 The exact words were closer to, "It's OK to screw it up. Just don't screw it up a second time." But I prefer to remember it in the more sparkly DevOps wording.

surprisingly, most managers find these two tasks very difficult, and some outright reject it. Those managers who can go through these two exercises are likely mentally prepared to be effective, transformational leaders.

Voice

The third leg of transformative leadership is giving product teams a voice. When teams trust management and begin acting more autonomously, they'll need to have the freedom to speak up and suggest ways to improve not only the product, but the way they work. A muzzled product team is much less valuable than one that can speak freely. As the DORA report defines it:

> Voice is how strongly someone feels about their ability and their team's ability to speak up, especially during conflict; for example, when team members disagree, when there are system failures or risks, and when suggesting ideas to improve their work.

Put another way, you don't want people to be "courageous." Instead, you want open discussions of failure and how to improve to be common and ordinary, "boring," not "brave." The opposite of giving your team's voice is suppressing their suggestions, dismissing the team's suggestions because "that's not your job," and explaining why such thinking is dangerous or "won't work here."

Traditional managers tend to be deeply offended when "their" staff speaks to the rest of the organization independently, when they "go around" their direct line managers. This kind of thinking is a good indication that the team lacks true voice. Although it's certainly more courteous to involve your manager in such discussions, management should trust teams to be autonomous enough to do the right thing.

In an organization like the US Air Force, in which you literally must ask permission to "speak freely," giving product teams voice can seem impossible. To solve this problem, the Kessel Run team devised a relatively simple fix: it asked the airmen and women to wear civilian clothes when they were working on their products. Without the explicit reminder of rank that a uniform and insignia enforce, team members found it easier to talk freely with one another, regardless of rank. Of course, managers also are explicitly told and encouraged this behavior. Other organizations such as Allstate have used this same sartorial trick, encouraging managers to change from button-up shirts and suits to t-shirts and hoodies, instead. Dress can be sur-

prisingly key for changing culture. Matt Curry, then at Allstate, lays out the case:

> T-shirts are a symbol. Since I have given that talk, I have received probably 10 emails from Allstate people asking for a shirt. Like I said in the talk, the shirt gives people permission to dress differently, which surprisingly enough makes them act differently. It is really interesting how much more collaborative and conversational people are when they are in casual dress. At the end of the day, we are just looking to start by changing one behavior, which can get the momentum spinning to drive a much larger cultural change. T-shirts and hoodies are a very fun and non-confrontational way to get that change started and begin a discussion about how things should be.

As you can see, culture changes can sometimes start with something as simple as a T-shirt. Better get some printed.

Case study: the line worker knows best at Duke Energy

One of Duke Energy's first product-centric applications is a good example of autonomy, trust, and voice in action. Duke wanted to improve how line workers coordinated their field work. At first, the unit's vice president reckoned that a large map showing each line workers' location would help him improve scheduling and work queues. The executive would have more visibility and, thus, more control to optimize truck rolls.

Working autonomously, the product team went further than just trusting the VP's first instincts, deciding that it should do field research with the actual line workers. After getting to know the line workers, the team discovered a solution that redefined the problem and the solution. Even though the VP's map would be a fine dashboard and give more information to the central office, what really helped was developing a much more automated job assignment application for line workers. This app would also let line workers locate their peers to, for example, let line workers elect to partner on larger jobs, and avoid showing up at the same job. The app also introduced an Uber-like queue of work within which line workers could self-select which job to do next.

The organization trusted the product team to come up with the right solution, and as the team worked with the line workers through changes each week, some features working and some not, management trusted that the team would get it right, eventually. The

team was also given the voice needed to first question and even upend the original request from the VP. If the team had lacked autonomy, trust, and voice, they likely would have ended up with a very functional map that showed the location of each line worker, a solution that would have likely been much less effective.

Monitoring Culture Change

Improving culture is a never-ending, labor-intensive process. There are several ways to monitor the progress of building a new culture:

Surveys
: Employee surveys are a good way to monitor progress. You should experiment with what to put in these surveys, and even other means of getting feedback on your organization's culture. Dick's Sporting Goods narrowed down to Employee Net Promoter Score (ENPS) as small and efficient metric.

Silence
: Whatever you ask, a huge warning sign is if people tell you nothing and just say everything is fine. All this change should be stressful and dramatic for people: a lack of problems is a troubling sign and a lack of people telling you about those problems means they don't trust you to give them a voice.

Losing people
: Most change leaders say some people aren't interested in the new culture, choosing to leave instead of change. Experiencing some staff departures is normal, and maybe even feedback that you're actually changing enough to ruffle their feathers.

Regaining people
: On the other hand, you might pick up more people on the other side of this, though. Dick's Jason Williams says that they've seen some former employees come back to their team; this is another good piece of feedback for how well you're managing your organization's cultural change.

The ultimate feedback, of course, will be whether you achieve the business goals derived from your strategy. But, you need to make sure that success isn't at the cost of incurring cultural debt that will come due in the future. This debt often comes due in the form of stressed out staff leaving or, worse, going silent and no longer telling you about what they're learning from failures. Then, you're back in

the same situation from which you were trying to escape all this digital transformation, an organization that's scared and static, rather than savvy and successful.

Giving Feedback

Product teams have several, frequent feedback mechanisms built into each release cycle and daily: unit tests, automated governance and policy tests, releases to production, and actual user feedback on its design choices. Teams typically, however, have little feedback from management that indicates how well they're adapting to the new culture. This is especially true of teams that are just starting to change.

This means that you'll more than likely need to change how and when you give feedback to teams and people in your organization. Traditional performance plans and reviews are a very poor place to give feedback and set expectations. They're the Waterfall equivalent of management, a 12-month cycle that too often fails to meet real needs.

Switching from traditional performance reviews and plans to tracking Objectives and Key Results (OKRs) is a popular tactic to improve the review process. An OKR states a very clear objective that's time-bound (usually to a quarter) and one or more metrics that measure success.[7] These OKRs are also shared with the relevant groups in the organization, allowing people to collaborate and partner on OKRs. Typically, OKRs are on a much shorter cycle than annual performance reviews, at the most a quarter, if not monthly.

Finally, because metrics are built into the OKR, each individual and team can self-monitor its progress and remove vague performance evaluations. No performance measurement is perfect, of course. Without a blameless culture that seeks to fix systemic problems instead of blaming individuals, OKRs can be equally abused when metrics aren't reached. However, as a tool, they fit well with a small-batch-minded organization.

You might need to improve how you give casual feedback, as well. A product-driven culture benefits from a generative mindset: an

7 For a much better overview and handbook for putting OKRs in place, see Christina Wodtke's book on the topic, *Introduction to OKRs* (O'Reilly).

approach that emphasizes learning and discovery, not perfection. Again, this is the groundwork for innovation. The words you use and the questions you ask can discourage or encourage an innovative culture. Clearly, removing blame-storming from conversations is key, as we discussed earlier.

What management says and asks has a huge effect on how staff behave. Linda Rising suggests several ways to improve how you talk with staff:

- Instead of praising people for their smarts, praise the effort they put in and the solution they discovered. You want to encourage their energy and the use of problem-solving strategies instead of relying on innate perfection or trusting a preset process. "How did you solve that bug?" you might say, rather than "you're so clever!"
- Instead of asking for just schedule progress, ask what the team has learned in recent releases. This demonstrates that getting better is the goal, not just achieving goals.
- If staff are discouraged, share your own stories of failing and learning. Also refocus them on the failings of the system you have in place and discuss what changes could be put in place to prevent future failures.

There are plenty more tactics for giving generative feedback to staff. Spend a sizable chunk of time thinking about, and then experimenting with, how to be not only encouraging in your conversations with staff, but also more caring. Among many other things, this will help build the trust and rapport needed to change old behaviors.

Managing ongoing culture change

How you react to this feedback is even more important than gathering the feedback. Just as you expect your product teams to go through a small-batch process, reacting to feedback from users, you should cycle through organizational improvement theories, paying close attention to the results.

Your ongoing small batching will also demonstrate to staff that you mean what you say, that you can be trusted, that you actually care about improving, and that it's safe for them to take risks. All this small batching will let you also participate in the blameless postmortems as you innovate.

Building Your Business Case

The mystic protocols of corporate budgeting are always situational. The important thing to know is when to start prewiring for the annual finance cycle, when it's due, and the culture of approval and revisions. If you don't know the budget cycles of your company, figure that out from your manager and other managers. Make sure that you're participating in the cycle instead of just being told what your budget is.

After that, there are two budgets to garden.

First, your initial proposal to transform and the expenses incurred like training, facilities, cloud, platform software, and other tools. This is often called a business case.

Second, and the one you'll spend the rest of your time on: the ongoing budget, or the ongoing business case. Hopefully you have business input about goals and expected returns. If the business side lacks firm financials or, worse, doesn't include you in those conversations, start worrying. The point of all this transformation suffering is to drive growth, to help the business side.

The Business Case

When you're pitching a new idea, you need to show that the costs and benefits will be worth it, that it will be profitable and fit within budget. For innovation-driven businesses, "budget" can be nebulous. Often, there's no exact budget specified, and the business case builds a rationale for how much money is needed. At the very least, it's often good to ask for more, backed up with proof that it might be a good idea.

A digital transformation business case is a tool for getting cash, but also a tool for discovering the work to be done and educating the organization about the goals of transformation.

First, you want to show all the costs involved: staff salary, cloud hosting costs or private cloud build-out expenses, training and consulting, changes to facilities and other capital expenses like laptops and chairs, and even snacks for all those ping pong balls and foosball tournaments. This is the baseline; the "before" set of numbers.

The second set of numbers you create are savings estimates. These estimates are based on the expected productivity gains you'll realize

by improving staff productivity, replacing aged platforms and middleware, reducing time spent fixing bugs and addressing security issues, and the overall reduction of waste in your organization.

Where do these estimates come from, though?

- You can use industry norms for improvement, although finding trustworthy ones can be difficult. Every vendor and proponent of "The New Way" will have numbers at the ready; but they, of course, are biased to prove that their solution results in the best numbers. With that vendor disclaimer in your mind, you could start with a Forrester study that Pivotal commissioned to baseline before and after improvements.

There are the costs of software licensing (or "subscriptions," to be technically correct, in most cases now) for platforms and tools as well as infrastructure costs. Some of these costs might seem high on their own, but compared to maintenance costs of older systems and lacking the savings from staff productivity, these new costs usually turn out to be worth it.

You need to account for the costs of transformation. This will be variable but should include things like training, consulting, initial services needed, and even facility changes.

With these numbers you have a comparison between doing nothing new and transforming. If the traditional way is cheaper, perhaps it's more efficient and you should not actually change. But, it must be cheaper *and* get you the productivity, time to market, and software design improvements, as well, not just be cheaper than improving. More than likely, your transformation case will be better.

If you're lucky enough to have business revenue projections, you can show how you contribute to revenue generation. Additionally, you should try to model some of the less tangible benefits. For example, because you'll be delivering software faster and more frequently, your time-to-market will decrease. This means that your organization could begin earning money sooner, affecting time-based budgeting calculations like net present value (NPV) and cash flows.

The small-batch feedback loop, which quickly corrects for incorrect requirements, will improve your risk profile. Delivering weekly also means that you'll begin putting value on the balance sheet sooner rather than, for example, delivering by batch in an annual release.

These "value soon, not later" effects might be useful, for example, when comparing internal rate of return (IRR) calculations to other options: your budget might be higher, but it could be less risky than even lower budgeted programs.

There are many other fun and baroque budgeting dances to do with Agile-driven software. For example, you could do real options analysis to put a dollar figure around the agility you'll gain. If you have the time and interest from finance to do such things, go ahead. For your initial business cases, however, that'll likely be too much work for the payoff. The business case on sophisticated business cases is often poor.

The Ongoing Business Case

As soon as you get rolling, your ongoing business case will adjust, sometimes dramatically and sometimes just incrementally. Organizations often change how they do budgets—hopefully not every year. But new executives, boards, and regulations might require new budget models. A private equity firm will have different expectations than the public markets, both of which will be much different than operating in startup mode.

In addition to creating high-quality and well-designed software, when calibrated over several months, the small-batch process should result in more stability and predictability. A calibrated product team knows how many stories it can release each week and is familiar with its ongoing costs. As you create more product teams by slowly scaling up your product-centric approach, you'll see similar calibrations which should, hopefully, be similar to the first few baselines.

This calibration can then be applied back to the budget to test the original assumptions behind the business case. Just as you can verify your theories about features with a small-batch approach, improving the software week to week, you can test your budgeting assumptions and correct them.

This means that feedback from each small-batch budgeting cycle adds more knowledge and gives finance more control. For finance, this is hopefully appealing because it adds more discipline into budgeting and addresses one of the primary responsibilities of the CFO: to ensure that money is spent wisely and responsibly. Even better, it puts a reliable process in place that ensures that money is well spent.

From the CFO's perspective, a small-batch budgeting will be a refreshing take on traditional IT budgets. Rather than wrangling over a large sum of money and then being given just one chance per year to course correct, the CFO will have more accurate estimates, transparency throughout the process, and the ability to adjust budgets up or down throughout the year.

Finance teams can also use these new controls to stop projects that are obviously failing, reducing losses and reallocating unspent budget to new programs. This might seem bad for you—the one getting less budget—which arguably it is if you work at a company that punishes such failure. But don't worry! There are always new organizations eager to hire ambitious, innovative people like yourself!

Gated Funding as a Defensive Tactic

Most people don't like living with "gated funding." This means that you're not given a lump annual sum, but instead have to prove yourself many times during the course of the year. That lump sum might be allocated, at least in a spreadsheet, but you're running the risk of not making it through the gates and increasing your chances of suffering from budget cuts.

But gated funding is often advantageous at first. In fact, it better matches the spirit of a small batch process. First, it's easier to get the cash for just one or two teams to try out your transformation program. In the context of the entire corporate budget, the amount needed is likely a "rounding error," allowing you to start the process of scaling up with a series of small projects and building up trust.

Rather than asking for a giant, multiyear bundle, asking for less might actually yield success—and that bigger budget—sooner. As Allstate's Opal Perry explains, "By the time you got permission, ideas died." But with a start-small approach, she contrasts, "A senior manager has $50,000 or $100,000 to do a minimum viable product," allowing them to get started sooner and prove out the new approach.

You still need to prove that the new methods work, of course. This is why, as discussed elsewhere, it's important to pick a small series of products to build up success and win over organizational trust with internal marketing.

Jon Osborn came across a second tactical benefit of gated funding: holding executive sappers at bay. Annual budgeting is a zero-sum game: there's a fixed pool of cash to allocate.[8] In the annual squabble for budgeting, most of your peers view your budget wins as budget they lost. Rolling over a bit by accepting gated funding can help muzzle some of your opponents in the budgeting dog fight.

Considering the Enterprise Architect

> *We had assumed that alignment would occur naturally because teams would view things from an enterprise-wide perspective rather than solely through the lens of their own team. But we've learned that this only happens in a mature organization, which we're still in the process of becoming.*
>
> —Ron van Kemenade, ING.

The enterprise architect's role in all of this deserves some special attention. Traditionally, in most large organizations, enterprise architects define the governance and shared technologies. They also enforce these practices, often through approval processes and review boards. An enterprise architect (EA) is seldom held in high regard by developers in traditional organizations. Teams (too) often see EAs as "enterprise astronauts," behind on current technology and methodology, meddling too much in day-to-day decisions, sucking up time with CABs, and forever working on tasks that are irrelevant to "the real work" done in product teams. Yet, although traditional EAs might do little of value for high-performing organizations, the role does play a significant part in cloud-native leadership.

First, and foremost, EAs are part of leadership, acting something like the engineer to the product manager on the leadership team. An EA should intimately know the current and historic state of the IT department, and also should have a firm grasp on the actual business IT supports.

Even though EAs are made fun of for ever-defining their enterprise architecture diagrams, that work is a side effect of meticulously keeping up with the various applications, services, systems, and dependencies in the organization. Keeping those diagrams up to

8 The pool can be widened by outside funding, but even then, your organization likely has a cap on the amount of outside funding it wants to take on.

date is a hopeless task, but the EAs who make them at least have some knowledge of your existing spaghetti of interdependent systems. As you clean up this bowl of noodles, EAs will have more insight into the overall system. Indeed, tidying up that wreckage is an underappreciated task.

Gardening the Organization

I like to think of the work EAs do as "gardening" the overall organization. This contrasts with the more top-down idea of defining and governing the organization, down to technologies and frameworks used by each team. Let's look at some of an EAs gardening tasks.

Setting technology and methodology defaults

Even if you take an extreme developer-friendly position, saying that you're not going to govern what's inside each application, there are still numerous points of governance about how the application is packaged and deployed, how it interfaces and integrates with other applications and services, how it should be instrumented to be managed, and so on. In large organizations, EAs should play a substantial role in setting these "defaults." There can be reasons to deviate, but these are the prescribed starting points.

As Pivotal's Stuart Charlton explains:

> I think that it's important that as you're doing this you do have to have some standards about providing a tap, or an interface, or something to be able to hook anything you're building into a broader analytics ecosystem called a data lake—or whatever you want to call it—that at least allows me to get at your data. It's not, you know, like, "hey I wrote this thing using a gRPC and golang and you can't get at my data!" No, you got to have something where people can get at it, at the very least.

Beyond software, EAs can also set the defaults for the organization's "meatware"—all the process, methodology, and other "code" that actual people execute. Before the Home Depot began standardizing its process, Tony McCully, the company's Senior Manager for Engineering Enablement, says, "everyone was trying to be agile and there was this very disjointed fragmented sort of approach to it… You know I joke that we know we had 40 scrum teams and we were doing it 25 different ways." Clearly, this is not ideal, and standardizing how your product teams operate is better.

It can seem constricting at first, but setting good defaults leads to good outcomes like Allstate reporting going from 20% developer productivity to more than 80%. As someone once quipped: they're called "best practices" because they are the best practices.

Gardening product teams

First, someone needs to define all the applications and services that all those product teams form around. At a small scale, the teams themselves can do this, but as you scale up to thousands of people and hundreds of teams, gathering together a *Star Wars*–scale Galactic Senate is folly. EAs are well suited to define the teams, often using DDD to first find and then form the domains and bounded contexts that define each team. A DDD analysis can turn quickly into its own crazy wall of boxes and arrows, of course. Hopefully, EAs can keep the lines as helpfully straight as possible.

Rather than checking in on how each team is operating, EAs should generally focus on the outcomes these teams have. Following the rule of team autonomy (described elsewhere in this report), EAs should regularly check on each team's outcomes to determine any modifications needed to the team structures. If things are going well, whatever's going on inside that black box must be working. Otherwise, the team might need help, or you might need to create new teams to keep the focus small enough to be effective.

Rather than policing compliance to rules and policy, an EA's work is more akin to the never ending, but pleasurable, activity of gardening.

Gardening microservices

Most cloud-native architectures use microservices; hopefully, to safely remove dependencies that can deadlock each team's progress as they wait for a service to update. At scale, it's worth defining how microservices work, as well, for example: are they event based, how is data passed between different services, how should service failure be handled, and how are services versioned?

Again, a senate of product teams can work at a small scale, but not on the galactic scale. EAs clearly have a role in establishing the guidance for how microservices are done and what type of policy is followed. As ever, this policy shouldn't be a straightjacket. The era of service-oriented architecture (SOA) and enterprise service buses

(ESBs) has left the industry suspicious of EAs defining services. Those systems became cumbersome and slow moving, not to mention expensive in both time and software licensing. We'll see if microservices avoid that fate, but keeping the overall system lightweight and nimble is clearly a gardening chore for which EAs are well suited.

Platform operations

As we discuss later, at the center of every cloud-native organization is a platform. This platform standardizes and centralizes the runtime environment, how software is packaged and deployed and how it's managed in production, and otherwise removes all the toil and sloppiness from traditional, bespoke enterprise application stacks. Most of the platform cases studies I've been using, for example, are from organizations using Pivotal Cloud Foundry.

Occasionally, EAs become the product managers for these platforms. The platform embodies the organization's actual enterprise architecture and developing the platform; thus evolves the architecture. Just as each product team orients its weekly software releases around helping its customers and users, the platform operations team runs the platform as a product.

EAs might also become involved with the tools groups that provide the build pipeline and other shared services and tools. Again, these tools embody part of the overall enterprise architecture—more of the running cogs behind all those boxes and arrows.

"They enable us," Discover's Dean Parke says, describing the enterprise architect's new roles. "They provide a lot of these core architectural libraries that we can utilize. They help out a lot with our CI/CD pipeline and making those base components available to us," he goes on.

As a side effect of product managing the platform and tools, EAs can establish and enforce governance. The packaging, integration, runtime, and other "opinions" expressed in the platform can be crafted to force policy compliance. That's a command-and-control way of putting it, and you certainly don't want your platform to be restrictive. Instead, by implementing the best possible service or tool, you're getting product teams to follow policy and best practices by bribing them with ease of use and toil reduction.

Think of it as a governance as code. EAs in this scheme "provide guardrails and enable the teams to move forward," Parke says, "so it's less of an oversight on how we should do design and governance model. There's still that there, obviously, but it's also more of a pushing forward of the architecture and enabling the teams."

The Shifting yet Never-Changing Role of the EA

I've highlighted just three areas EA contribute to in a cloud-native organization. There are more, many of which will depend on the peccadilloes of your organization; for example:

- Identifying and solving sticky cultural change issues is one such situational topic. EAs will often know individuals' histories and motivations, giving them insight into how to deal with grumps that want to stall change.
- EA groups are well positioned to track, test, and recommend new technologies and methodologies. This can become an "enterprise astronaut" task of being too far afield of actual needs and not understanding what teams need day to day, of course. But, coupled with being a product manager for the organization's platform, scouting out new technologies can be grounded in reality.
- EAs are well positioned to negotiate with external stakeholders and blockers. For example, as covered later, auditors often end up liking the new, small-batch and platform-driven approach to software because it affords more control and consistency. Someone needs to work with the auditors to demonstrate this and be prepared to attend endless meetings for which product team members are ill suited and ill tempered.

What I've found is that EAs do what they've always done. But, as with other roles, EAs are now equipped with better process and technology to do their jobs. They don't need to be forever struggling eyes in the sky and can actually get to the job of designing, refactoring, and programming the enterprise architecture. Done well, this architecture becomes a key asset for the organization—often *the* key asset of IT.

Though he poses it in terms of the CIO's responsibility, Mark Schwartz describes the goals of EAs well:

> The CIO is the enterprise architect and arbitrates the quality of the IT systems in the sense that they promote agility in the future. The systems could be filled with technical debt but, at any given moment, the sum of all the IT systems is an asset and has value in what it enables the company to do in the future. The value is not just in the architecture but also in the people and the processes. It's an intangible asset that determines the company's future revenues and costs and the CIO is responsible for ensuring the performance of that asset in the future.

Hopefully the idea of designing and then actually creating and gardening that enterprise asset is attractive to EAs. In most cases, it is. Like all technical people, they pine for the days when they actually wrote software. This is their chance to get back to it.

Tackling a Series of Small Projects

Every journey begins with a single step, according to Lao Tzu. What they don't tell you is that you need to pick your first step wisely. And there's also step two, and three, and then all of the $n + 1$ steps. Picking your initial project is important because you'll be learning the ropes of a new way of developing and running software and, hopefully, of running your business.

When it comes to scaling change, choosing your first project wisely is also important for internal marketing and momentum purposes. The smell of success is the best deodorant, so you want your initial project to be successful. And...if it's not, you quietly sweep it under the rug so that no one notices. Few things will ruin the introduction of a new way of operating into a large organization than initial failure. Following Larman's Law, the organization will do anything it can—consciously and unconsciously—to stop change. One sign of weakness early, and your cloud journey will be threatened by status quo zombies.

In contrast, let's look at how the series of small projects strategy played out at the US Air Force.

The US Air Force had been working for at least five years to modernize the 43 applications used in Central Air Operations Command, going through several hundreds of millions of dollars. These applications managed the daily air missions carried out by the United States and its allies throughout Iraq, Syria, Afghanistan, and nearby countries. No small task of import. The applications were in sore need of modernizing, and some weren't even really applica-

tions: the tanker refueling scheduling team used a combination of Excel spreadsheets and a whiteboard to plan the daily jet refueling missions.

Realizing that their standard 5- to 12-year cycle to create new applications wasn't going to cut it, the US Air Force decided to try something new: a truly Agile, small-batch approach. Within 120 days, a suitable version of the tanker refueling application was in production. The tanker team continued to release new features on a weekly, even daily basis. This created valuable cultural and trust changes between IT and its end users, as recounted in a paper that highlights the transformation:

> As functional capabilities were delivered weekly, warfighter confidence grew. And through regular weekly feedback sessions, the collaboration process became stronger. Each week, the warfighter received a suitable feature to complete the overall mission. Perhaps more importantly, the new version did not sacrifice capability.… This process enhanced the confidence of the warfighter in using the application and communicating additional feature requests. What's more, it established a rhythm for continuous fielding across the complete development, test, accredit and deploy cycle.

The project was considered a wild success: the time to make the tanker schedule was reduced from 8 hours to 2, from 8 airmen to 1, and the US Air Force ended up saving more than $200,000 per day in fuel that no longer needed to be flown around as backup for error in the schedule.

The success of this initial project, delivered in April of 2017 and called JIGSAW, proved that a new approach would work, and work well. This allowed the group driving that change at the US Air Force to start another project, and then another one, eventually getting to 13 projects in May of 2018 (5 in production and 8 in development). As of this writing, the team estimates that in January 2019, they should have 15 to 18 applications in production, as shown in Figure 2-1.

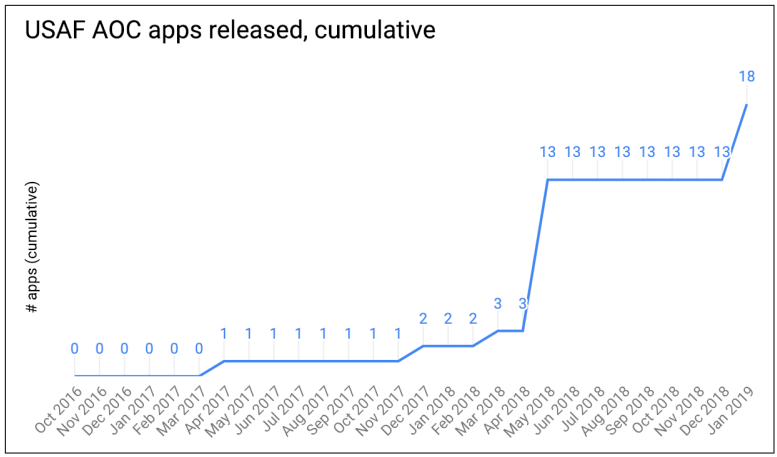

Figure 2-1. The number of US Air Force CAOC transformed applications over time, starting with 0 and ending with an estimated 18. (Sources from several US Air Force presentations and write-ups.)

The team's initial success, though just a small part of the overall 43 applications, gave them the momentum to begin scaling change to the rest of the organization and more applications.

Choosing Projects

Picking the right projects to begin with is key. Here are some key characteristics:

- They should be material to the business, but low risk.
- They should be small enough that you can quickly show success in the order of months.
- They need to be technically feasible for using cloud technologies.

These shouldn't be science projects or automation of low-value office activities—no augmented reality experiments or conference room schedulers (unless those are core to your business). On the other hand, you don't want to do something too big, like move the .com site. Christopher Tretina recounts Comcast's initial cloud native ambitions:

> We started out with a very grandiose vision... And it didn't take us too long to realize we had bitten off a little more than we could chew. So around mid-year, last year, we pivoted and really tried to

> hone in and focus on what were just the main services we wanted to deploy that'll get us the most benefit.

Your initial projects should also enable you to test out the entire software life cycle—all the way from conception to coding to deployment to running in production. Learning is a key goal of these initial projects, and you'll do that only by going through the full cycle.

The Home Depot's Anthony McCulley describes the applications his company chose in the first six or so months of its cloud-native rollout. "They were real apps. I would just say that they were just, sort of, scoped in such a way that if there was something wrong, it wouldn't impact an entire business line." In the Home Depot's case, the applications were projects like managing (and charging for!) late tool rental returns and running the in-store, custom paint desk.

Microservices: A Special Case for Choosing Initial Projects

A special case for initial projects is picking a microservice to deploy. Usually, such a service is a critical backend service for another application. A service that's taken forever to actually deliver or has been unchanged and ancient for years is an impactful choice. This is not as perfect a use case as a full-on, human-facing project, but it will allow you to test out cloud-native principals and rack up a success to build momentum. The microservice could be something like a fraud detection or address canonicalization service. Citi, for example chose a payment validation service as one of its initial projects: the perfect mix of size and business value to begin with. This is one approach to moving legacy applications in reverse order, a strangler[9] from within!

Picking Projects by Portfolio Pondering

There are several ways to select your initial projects. Many Pivotal customers use a method perfected over the past 25 years by Pivotal Labs called *discovery*. In the abstract, it follows the usual Boston Consulting Group (BCG) matrix approach, flavored with some

9 The strangler pattern and how it's applied to moving legacy services is covered in "A Note on Labs and Legacy Organizations" on page 57.

Eisenhower matrix. This method builds in intentional scrappiness to do a portfolio analysis with the limited time you can secure from all of the stakeholders. The goal is to get a ranked list of projects based on your organization's priorities and the easiness of the projects.

First, gather all of the relevant stakeholders. This should include a mix of people from the business and IT sides as well as the actual team that will be doing the initial projects. A discovery session is typically led by a facilitator, preferably someone familiar with coaxing a room through this process.

The facilitator typically hands out stacks of sticky notes and markers, asking everyone to write down projects that they think are valuable. What "valuable" means will depend on each stakeholder. We'd hope that the more business minded of them would have a list of corporate initiatives and goals in their heads (or a more formal one they brought to the meeting). One approach used in Lean methodology is to ask management this question: "If we could do one thing better, what would it be?"[10] Start from there, maybe with some five-whys spelunking.

After the stakeholders have listed projects on their sticky notes, the discovery process facilitator draws or tapes up a 2x2 matrix that looks like the one shown in Figure 2-2.

Participants then place their sticky notes in one of the quadrants; they're not allowed to weasel out and put the notes on the lines. When everyone finishes, you get a good sense of projects that all stakeholders think are important, sorted by the criteria I mentioned, primarily that they're material to the business (important) and low risk (easy). If all of the notes are clustered in one quadrant (usually, in the upper right, of course), the facilitator will redo the 2x2 lines to just that quadrant, forcing the decision of narrowing down to just projects to do now. The process might repeat itself over several rounds. To enforce project ranking, you might also use techniques like *dot voting*, which will force the participants to really think about how they would prioritize the projects, given limited resources.

10 This is based on the answer to a question that I asked Jeffrey Liker at the 2016 Agile and Beyond conference, related to how lean manufacturing organizations choose which products to build that, in some sense, define their strategy.

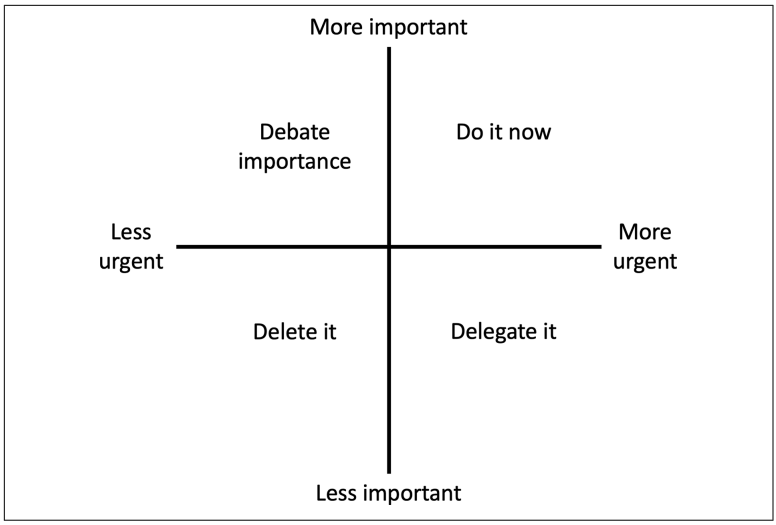

Figure 2-2. A 2x2 decision-making matrix that can help you to sort through your portfolio when choosing projects.

At the end, you should have a list of projects, ranked by the consensus of the stakeholders in the room.[11]

Planning Out the Initial Project

You might want to refine your list even more, but to get moving, pick the top project and start breaking down what to do next. How you proceed to do this is highly dependent on how your product teams breaks down tasks into stories,[12] iterations, and releases. More than likely, following the general idea of a small-batch process, you'll do the following:

11 Although written to help organizations pick advanced analytics projects to pursue, BCG has a good article that captures this general type of thinking, especially with respect to how leadership can think through the strategic parts of this process.

12 Whether you use "stories" or not, you'll have some unit of work, be they use cases, requirements, or what have you. Stories have emerged as one of the more popular and proven useful ways to encapsulating these units of work with an extensive body of work and tools to support their creation and management. See also a description of using stories with Pivotal Tracker.

1. Create an understanding of the user(s) and the challenges they're trying to solve with your software through personas and approaches like scenarios or Jobs to be Done.
2. Come up with several theories for how those problems could be solved.
3. Distill the work to code and test your theories into stories.
4. Add in more stories for nonfunctional requirements (like setting up build processes, CI/CD pipelines, testing automation, etc.).
5. Arrange stories into iteration-sized chunks without planning too far ahead (lest you're not able to adapt your work to the user experience and productivity findings from each iteration).

Starting small ensures steady learning and helps to contain the risk of a fail-fast approach. But as you learn the cloud-native approach better and build up a series of successful projects, you should expect to ramp up quickly. Figure 2-3 shows the Home Depot's ramp up in the first year.

The chart measures application instances in Pivotal Cloud Foundry, which does not map exactly to a single application. As of December 2016, the Home Depot had roughly 130 applications deployed in Pivotal Cloud Foundry. What's important is the general shape and acceleration of the curve. By starting small, with real applications, the Home Depot gained experience with the new process and at the same time delivered meaningful results that helped it scale its transformation.

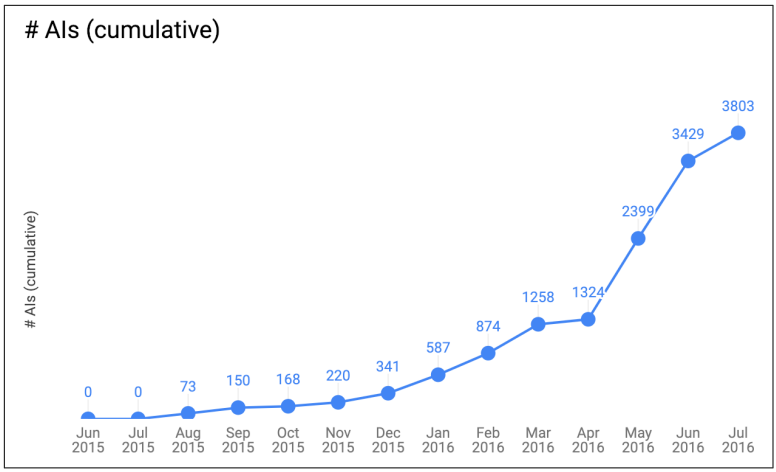

Figure 2-3. This chart shows the number of application instances, which is not 1:1 to applications. The end point represents about 130 applications, composed of about 900 services. (Sources: "From 0 to 1000 Apps: The First Year of Cloud Foundry at The Home Depot," Anthony McCulley, The Home Depot, Aug 2016. "Cloud Native at The Home Depot, with Tony McCulley," Pivotal Conversations #45.)

Assemble the Right Team

> *I met a programmer with 10x productivity once. He was a senior person and required 10 programmers to clean up his brilliant changes.*
>
> —Anonymous on the c2 wiki.

Skilled, experienced team members are obviously valuable and can temper the risk failure by quickly delivering software. Everyone would like the mythical 10x developer, and would even settle for a 3 to 4x "full-stack developer." Surely, management often thinks doing something as earth shattering as "digital transformation" only works with highly skilled developers. You see this in surveys all the time: people say that lack of skills is a popular barrier to improving their organization's software capabilities.

This mindset is one of the first barriers to scaling change. Often, an initial, team of "rockstars"[13] has initial success, but attempts to clone them predictably fails and scaling up change is stymied. It's that "lack of skills" chimera again. It's impossible to replicate these people, and companies rarely want to spend the time and money to actually train existing staff.

Worse, when you use the "only ninjas need apply tactic," the rest of the organization loses faith that they could change, as well. "When your project is successful," Jon Osborn explains, "and they look at your team, and they see a whole bunch of rockstars on it, then the excuse comes out, 'well, you took all the top developers, of course you were successful.'"

Instead of recruiting only elite developers (whatever that bravado diction means), also staff your initial teams with a meaningful dose of regular developers. This will not only help win over the rest of the organization as you scale, but also means that you can actually find people. A team with mixed skill levels also allows you train your "junior" people on the job, especially when they pair with your so-called rockstars.

Volunteers

When possible, recruiting volunteers is the best option for your initial projects, probably for the first year. Forcing people to change how they work is a recipe for failure, especially at first. You'll need motivated people who are interested in change or, at least, will go along with it instead of resisting it.

Osborn describes this tactic at Great American Insurance Company:

> We used the volunteer model because we wanted excited people who wanted to change, who wanted to be there, and who wanted to do it. I was lucky that we could get people from all over the IT organization, operations included, on the team...it was a fantastic success for us.

This might be difficult at first, but as a leader of change, you need to start finding and cultivating these change-ready volunteers. Again,

13 Of course, "rockstars" are often temperamental and a handful. They have a bad habit of destroying hotel rooms and themselves. That label is also often situational and the result of a culture that awards the lone wolf hero instead of staff that helps and supports one another.

you don't necessarily want rockstars so much as open-minded people who enjoy trying new things.

Sadly, some of your staff won't make the transition. You can often find ways of motivating them to get genuinely excited, but some people resist change to the end. Over the years, executives I've talked with say that anywhere between 30 to 70% of staff will fall into this well of obstinacy. "You will lose people, not everybody's gonna be on board," Dick's Sporting Good's Jason Williams says. "You will have some attrition but one of the key things that's happened to us recently, we've actually had engineers that have come back to us and rejoined our organization because they like the way we're going."

Of course, this problem isn't confined to staff. Management and leadership can be just as resistant, as discussed in "Communicating the Vision and Strategy" on page 22.

Rotating Teams to Spread Digital Transformation

Few organizations have the time or budget-will to train their staff. Management seems to think that a moist bedding of O'Reilly books in a developer's dark room will suddenly pop up genius skills like mushrooms. Rotating pairing in product teams addresses this problem in a minimally viable way within a team: team members learn from one another on a daily basis. Event better, staff is actually producing value as they learn instead of sitting in a neon-light buzzing conference room working on dummy applications.

To scale this change, you can selectively rotate staff out of a well-functioning team into newer teams. This seeds their expertise through the organization, and when you repeat this over and over, knowledge will spread faster. One person will work with another, becoming two skilled people, who each work with another person, become four skilled people, and then eight, and so on. Organizations like Synchrony go so far as the randomly shuffle desks every six months to ensure that people are moving around. Discover has institutionalized this practice with a dedicated group that works with a new team for six weeks to experience the new culture, sending them back into the organization to seed those new skills.

More than just skill transfer and on-the-job training, rotating other staff through your organization will help spread trust in the new process. People tend to trust their peers more than leaders throwing

down change from high, and much more than external "consultants" and, worse, vendor shills like myself. As ever, building this trust through the organization is key to scaling change.

Orange France is one of the many examples of this strategy in practice. After the initial success revitalizing its server message block (SMB) customer service app, Orange began rotating developers to new teams. Developers who worked on the new mobile application pair with Orange developers from other teams, the website team. As ever with pairing, they both teach their peers how to apply Agile and improve the business with better software at the same time. Talking about his experience with rotating pairing, Orange's Xavier Perret says that "it enabled more creativity in the end. Because then you have different angles, [a] different point of view. As staff work on new parts of the system, they get to know the big picture better and engage in in "more creative problem solving" to each new challenge, Perret adds.

Even though you might start with ninjas, you can take a cadre of volunteers and slowly but surely build up a squad of effective staff that can spread transformation throughout your organization. All with less throwing stars and trashed hotel rooms than those 10x rockstars leave in their wake.

A Note on Labs and Legacy Organizations

At this point, you might be thinking that changing a large, existing organization sounds difficult. Well, it is! People and processes are in place, and often no one wants to change themselves or revise those processes. To get around this, many organizations create new groups, usually called "labs." Product by product, and team by team, new software and people are moved to the new group. Volunteers are accepted in the initial stages (remember: team composition matters), and only favorable products are picked. As news of teams' successes spreads, this new organization is given corporate permission to take on new products and reluctant staff are more trusting that changing their work habits is a good idea.

The existing—now "legacy"—organization continues to operate as needed, but projects and people are slowly moved to the new organization. Service dependencies from new to old are mediated and hidden behind APIs and facades, trying to decouple the two into a

sort of reverse-quarantine: the old organization is blocked off from infecting the new group.

The new organization follows all the new-fangled notions and norms, from the pedestrian practice of free lunches and massages to paired programming to fully automated build pipelines—all enabling the magic of small batches that result in better software.

Synchrony Financial, for example, used this approach to shift from a functional to a product organization. The trick here is to begin with a blank slate across the board, including organization structure, governance, culture, and tools used. Getting to that point takes high-level support; in Synchrony's case, support came directly from the CEO. You don't always need that high a level of support, but if things go well, it can be easier to climb up the authority ladder to ask for that support. What's important is to somehow carve out the permission and space for a new organization, no matter how small, and put in place a strategy to grow that organization.

The magic of this method is that it avoids having to unfreeze the glacier, namely, the people who don't want to work in a new way. Instead of doing the difficult work of changing the old organization, management slowly moves over willing people, reassembling them into new teams and reporting structures. The old organization, though perhaps de-peopled, is left to follow its Waterfall wont.

After some initial success, you might even consider naming and branding the new organization to give the people in it a sense of identity. Allstate did this with its CompoZed Labs, creating a new logo and name. As of 2017, CompoZed was handling 40% of Allstate's software development, following the new, small-batch approach to software.

The important thing to remember is that even people and processes that seem immutable *can* be changed. It's just going to take planning, patience, and finesse.

Building Trust with Internal Marketing, Large and Small

As you build up initial successes, internal marketing is a key tool for scaling up digital transformation. And it's a lot more than most people anticipate.

Beyond Newsletters

After you nail down some initial successful applications, start a program to tell the rest of the organization about these projects. This is beyond the usual email newsletter mention. Instead, your initial content-driven marketing should quickly evolve to internal "summits" with speakers from your organization going over lessons learned and advice for starting new cloud-native projects.

You have to promote your change, educate, and overall "sell" it to people who either don't care, don't know, or are resistant. These events can piggyback on whatever monthly "brown-bag" sessions you have and should be recorded for those who can't attend. Somewhat early on, management should set aside time and budget to have more organized summits. You could, for example, do a half-day event with three to four talks by team members from successful projects, having them walk through the project's history and provide advice on getting started.

My colleagues and I are often asked to speak at these events, and we of course delight in doing so. However, don't be too reliant on external speakers: there's only so much trust a smooth talking, self-deprecating vendor like me can instill in an organization. Make sure you have speakers who work at your company. Fellow employees, especially peers in the career hierarchy, are very powerful agents for change on the dais.

Fostering Trust

This internal marketing works hand in hand with starting small and building up a collection of successful projects. As you're working on these initial projects, spend time to document the "case studies" of what worked and didn't work, and track the actual business metrics to demonstrate how the software helped your organization. You don't so much want to just show how fast you can now move, but you want to show how doing software in this new way is strategic for the business as well as individuals' compensation, career, reputation, and personal happiness. You know: motivate them.

Content-wise, what's key in this process is for staff to talk with one another about your organization's own software, market, and challenges faced. This is a good chance to kick up that Kotterian "sense of urgency," as well, if you think things have become too languid.

Also, you want to demonstrate that change is possible, despite how intractable people might think the status quo is. I find that organizations often think that they face unique challenges. Each organization does have unique hang-ups and capabilities, so people in those organizations tend to be most interested in how they can apply the wonders of cloud native to their jobs, regardless of whatever success they might hear about at conferences or, worse, vendors with an obvious bias (that would be me!). Hearing from one another often passes beyond this sentiment that "change can't happen here."

After your organization begins hearing about these successes, you'll be able to break down some of the objections that stop the spread of positive change. As Amy Patton at SPS Commerce put it, "Having enough wins, like that, really helped us to keep the momentum going while we were having a culture change like DevOps."

Winning Over Process Stakeholders

The IRS provides an example of using release meetings to slowly win over resistant middle-management staff and stakeholders. Stakeholders felt uncomfortable letting their usually detailed requirements evolve over each iteration. As with most people who are forced—er, "encouraged—to move from Waterfall to Agile, they were skeptical that the final software would have all the features they initially wanted.

While the team was, of course, verifying these evolving requirements with actual, in-production user testing, stakeholders were uncomfortable. These skeptics were used to comforting, thick upfront analysis and requirements, exactly spelling out which features would be implemented. To begin getting over this skepticism, the team used its release meetings to show off how well the process was working, demonstrating working code and lessons learned along the way. These meetings went from five skeptics to packed, standing-room-only meetings with more than 45 attendees. As success was built up and the organizational grapevine filled with tales of wins, interest grew as well as trust in the new system.

The Next Step: Training by Doing

As the organizations I've mentioned earlier and others like Verizon and Target demonstrate, internal marketing must be done "in the

small," like this IRS case and, eventually, "in the large" with internal summits.

Scaling up from marketing activities is often done with intensive, hands-on training workshops called *dojos*. These are highly structured, guided, but real release cycles that give participants the chance to learn the technologies and styles of development. And because they're working on actual software, you're delivering business value along the way: it's training and doing.

These sessions also enable the organization to learn the new pace and patterns of cloud-native development, as well as set management expectations. As Verizon's Ross Clanton put it recently:

> The purpose of the dojo is learning, and we prioritize that over everything else. That means you have to slow down to speed up. Over the six weeks, they will not speed up. But they will get faster as an outcome of the process.

Scaling up any change to a large organization is mostly done by winning over the trust of people in that organization, from individual contributors, to middle-management, to leadership. Because IT has been so untrustworthy for so many decades—how often are projects not only late and over budget, but then anemic and lame when finally delivered?—the best way to win over that trust is to actually learn by doing and then market that success relentlessly.

Tracking Your Improvement with Metrics

Tracking the health of your overall innovation machine can be both overly simplified and overly complex. What you want to measure is how well you're doing at software development and delivery as it relates to improving your organization's goals. You'll use these metrics to track how your organization is doing at any given time and, when things go wrong, get a sense of what needs to be fixed. As ever with management, you can look at this as a part of putting a small batch process in place: coming up with theories for how to solve your problems and verifying if the theory worked in practice or not.

Monitoring

In IT, most of the metrics you encounter are not actually business oriented and instead tell you about the health of your various IT systems and processes: how many nodes are left in a cluster, how much

network traffic customers are bringing in, how many open bugs development has, or how many open tickets the help desk is dealing with on average.

All of these metrics can be valuable, just as all of them can be worthless in any given context. Most of these technical metrics, coupled with ample logs, are needed to diagnose problems as they come and go. In recent years, there have been many advances in end-to-end tracing thanks to tools like Zipkin and Spring Sleuth. Log management is well into its newest wave of improvements, and monitoring and IT management analytics are just ascending another cycle of innovation—they call it "observability" now; that way, you know it's different this time!

Instead of looking at all of these technical metrics,[14] I want to look at a few common metrics that come up over and over again in organizations that are improving their software capabilities.

Six Common Cloud-Native Metrics

Certain metrics come up consistently when measuring cloud-native organizations. Let's take a look at each of them.

Lead time

Lead time is how long it takes to go from an idea to running code in production; it measures how long your small-batch loop takes. It includes everything in between, from specifying the idea, writing the code and testing it, passing any governance and compliance needs, planning for deployment and management, to getting it up and running in production, as shown in Figure 2-4.

14 To get more into technical metrics, see Brendan Gregg's discussion of the USE method, a deep, but brief start on health metrics. Also, check out the relevant chapter in the Google SRE book and Steve Mushero's overview of how to exactly gather SRE-type metrics. Finally, the Pivotal Cloud Foundry monitoring documents will give you a good idea for cloud-native platform metrics.

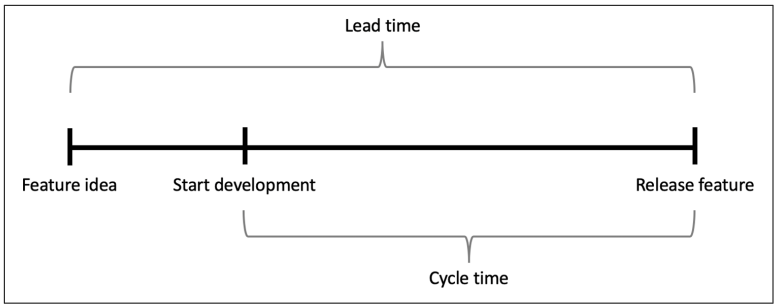

Figure 2-4. This diagram shows the elements that you need to consider when calculating lead time.

If your lead time is consistent enough, you have a grip on IT's capability to help the business by creating and deploying new software and features. As such, you want to monitor your lead closely. Ideally, it should be a week. Some organizations go longer, up to two weeks, and some are even shorter, like daily. Target and then track an interval that makes sense for you. If you see your lead time growing, you should stop everything, find the bottlenecks, and fix them. If the bottlenecks can't be fixed, you probably need to do less each release.

Value-stream mapping is a common tool for measuring lead time. A value-stream map shows all of the activities needed, from thinking of a new feature to a person using that feature. Functional organizations often focus on just one group's contribution to that overall task, which too often leads to locally optimizing each part of the value stream instead of the whole.

A value-stream map will also help you find wasted time in your release cycle, helping you to reduce lead time. "Waste" can be vague, but in general anything that the end user doesn't value or care about is waste. You can't always get rid of waste, but you should try to remove as much as possible. For example, using a value-stream map, one of Daimler's product teams identified time spent ordering, configuring, and deploying servers as waste. People walking into a Mercedes-Benz dealership rarely started the conversation with, "Boy, I want to get a Mercedes because you provision servers so well!" After this was identified, the team prioritized automating this and other waste in its value stream and brought its lead time down from 30 days to 3 days.

Velocity

Velocity shows how many features are typically deployed each release, for example, weekly. Whether you call features "stories," "story points," "requirements," or whatever else, you want to measure how many the team can complete each week; I'll use the term "story." Velocity is used to create realistic schedule and roadmap expectations and then monitor the health of the team's ability to deliver each week.

After you establish how many features a team can deliver each week, you can plan more reliability. If a team can deliver, for example, only 3 stories each week, asking it to deliver 20 stories in a month is absurd. The team simply is not capable of doing that. Ideally, this means that your estimates are no longer, well, always wrong. In most cases, organizations find reliability far more valuable than being lied to (knowingly or not) about how much work can actually be done.

Tracking velocity over time also gives you an early warning sign that something's wrong. If the team's velocity begins to drop, it means that the team is scoping its stories incorrectly: it's taking on too much work or someone is forcing the team to do so. On the other hand, if the team is suddenly able to deliver more stories each week or finds itself with lots of extra time each week, this means that it should take on more stories each week.

There are numerous ways to calibrate the number of stories a team can deliver each week, and managing that process at first is very important. The best method is to simply track how many stories are released each week for a month or so and use that number to calibrate. During this calibration process, your teams will, no doubt, get it wrong for many releases, which is to be expected (and will provide motivation for picking small projects at first instead of big, important ones). Other reports like burn-down charts can help illustrate progress toward major milestones and help you monitor any deviation from what's normal.

Latency

Speed drives a large part of customer satisfaction. Few people consider slow software good software. It's tempting to just measure the raw speeds of various components, but thinking about speed in terms of the end user is better. Latency measures just that, continu-

ally measuring how long a request takes, end to end, to process and return it back to the user.

Latency is different than the raw "speed" of the network. For example, a fast network will send a static file very quickly, but if the request requires connecting to a database to create and then retrieve a custom view of last week's Austrian sales, it will take a while and, thus, the latency will be much longer than downloading an already made file.

From a user's perspective, latency is important because an application that takes three minutes to respond versus three milliseconds might as well be "unavailable." As such, latency is often the best way to measure whether your software is working. Latency won't specify what's wrong with your application, but it will very quickly alert you that *something* is wrong, kicking off further analysis.

Measuring latency can be tricky...or really simple. Because it spans the entire transaction, you often need to rely on patching together a full view, or "trace," of any given user transaction. You can do this by looking at locks, doing real or synthetic user-centric tracing, and using any number of application performance monitoring (APM) tools. Ideally, the platform you're using will automatically monitor all user requests and also catalog all of the subprocesses and sub-subprocesses that make up the entire request. That way, you can begin to determine *why* things are so slow.

Error rates

Your systems and software will report when there's an error: an exception is thrown in the application layer because the email service is missing; an authentication service is unreachable so the user can't login; a disk is failing to write data. In addition to production errors, also pay attention to error rates in development and testing. You can look at failed tests as error rates as well as broken builds and failed compliance audits. In production and development, looking at ongoing error rates will give you a quick health check and show you where to begin drilling down and troubleshooting.

Fixing errors in development can be easier and more straightforward, whereas triaging and sorting through errors in production is an art. What's important to track with errors is not just that they happened, but the rate at which they happen, perhaps as errors per second. You'll need to establish an acceptable level of errors because

there will be many of them. What you do about all these errors will be driven by your service targets. These targets might be foisted on you in the form of heritage Service-Level Agreements (SLAs) or you might have been lucky enough to negotiate some sane targets à la SREs.

Chances are, a certain rate of errors will be acceptable. (Have you ever noticed that sometimes, you just need to reload a web page?) Each part of your stack will throw off and generate different errors that you should handle differently:

- Errors can be meaningless. For example, warnings about using deprecated code or the inability to print framework banners in the logs. Perhaps those errors should be reclassified as a lower priority or, better still, turned off altogether.
- Errors can be too costly or even impossible to fix. For example, "1% of user's audio uploads fail because their upload latency and bandwidth are too slow." You have no control over the user's internet connection, so you can't do much to improve it.
- Errors can be important above all else, demanding all of your attention. If an email server is losing emails every five minutes, something is terribly wrong.

Ideally, developers will write good error detection, logging, and even remediation. There's a reason I used the word "ideally" there: developers are rarely this meticulous. If you're putting orthodox DevOps in place, your developers will be responsible for diagnosing and remediating problems in production. At this point, they'll be the ones who suffer from their bad error reporting, hopefully leading to much better error reporting and even remediation into their software.

Mean-time-to-repair

Your software will break, there's no way around it. Tracking how often it breaks is important, but tracking how quickly you can fix those problems is a key metric. It's bad when an error happens, but it's really bad if it takes you a long time to fix it. Tracking mean-time-to-repair (MTTR) is the ongoing measurement of how quickly you can recover from errors. As with most metrics, as you establish an acceptable baseline, you can monitor MTTR to ensure that the problem is not getting worse.

If you're following cloud-native practices and using a good platform, you can usually shrink your MTTR by adding the ability to roll back changes. If a release goes bad, you can back it out quickly, remove the problem, and restore the previous working version of the software.

Measuring MTTR might require tracking support tickets and otherwise manually tracking the time between incident detection and fix. As you automate remediations, you might be able to easily capture those rates. What's important in the long term is tracking changes to your acceptable MTTR and identifying why negative changes are happening.

Costs

Everyone wants to measure cost, and there are many costs to measure. In addition to the time spent developing software and the money spent on infrastructure, there are ratios that you'll want to track, like the number of applications to platform operators. Typically, these kinds of ratios give you a quick sense of how efficiently IT runs. If each application takes one operator, something is probably missing from your platform and process. T-Mobile, for example, manages 11,000 containers in production with just 8 platform operators.

Costs are typically measured in terms of staff time; infrastructure and networking; and any tools, software licenses, and subscriptions needed. There are also less-direct costs like opportunity and value lost due to waiting on slow release cycles. You need to obviously track the direct costs of what you're doing, but you also need to track the costs of doing nothing, which might be much higher.

Business Value

Of course, none of the metrics so far has measured the most valuable but difficult metric: value delivered. How do you measure your software's contribution to your organization's goals? Measuring how the process and tools you use contribute to those goals is usually more difficult.

Somehow, you need to come up with a scheme that shows and tracks how all this cloud-native stuff you're spending time and money on is helping the business grow. You want to measure value delivered over time to do the following:

- Prove that you're valuable and should keep living and get more funding.
- Recognize when you're failing to deliver so that you can fix it.

There are a few prototypes of linking cloud-native activities to business value delivered. Let's look at a few examples:

- As described in the earlier case study of when the IRS replaced call centers providing limited availability with software, IT delivered clear business value. Latency and error rates decreased dramatically (with phone banks, only 37% of calls made it through) and the design improvements it discovered led to increased usage of the software, pulling people away from the phones. The business value was clear: by the fall of 2017, this application had collected $440 million in back taxes.
- Running existing businesses more efficiently is a popular goal, especially for large organizations. In this case, the value you deliver with cloud native will usually be speeding up businesses' processes, removing wasted time and effort, and increasing quality. Duke Energy's line worker case is a good example here. Duke gave line workers a better, highly tuned application that queues and coordinates field work. The software increased line workers' productivity and reduced waste, directly creating business value in efficiencies.
- The US Air Force's tanker scheduling case study is another good example here: by improving its software capabilities, the US Air Force was able to ship the first version in 120 days and began saving $100,000's in fuel costs each week. Accuracy and speed in scheduling delivered clear business value, as well.
- Then, of course, there comes raw competition. This most easily manifests itself as time-to-market, either to match competitors or get new features out before them. Liberty Mutual's ability to enter the Australian motorcycle market from scratch in only six months is a good example. Others such as Comcast demonstrate competing with major disruptors like Netflix.

It's easy to become very nuanced and detailed when you're mapping IT to business value. To avoid spiraling into a nuance vortex, management needs to keep things as simple as possible, or, put another way, only as complex as needed. As with the previous example,

clearly link your cloud-native efforts to straight forward business goals. Simply "delivering on our commitment to innovation" isn't going to cut it. If you're suffering under vague strategic goals, make them more concrete before you begin using them to measure yourself. On the other end, just lowering costs might be a bad goal to shoot for. I talk with many organizations who used outsourcing to deliver on the strategic goal of lowering costs and now find themselves incapable of creating software at the pace their business needs to compete.

Fleshing Out Metrics

The preceding sections provided a simplistic start at metrics. Each layer of your organization will want to add more detail to get better insights into itself. Creating a comprehensive, umbrella metrics system is impossible, but there are many good templates with which you can start. Platform operators, for example, should probably begin by learning how the Google SRE team measures and manages Google.

The five S's

Pivotal's five S's is one such template. The S's stand for speed, stability, scalability, security, and savings. These metrics cover platform operations, product, and business metrics. Not all organizations will want to use all of the metrics, and there's usually some that are missing. But, this five S's template is a good place to begin. You might not need all the metrics, and there might be others not included. Nonetheless, it's good to have something more detailed than "it depends" when it comes to metrics.

IT metrics

Speed
: Latency, number of deploys to production, number of bugs, bug-fix duration, new release duration, number of code check-ins, number of tickets, time-to-deployment

Stability
: MTTR, platform failures, release failures, error rates, number of incidents, avg. release time, minutes of development, production outages per year

Scalability
: Transaction throughput, transaction latency, container scaling, CPU usage, memory usage, disk capacity

Security
: Time to patch, downtime due to security, security VSM time

Savings
: Number of tickets, dollar savings, time-to-resolution-gain, increase in deployments, number of bugs

Business metrics

Speed
: Time-to-market, lead time, deployment frequency (velocity)

Stability
: Volatility, number of bugs/dev/year, change fail rate, velocity

Scalability
: Products in development versus production, dev-to-ops spend, hiring and staff churn rates, resource elasticity

Security
: Number of incidents, percent using CI/CD, production patch frequency

Savings
: VSM waste, products-to-dev, number of design failures, dev-to-ops spend, legacy middleware licensing, cost of delay

Metrics' Utility

Whatever the case, make sure the metrics you choose are 1) targeting the end goal of putting in place a small-batch process to create better software, 2) reporting on your ongoing improvement toward that goal, and 3) alerting you that you're slipping and need to begin addressing the problem before it becomes worse.

Metrics are not only useful for day-to-day operations. You can also start using them to demonstrate to the rest of the organization that your transformation strategy is working and can be trusted. Showing, for example, that your team's velocity is reliable and stable will demonstrate a level of professionalism that's usually not expected by teams that have historically delivered late and over budget. When it

comes time to ask for more budget, reporting on business value delivered will be most helpful.

Tending to Compliance and Regulation

"Compliance" will be one of your top bugbears as you improve how your organization does software. As numerous organizations have been finding, however, compliance is a solvable problem. In most cases, you can even *improve* the quality of compliance and risk management with your new processes and tools, introducing more, reliable controls than traditional approaches.

But First, What Exactly Is "Compliance"?

If you're a large organization, the chances are that you'll have a set of regulations with which you need to comply. These are both self- and government-imposed. In software, the point of regulations is often to govern the creation of software, how it's managed and run in production, and how data is handled. The point of most compliance is risk management; for example, making sure developers deliver what was asked for, making sure they follow protocol for tracking changes and who made them, making sure the code and the infrastructure is secure, and making sure that people's personal data is not needlessly exposed.[15]

Compliance often takes the form of a checklist of controls and verifications that must be passed. Auditors are staff that go through the process of establishing those lists, tracking down their status in your software, and also negotiating whether each control must be followed. The auditors are often involved before and after the process to establish the controls and then verify that they were followed. It's rare that auditors are involved during the process, which unfortunately ends up creating more wasted time, it turns out. Getting auditors involved after your software has been created requires much compliance archaeology and, sadly, much cutting and pasting between emails and spreadsheets, paired with infinite meeting scheduling.

15 This is a shallow definition of compliance, there are numerous, better explanations including Dmitry Didovicher's overview in a recent talk. Also, of note, is that I don't cover a huge swatch of compliance woes: project management compliance. That's a deep well.

When you're looking to transform your software capabilities, these traditional approaches to compliance, however, often end up hurting businesses more than helping them. As Liberty Mutual's David Ehringer describes it:

> The nature of the risk affecting the business is actually quite different: the nature of that risk is, kind of, the business disrupted, the business disappearing, the business not being able to react fast enough and change fast enough. So, this is not to say that some of those things aren't still important, but the nature of that risk is changing.

Ehringer says that many compliance controls are still important, but there are better ways of handling them without worsening the largest risk: going out of business because innovation was too late.

Avoiding *that* risk requires tweaking your compliance processes. I've seen three approaches to dealing with compliance, often used together as a sort of maturity model: compliance unchained, minimum viable compliance, and transform compliance. Let's take a look at each.

Compliance Unchained

Although it's just a quick fix, engineering a way to avoid compliance is a common first approach. Early on, when you're learning a new mindset for software and build up a series of small successes, you'll likely work on applications that require little to no compliance. These kinds of applications often contain no customer data, don't directly drive or modify core processes, or otherwise touch anything that'd need compliance scrutiny.

These might seem disconnected from anything that matters and thus not worth working on. Early on, though, the ability to get moving and prove that change is possible often trumps any business value concerns. You don't want to eat these "empty calories" projects too much, but it's better than being killed off at the start.

Minimal Viable Compliance

Part of what makes compliance seem like toil is that many of the controls seem irrelevant. Over the years, compliance builds up like plaque in your steak-loving arteries. The various controls might have made sense at some time—often responding to some crisis that occurred because this new control wasn't followed. At other times,

the controls simply might not be relevant to the way you're doing software. If that's the case, you'll have some work to do to determine and fulfill minimum viable compliance.

Clearing away old compliance

When you really peer into the audit abyss, you'll often find out that many of the tasks and time bottlenecks are caused by too much ceremony and processes no longer needed to achieve the original goals of auditability. Target's Heather Mickman recounts her experience with just such an audit abyss clean-up in *The DevOps Handbook* (IT Revolution Press):

> As we went through the process, I wanted to better understand why the TEAP-LARB [Target's existing governance] process took so long to get through, and I used the technique of "the five whys," which eventually led to the question of why TEAP-LARB existed in the first place. The surprising thing was that no one knew, outside of a vague notion that we needed some sort of governance process. Many knew that there had been some sort of disaster that could never happen again years ago, but no one could remember exactly what that disaster was, either.

Finding your path to minimal viable compliance means you'll actually need to talk with auditors and understand the compliance needs. You'll likely need to negotiate if various controls are needed or not, more or less proving that they're not.

Boston Scientific's CeeCee O'Connor described one method for reviewing such controls. When working with auditors on an application that helped people manage a chronic condition, O'Connor's group first mapped out what they called "the path to production."

This was a value-stream-like visual that showed all of the steps and processes needed to get the application into production, including, of course, compliance steps. Representing each of these as sticky notes on a wall allowed the team to quickly work with auditors to go through each step—each sticky note—and ask whether it was needed. Answering such a question requires some criteria, so applying Lean, the team asked the question, "Does this process add value for the customer?"

This mapping and systematic approach allowed the team and auditors to negotiate the actual set controls needed to get to production. At Boston Scientific, the compliance standards had built up over 15

years, growing thick, and this process helped thin them out, speeding up the software delivery cycle.

You're already helping compliance

The opportunity to work with auditors will also let you demonstrate how many of your practices are already improving compliance. For example, pair programming means that all code is continuously being reviewed by a second person and detailed test suite reports show that code is being tested. When you understand what your auditors need, there are likely other processes that you're following that contribute to compliance.

Discussing his work at Boston Scientific, Pivotal's Chuck D'Antonio describes a happy coincidence between lead design and compliance. When it comes to pacemakers and other medical devices, you're supposed to build only exactly the software needed, removing any extraneous software that might bring bugs. This requirement matches almost exactly with one of the core ideas of minimum viable products and Lean: deliver only the code needed. Finding these happy coincidences, of course, requires that you work closely with auditors, from day one. It will be worth a day or two of meetings and tours to show your auditors how you do software and ask them if anything lines up already.

Transform Compliance

Although you might be able to avoid compliance or eliminate some controls, regulations are more likely unavoidable. Speeding up the compliance bottleneck, then, requires changing how compliance is done. Thankfully, using a build pipeline and cloud platforms provides a deep set of tools to speed up compliance. Even better, you'll find cloud-native tools and processes improve the actual quality and accuracy of compliance.

Compliance as code

Many of the controls that auditors need can be satisfied by adding minor steps into your development process. For example, as Boston Scientific found, one of its auditors' controls specified that a requirement had to be tracked through the development process. Instead of having to verify this *after* the team was code complete, it made sure to embed the story ID into each Git commit, automated build, and

deploy. Along these lines, the OpenControl project has put several years of effort into automating even the most complicated government compliance regimes. Chef's InSpec project is also being used to automate compliance.

Proactively putting in these kinds of tracers is a common pattern for organizations that are looking to automate compliance. There's often a small amount of scripting required to extract these tracers and present them in a human readable format, but that work is trivial in comparison to the traditional audit process.

Put compliance in the platform

Another common tactic is to put as much control enforcement into your cloud platform as possible. In a traditional approach, each application comes with its own set of infrastructure and related configuration: not only the "servers" needed, but also systems and policy for networking, data access, security settings, and so forth.

This makes your entire stack of infrastructure and software a single, unique unit that must be audited each release. This creates a huge amount of compliance work that needs to be done even for a single line of code: everything must be checked, from dirt to screen. As Raytheon's Keith Rodwell lays out, working with auditors, you can often show them that by using the same, centralized platform for all applications you can inherit compliance from the platform. This allows you to avoid the time taken to reaudit each layer in your stack.

The US federal government's cloud.gov platform provides a good example of baking controls into the platform. 18F, the group that built and supports cloud.gov described how its platform, based on Cloud Foundry, takes care of 269 controls for product teams:

> Out of the 325 security controls required for moderate-impact systems, cloud.gov handles 269 controls, and 41 controls are a shared responsibility (where cloud.gov provides part of the requirement, and your applications provide the rest). You only need to provide full implementations for the remaining 15 controls, such as ensuring you make data backups and using reliable DNS (Domain Name System) name servers for your websites.

Organizations that bake controls into their platforms find that they can reduce the time to pass audits from months (if not years!) to just weeks or even days. The US Air Force has had similar success with

this approach, bringing security certification down from 18 months to 30 days, and sometimes even just 10 days.

Compliance as a service

Finally, as you get deeper into dealing with compliance, you might even find that you work more closely with auditors. It's highly unlikely that they'll become part of your product team; though that could happen in some especially compliance-driven government and military work where being compliant is a huge part of the business value. However, organizations often find that auditors are involved closely throughout their software life cycle. Part of this is giving auditors the tools to proactively check on controls first hand.

The Home Depot's Tony McCulley suggests giving auditors access to your continuous delivery process and deployment environment. This means auditors can verify compliance questions on their own instead of asking product teams to do that work. Effectively, you're letting auditors peer into and even help out with controls in your software.

Of course, self-service compliance works only if you have a well-structured, standardized platform supporting your build pipeline with good UIs that nontechnical staff can access. As Discover's Dean Parke explains, describing how they handle separation of duties, "Make it easy and make it where it's just a click of a button for people to advance the pipeline to move it forward."

Improving Compliance

> *Compliance is easier with automation because it is repeatable and I can let the compliance people do self-service. They can stop scheduling meetings to look at information they now have access to.*
>
> —Jon Osborn, Great American Insurance Company.

The net result of all of these efforts to speed up compliance often improves the quality of compliance itself:

- Understanding and working with auditors gives the product team the chance to write software that more genuinely matches compliance needs.
- The traceability of requirements, authorization, and automated test reports gives auditors much more of the raw materials needed to verify compliance.

- Automating compliance reporting and baking controls into the platform create much more accurate audits and can give so called "controls" actual programmatic control to enforce regulations.

As with any discussion that includes the word "automation," some people take all of this to mean that auditors are no longer needed. That is, we can get rid of their jobs. This sentiment then gets stacked up into the eternal "they" antipattern: "well, *they* won't change, so we can't improve anything around here.

But, *also* as with any discussion that includes the word "automation," things are not so clear. What all of these compliance optimizations point to is how much waste and extra work there is in the current approach to compliance.

This often means auditors working overtime, on the weekend, and over holidays. If you can improve the tools auditors use, you don't need to get rid of them. Instead, as we can do with previously overworked developers, you end up getting more value out of each auditor and, at the same time, they can go home on time. As with developers, happy auditors mean a happier business.

Building Your Pipeline and Choosing Your Platform

"The technology is the least important thing," you'll often hear. I might have even typed that here! There's truth to that: a tool without the skills to use it, let alone the human to wield it, isn't that useful. However, there are two technologies that are vital for improving how your organization does software: pipelines and platforms. Without them, you would have a much more difficult, if impossible time of transforming.

The Build Pipeline

Your build pipeline is one of your most important software delivery tools. It's the connection between a developer committing code and adding new functionality to the application in production. Between these two points, the code is built into software, verified with multiple types of tests, checked against audit and security controls, prepared for deployment, and, in some cases, even automatically

deployed to production. You'll likely hear the pipeline referred to as "CI/CD"; that is, continuous integration and continuous delivery.

Getting a build pipeline in place is key. If you don't have one already —let alone just continuous integration—drop everything and put a pipeline in place. Gary Gruver summarizes how critical a pipeline is in his short, excellent book *Start and Scaling Devops in the Enterprise* (BookBaby):

> [Deployment pipelines] address the biggest opportunity for improvement that does exist in more large traditional organizations which is coordinating the work across teams. In these cases, the working code in the deployment pipeline is the forcing function used to coordinate work and ensure alignment across the organization. If the code from different teams won't work together or it won't work in production, the organization is forced to fix those issues immediately before too much code is written that will not work together in a production environment. Addressing these issues early and often in a deployment pipeline is one of the most important things large traditional organizations can and should be doing to improve the effectiveness of their development and deployment processes.

Now, let's briefly look at each component of a build pipeline.

Continuous integration

Many organizations are not enjoying the benefits of *continuous integration* (CI), nevermind *continuous delivery* (CD). CI has been around since the early 1990s; it took hold especially with Extreme Programming. The idea is that at least once per day, if not for each code check-in, you build the entire system and run tests. That is, you *integrate* all code from all developers together.

By integrating smaller chunks of code more frequently, teams reduce the amount of time it takes to integrate new changes into their product. This also reduces the risk of delaying releases because the build is broken and requires a fix. Thus, having the ability to build and test multiple times each day is key to keeping the release schedule running smoothly. This is the "continuous integration" part of CI/CD.

There are numerous tools and practices to automate and make this feasible. The DevOps reports have found a strong correlation between CI and high-performing organizations each year.[16]

Continuous deployment

After your release is built, tested, and properly logged for compliance and security checks, you need to somehow get it to production. One of the key insights of DevOps is that the tested, certified build is only half of the work. Releasing to production is the other half, and it's just as much the team's responsibility as writing and building the code. You're not done until your code is running in production.

Here, CD works hand-in-hand with your cloud platform. As illustrated in Figure 2-5, the pipeline takes the packaged build through a series of tests on staging and, ideally, production environments. To do this, the pipeline relies on another DevOps principal: treating infrastructure as code. All of the configuration and processes needed to deploy the release to production are managed as if they were application code: checked into version control, tested, and tracked as they should be, as part of the release.

Your pipeline takes this production configuration and bundles it with the release. It is then ready for your platform's automated deployment capabilities to do the final release. This entire process should take minutes. Great American Insurance Company, as reported by Jon Osborn, can do pipeline releases in about 10 minutes. Sometimes, the cycle is longer due to complexity or compliance concerns, but it should take less than a day; otherwise, your small-batch process will dramatically slow down.

16 These reports have found that "working off trunk"—that is, *not* branching code for more than a day—is indicative of high performance, as well.

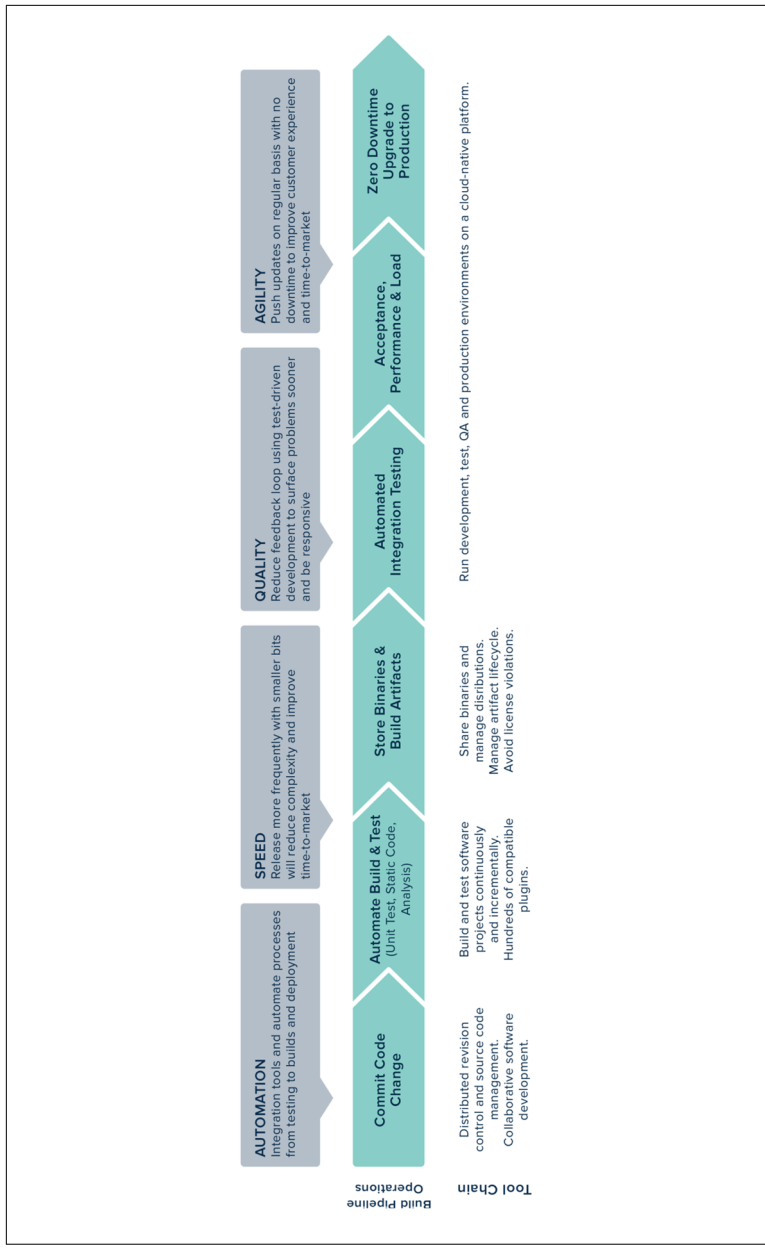

Figure 2-5. An illustration showing the elements of your build pipeline. (From "Speed Thrills," Ben Kamysz and Jared Ruckle, Aug 2017.)

With a fully automated build pipeline, the only human fingers involved are in that first commit. The pipeline does *everything* else. This amount of automation isn't practical for some organization's compliance policies. In such cases, that button gets moved down the pipeline. The release pauses right before deploying to production when a human gets the chance to approve the deploy.[17]

Properly done, a pipeline will automate the vast majority of the work required to get a build in production, including compliance and security checks. When you can be ready to deploy to production in minutes, you'll rely on a fully automated cloud platform like Pivotal Cloud Foundry at the end of the pipeline. The two together will have you speeding up your software development life cycle and quickly improving how your organization functions. As Pivotal's Matthew Parker puts it "[a] team that can't ship, can't learn. And the longer you're not learning, the greater the risk that you're wasting time and money building the wrong thing."

Putting a pipeline in place

Many people report a simple process to start putting a pipeline in place: figure out all the tasks needed to put just one new line of code into production. This could be something like changing the font or color used for a print button. Look through all of the activities needed and the time it takes and begin ruthlessly automating each activity as much as possible, or eliminating them entirely if they seem to "do nothing."

Pivotal's Jamie O'Meara suggests an even simpler and more telling test: what would it take to simply release the current version of your software, with no configuration or code changes? "If you tell me it still takes you three to six months," he says, "then that's pure process that's just sitting in front of the delivery mechanism." Many of those activities will just be governance ceremony and things that actually don't need to happen or can be drastically cut back or automated.

"You're going to find out that if it still takes you that time to deliver," he goes on, "there's a bunch of nonvalue added [work] that's block-

17 Some people distinguish between "continuous deployment" and "continuous delivery." *Deployment* means the build is automatically deployed each time, hopefully multiple times a day. *Delivery* means a human could choose to deploy it but you're not automatically deploying it. There's some more nuance to the distinction if you're into precision.

ing you, that's in your way." When you're building your pipeline, start by removing that "waste" from your pipeline and then keep up the work of removing waste from your pipeline with automation.

You're Gonna Need a Platform

The amount of automation, standardization, and controls required to deploy on a weekly, let alone daily, basis requires a degree of automation that's completely foreign, and even fantastical, to most IT organizations. You'll need a cloud platform to meet those needs. To prove this out, a common first parlor trick is to chart out all of the activities, approvals, and time that it takes to deploy just one line of code to production. Even better, assume that there's no new code and you're just deploying the current version from scratch. This is the simplest value-stream map a software-driven organization could make.

In this exercise, you can't cheat...er, rather, optimize and go against existing policy, bringing in your own infrastructure and SCP'ing a PHP file to a server with some purloined credentials. The goal is to see how long it takes to deploy one line of code following all of the official procedures for starting a new project, getting the necessary infrastructure, doing the proper documentation and policy reviews, and so forth all the way up to running the line of code in production.

The results, as you can imagine, are often shocking. It usually takes at least a month, or even years for some government organizations. Some organizations find that just gathering together all the activities and wait times to put a value-stream map together is interminable.

The ability to deploy code on a small-batch loop requires a platform that takes care of most all of the infrastructure needs—across servers, storage, networking, middleware, and security—removing the time drag associated with provisioning and caring for infrastructure. Gaining the trust of auditors, security experts, and other third-party gatekeepers requires building up trust in a repeatable, standardized stack of software.

It gets worse! These are just the day *one* problems of getting the first version of your software out the door into production. After that come the day 2+*n* problems of managing that application in production, updating all your applications weekly, and then updating the platform itself.

Cloud platform reference architecture

When you put together a reference architecture of all of the capabilities needed to support all of the days of your cloud-native life, you quickly realize how much the platform does. Figure 2-6 shows one reference architecture based on conversations my company, Pivotal, has had with organizations executing their cloud-native transformations.

Any good cloud platform has deep capabilities in these five domains (taken directly from the Pivotal whitepaper referenced in Figure 2-6).

Infrastructure
: Infrastructure is provided as a service, commonly thought of as IaaS, whether public or private. The platform requests, manipulates, and manages the health of this infrastructure via APIs and programmatic automation on behalf of the developers and their applications. In addition, the platform should provide a consistent way to address these APIs across providers. This ensures that the platform and its applications can be run on and even moved across any provider.

Operations
: Metrics and data are the lifeblood of a successful operations team. They provide the insights required to assess the health of the platform and applications running on it. When issues arise, operational systems help troubleshoot the problem. Log files, metrics, alerts, and other types of data guide the day-to-day management of the platform.

Figure 2-6. An example of what cloud platform reference architecture can look like. (From "The Upside-Down Economics of Building Your Own Platform," Jared Ruckle, Bryan Friedman, and Matt Walburn, 2018 ed.)

Deployment

Deployment tooling enables continuous building, testing, integration, and packaging of both application and platform source

code into approved, versioned releases. It provides a consistent and durable means to store build artifacts from these processes. Lastly, it coordinates releasing new versions of applications and services into production in a way that is automated, nondisruptive, and doesn't create downtime for consumers during the process.

Runtime, middleware, and data
: The components of the stack interact with custom code directly. This includes application runtimes and development frameworks, in addition to commercial and open source versions of databases, HTTP proxies, caching, and messaging solutions. Both closed and open source stacks must have highly standardized and automated components. Developers must access these features via self-service, eschewing cumbersome, manual ticketing procedures. These services must also consume API-driven infrastructure, operations tooling for ongoing health assessment, and CD tooling.

Security
: The notions of enterprise security compliance and rapid velocity have historically been at odds, but that no longer needs to be the case. The cloud-native era requires their coexistence. Platform security components ensure frictionless access to systems, according to the user's role in the company. Regulators might require certain security provisions to support specific compliance standards.

Building your own cloud platform is probably a bad idea

There are numerous—maybe even too many!—options out there for each component in the platform reference architecture. Selecting the tools, understanding them, integrating them with the platform, and then managing the full life cycle of each pillar in the reference architecture ends up requiring a team of people. And this isn't just a one-time build. The platform is a product itself requiring resolution of ongoing issues, road maps for adding new capabilities, and just basic maintenance of the code. What you have in front of you is a whole new product: your cloud platform, made up of many components, each requiring a dedicated team.

Building your own platform is, of course, technically feasible and an option. Many organizations start off building their own platform,

sometimes because several years ago when they started, there were no other options. Other times, it's a result of the fallacy of free software (if we can download open source software, it's free!), misjudging the total effort required, or giving in to the inescapable urge young developers have to build frameworks and platforms (every developer I know, including myself, has submitted to this siren many times).

For just $14 million, you too, can have your very own platform in two years

The decision to build or buy a platform shouldn't be driven by engineering thinking, but by business needs. Are time and money spent building and maintaining a platform the best use of those resources relative to, say, building the actual business applications?

In my experience, organizations that decide to roll their own platform realize a pesky truth quickly: it's more expensive than they thought. Their investment in platform engineers grows faster and higher than projected. First, selecting and understanding which components to use for each part of the platform takes time, and hopefully you pick the right ones the first time. Then, those components must be integrated with one another. And, of course, you'll need to keep them updated and patched—and you'll need a process and system to do that. To support all of this, you'll need multiple teams dedicated full time to developing each part of the platform.

Each subsystem demands multiple engineers and a product manager, and also staff to coordinate the entire effort—just like a real product! In working with numerous large organizations, even a minimal do-it-yourself platform team will consume something like 2 years of time and $14 million in payroll, across 60 engineers.[18] Worse, these organizations may need to wait up to two years to *start* their cloud-native transformation in earnest because they need to build the platform first, then they can get back to the original problem of building business applications.

18 In recent years, the capabilities and fame of Kubernetes have called many organizations to the do it yourself rocks. Although the core of Kubernetes is nothing to sniff at, all of the add-on layers needed to make a full platform tend to steer you back to those rocks.

Platform-as-a-product with the platform engineering team

There's a team of people who own and run your platform. At Pivotal, we call this role "platform engineers," others might call them Site Reliability Engineers (SRE), "DevOps," or any number of titles. As ever, what they're called doesn't matter. What they do and how they do it is the thing to focus on.

The platform engineering team's key principal is to treat the platform like a product, with the product teams as their customers.

Standing up a platform isn't a one-time project, a static service to be delivered with SLAs. It's the same never-ending series of small batches discussed earlier that takes in requirements from customers, prioritizes which ones to implement this week, develops and releases the software, and verifies that the customer's life was improved—trying it all over again if not. That continuous improvement is the *product* part of platform-as-a-product.

Platform engineers are typically more operations centric in their day-to-day work; however, they apply a programmer's mindset to solving problems: can this task be coded, automated so that people no longer need to deal with it directly? In SRE, that kind of manual work is called "toil," and cutting out toil is one of the top goals.

The platform engineering team is responsible for standing up the platform initially, upgrading the platform as new versions come out, and building in shared services for the product teams. For example, product teams might want to use Kafka to handle data. Instead of each team configuring and managing their own instances, the platform engineering team typically adds this into the platform. The platform engineering team might also add audit automation and self-service; for example, getting audit windows down from 10 months to less than a week, like the US Air Force. Or they might accelerate a bank's global growth by providing a shared banking-as-a-service platform like Scotiabank.

With the right amount of toil reduction and a continual focus on it, the platform engineering team automates an unfathomable amount of traditional operations work. "This made sure that our software engineers could just push from the CI tool without worrying about change tickets, security scanning, or approvals because it all happened through automation," says Matt Curry describing the degree of self-service given to product teams at Allstate.

This reduction in toil time not only means a much, much smaller operations team size but also means those platform engineers can focus most of their time building their product instead of frittering their time away on help-desk tickets.

Case study: selecting a platform at Rabobank

Rabobank's platform journey is a great example of well-reasoned platform strategy. As Rabobank's Vincent Oostindië explained, the company needed to replace its highly successful but now aged platform. Its existing Java-based platform had run the organization's online banking application for many years but could no longer keep up with new technologies, scale, and the "you build it, you own it" DevOps principles the bank needed.

"We also came to the conclusion that as a bank, we shouldn't be building a platform," Vincent explained. That work would require a lot of resources without directly adding value for the end user: "It would mean people working on that every day, and, well that's not bringing any business value."

As with most organizations, at Rabobank, choosing a new platform, traditionally, is driven by a committee wielding spreadsheets that list endless features and requirements. Each row lists a capability, feature, or type of "requirement" that the committee assumes each operator and developer will need. At this point, most enterprises would pick a platform using advanced column sorting strategies, vendor haruspex, and disciplined enterprise architecture futurology.

Instead, treating the developers as customers, Rabobank experimented with several different platforms by having developers actually use the platforms for small projects. Following the product approach, they then observed which platforms served the developers best. This working proof-of-concept (PoC) was driven by user validation, proving out which platform worked best. More important, it proved that developers liked the platform. "If you guys don't like it, you'll just go away," Vincent explains, "and we have a nice platform —or, technically nice platform—but, [with] no users on it, [there's] no point."

For virtually every organization, time and money spent building its own platform from scratch is waste. When evaluating which platform to use, I'd suggest using Rabobank's working PoC model, weighting the productivity and satisfaction of developers heavily.

Own Your Role

Anywhere there is lack of speed, there is massive business vulnerability:

Speed to deliver a product or service to customers

Speed to perform maintenance on critical path equipment

Speed to bring new products and services to market

Speed to grow new businesses

Speed to evaluate and incubate new ideas

Speed to learn from failures

Speed to identify and understand customers

Speed to recognize and fix defects

Speed to recognize and replace business models that are remnants of the past

Speed to experiment and bring about new business models

Speed to learn, experiment, and leverage new technologies

Speed to solve customer problems and prevent reoccurrence

Speed to communicate with customers and restore outages

Speed of our website and mobile app

Speed of our back-office systems

Speed of answering a customer's call

Speed to engage and collaborate within and across teams

Speed to effectively hire and onboard

Speed to deal with human or system performance problems

Speed to recognize and remove constructs from the past that are no longer effective

Speed to know what to do

Speed to get work done

—John Mitchell, Director of Digital Strategy and Delivery, Duke Energy

When enterprises need to change urgently, in most cases, the problem is with the organization and the system in place. Individuals, like technology, are highly adaptable and can change. They're both silly putty that wiggle into the cracks as needed. It's the organization that's obstinate and calcified.

How the organization works—its architecture—is totally the responsibility of the leadership team. That team owns it just like a product team owns its software. Leadership's job is to make sure the organization is healthy, thriving, and capable.

DevOps' great contribution to IT is treating culture as programmable. How your people work is as agile and programmable as the software. Executives, management, and enterprise architects—leadership—are product managers, programmers, and designers. The organization is leadership's product, and they should also apply the small-batch process to its creation and growth. They pay attention to their customers—the product teams and the platform engineers—and do everything possible to get the best outcomes, to make the product—the organization—as productive and well designed as possible.

I've tried to collect together what's worked for numerous organizations going through—again, even at the end, brace yourself and pardon me—digital transformation. Of course, as in all of life, the generalized version of Orwell's 6th rule applies: break any of these rules rather than doing anything barbarous.

As you discover new and better ways of doing software, I'd ask you to share those learnings as widely as possible, especially outside of your organization. There's very little written on the topic of how regular, large organizations manage the transformation to becoming software-driven enterprises.

Know that if your organization is dysfunctional, always late, and over budget, it's your fault. Your staff might be grumpy, seem underskilled, and your existing infrastructure and applications might be pulling you down like a black hole. All of that is your product: you own it.

As I recall, the conclusion of a book is supposed to be inspirational instead of a downer. So, here you go. You have the power to fix it. Hurry up and get to work.

About the Author

Michael Coté works on the advocate team at Pivotal. He focuses on how large organizations are getting better at building and delivering software to help their business run better and grow. He's been an industry analyst at RedMonk and 451 Research, worked in corporate strategy and mergers and acquisitions at Dell in software and cloud, and was a programmer for a decade before all that. He does several technology podcasts (such as Software Defined Talk), writes frequently on how large organizations struggle and succeed with agile development and DevOps, blogs at *cote.coffee*, and is @cote on Twitter. Texas Forever!