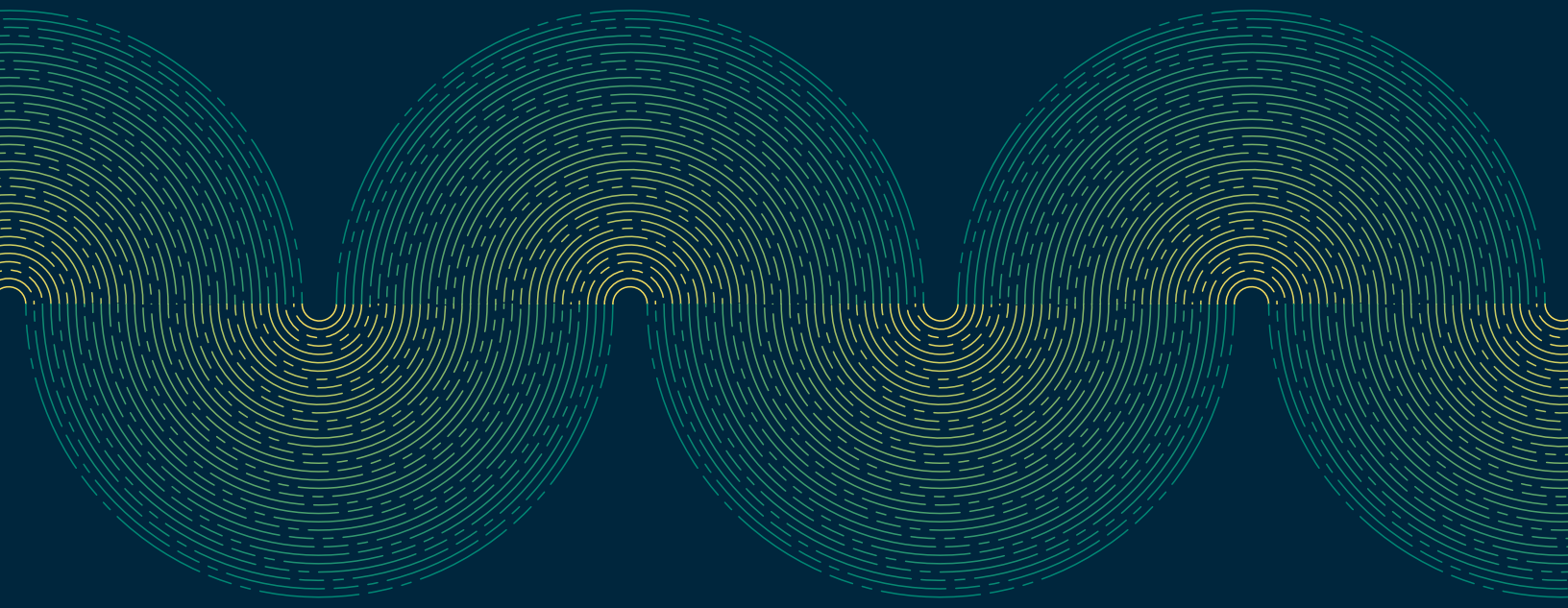
Crafting Your Cloud-Native Strategy

by Michael Coté



Pivotal

EBOOK

Table of Contents

IT's Role in the Era of Transient Advantage

I spend most of my time nowadays talking with people at large organizations in the thick of improving how they do software. Their "journeys" go by many names: "digital transformation," putting a "cloud strategy" in place, sometimes "DevOps," or as my company Pivotal puts it, "cloud-native."

Each of their efforts is trying to accomplish the same thing: improve how their organization manages and uses IT. On the surface, this often involves moving to cloud-based infrastructure, adding mobile interfaces to existing applications, and putting Internet of Things (IoT) capabilities in place. Below the surface, their changes are less about technologies and more about how IT functions.

I've found that most organizations prefer to stay at a surface level, hoping to simply install a new round of tools rather than change how they approach IT. And while the "unicorns" of the IT world have long sung the praises of the new methods they've discovered, just as with celebrity diet fads, the advice of unicorns is often a poor fit for the rest of us horses and donkeys lacking the ability to shoot magical rainbows of transformation from our hornless heads.

In this ebook, I hope to give the hornless just enough advice to first realize that becoming a cloud-native organization is possible, and second, to start planning their cloud-native strategy. It's an update, a second edition, to my similar collection from 2015, The Cloud-Native Journey. Since then, there's been a bounty of case studies from organizations transforming to a cloud-native approach and I've had countless conversations with organizations eager to start the transformation themselves. The interest in cloud-native is high and will only grow. This booklet reviews the organizational struggles and changes that come with becoming cloud-native and, hopefully, encourages you to get started, providing just enough of a toolkit to bootstrap transformation.[1]

The Goals of Cloud-Native and What the Term Means

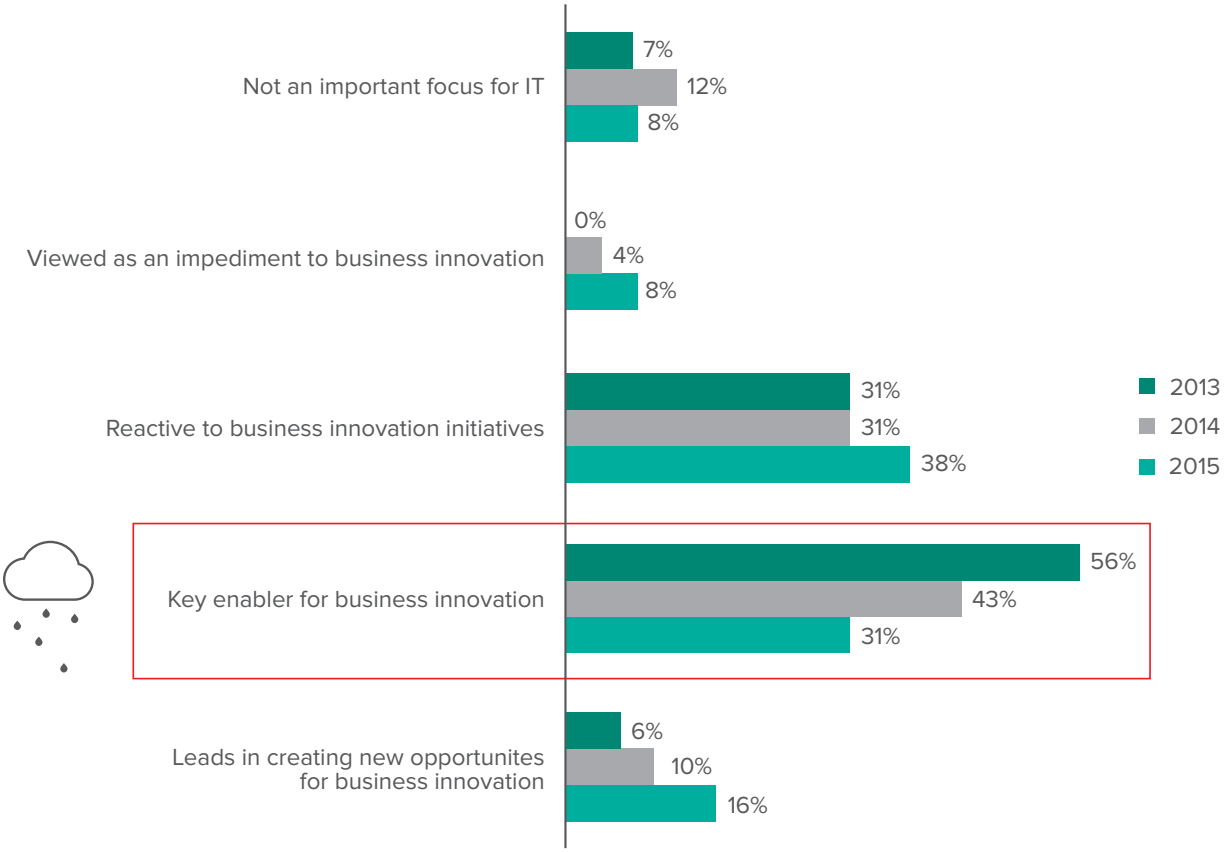
While the exact path to becoming cloud-native is unique to each organization, the goal is the same. Namely, organizations delivering valuable features to users in the form of custom-written software that's continually improved through rapid iteration. IT's challenge is putting cloud-native processes and technologies in place that enable those frequent deployments. and thus, freeing resources to focus on a user-centric approach to software design. As we'll discuss, these tools are largely drawn from the bucket of technology known as cloud, hence the term cloud-native.

In the commercial world, the goals fueling change are usually creating new businesses and expanding existing products and services to grow revenue and profit. In the public sector, the goals are usually to improve customer service and workforce productivity, along with, above all else, do more with less (citizens seem consistently allergic to increasing government budgets). In both cases, IT's goal is to ensure a steady, reliable stream of innovation that improves how the organization performs and fulfills its goals in relation to customers, staff, citizens, or whomever is considered the software's user.

1 For deeper technical overviews on the same topic, see Cloud-Native Java and Migrating to Cloud-Native Application Architectures.

"Survival is Optional. No One has to Change."[2]

These sound like IT truisms: computers can more efficiently automate tasks, and are often faster and cheaper than equivalent human effort. Indeed, the waves of IT innovation from punch cards to mainframes to spreadsheets to client/server to enterprise resource planning (ERP) to the web and now to mobile have delivered on productivity goals. For many years, however, mainstream IT has fallen behind on being useful—to put it kindly. We all make fun of "the nerds from IT" for their sullenness, but supporting this cliché are surveys that chart out just how far behind IT has fallen. In a three-year study, the Cutter Consortium found that just 30% of respondents felt that IT helped business innovate. As the chart below shows, this has fallen from about 50% in 2013.



Source: Cutter Benchmark Review, May 2015, n="80 organizations."

2 I always shy away from "CHANGE OR DIE" rhetoric, but this Dr. W. Edwards Deming thinking is apt for the innovation gap that IT is currently facing. The more popular rephrasing of this quote is "survival is not necessary," which certainly is more compact.

The Era of Transient Advantage Demands Cloud-Native IT

In this context of less-than-useful IT innovation, organizations are now faced with dramatic changes on the playing field. Although companies could once rely on establishing competitive advantage and high barriers to entry to sustain multi-decade, profitable runs, a new era of transient advantage is forcing businesses to continually innovate and scrap like never before for survival. In the 1960s, companies that "made it" could expect to stay in the S&P 500 an average of 60 years. The estimates now are that by 2020, the average will be closer to 12 years. Put another way, as R. Ray Wang puts it, "[s]ince 2000, 52% of the names on the Fortune 500 list are gone, either as a result of mergers, acquisitions or bankruptcies."

In previous eras, putting spreadsheets, ERP systems, and the required data centers in place could give companies competitive advantages against rivals that had yet to embrace such technologies. Even a strategy as simple as having a mobile workforce that could do business on devices, such as laptops and BlackBerry devices, could give companies an edge. Thanks to the simple passage of time and innovations like SaaS, ubiquitous Internet, and smartphones in everyone's pockets, achieving differentiation with off-the-shelf IT is near impossible.

Instead, the unique, IT-driven products and services an organization provides, by definition, are the only way to achieve meaningful differentiation in today's market. Companies such as Amazon and Netflix that have built their own suite of custom software and services have been demonstrating this over the last decade. In recent years, companies like Allstate, Ford, and The Home Depot have been revamping their approaches to IT to create custom-written software that makes their offerings stand out in stark contrast to competitors.

Of course, once success is found in a new mouse trap, competitors will be quick to copy it. We now take online banking for granted, but it seemed an odd, risky, even bogus idea in the late 1990s. Establishing just one, two, or even ten points of differentiation isn't enough. Instead, companies must continually innovate and improve their offerings. Instead of being experts at deploying and running applications, organizations must become experts at developing and evolving products. An organization's most durable, strategic asset is a culture of innovation.

For whatever reasons, the way that most IT organizations operate does not seem oriented around creating a thriving culture of innovation. The past decade of cost cutting and other "optimizations" are partly to blame, but I find that organizations also have lost faith in the idea that they can produce great, useful software. They become Eeyore, happy to live in their little hut made of sticks and discarded help desk tickets. Many organizations become collective experts at explaining why change isn't possible as competitors hurl themselves at the fences of their data centers.

This attitude won't do in the era of transient advantage. The first step to avoid it, then, is to make sure you have an effective way of managing your software's life cycle in place, which means looking at your development process and how you've organized your IT department.

PART 1:
Changing Your Process and Organization

PROCESS CHANGE

> "Culture is a set of beliefs and habits held in common by many people, and which only reveals its nature when it is held in common by many people."
>
> — Ryan Avent, The Wealth of Humans

A large organization must have a process in place to function at scale. By process, I mean the collection of explicit and tacit knowledge that defines how an organization goes about its daily work, and how all those days add up to weeks and months that end up with a finished product or service. It's the company's "know-how"—both the thinking and actual methods that define what the company does. While many balk and tut-tut at the word process (thinking of the word as more of a stifling straitjacket), I'd rather embrace the word and wangle it into a helpful tool rather than confine it to the jargon bin.

Whatever you want to call it, without a strong, end-to-end process in place, individuals and teams tend to locally optimize for just their jobs, failing to work together to achieve common goals. As I've studied how organizations create software, I've noticed that their chief problem is a lack of well-maintained and widely used process, in addition to lack of healthy introspection about their process that leads to continual improvement.

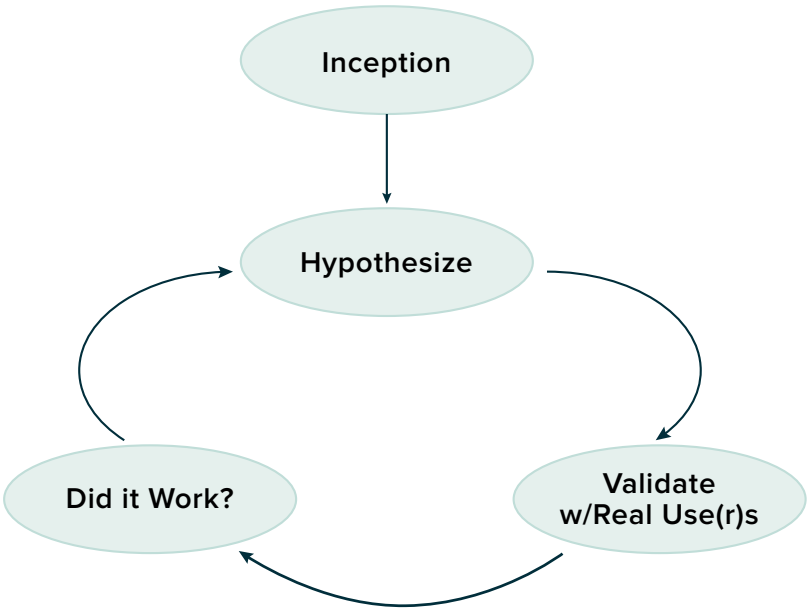
Management in most organizations brazenly assumes both that there is a process and that it's being followed. Staff often snicker and roll their eyes, but are as complicit in "process theater" as management—even more so because staff usually knows exactly what's wrong and how to fix it...if only management would listen.

Things are not always this bad, but no matter how well you think your organization is doing when it comes to process, you should ensure that you're at least continually learning and refining your process. Before that, of course, you should make sure you even have a process! And, if you're going to be a cloud-native organization, chances are you'll need to update your process to not only introduce new technologies and practices, but also to become a continually improving, learning organization.

When it comes to software, I've found that a small batch process is the most effective tool for the job.

Small Batch Thinking

Thinking in terms of small batches is one of the key mind shifts required to become a cloud-native organization. By "small batches," I mean identifying the problem to solve, formulating a theory about how to solve it, thinking of a hypothesis that would prove or disprove the theory, doing the smallest amount of application development and deployment needed 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.[3]



The Small batch loop

This whole process should take at most a week—perferably 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."

A large batch approach follows a different path: 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 once all of the requirements are implemented and QAed, code is finally deployed to production. With the large batch approach, this pile of unvalidated code creates a huge amount of risk.[4] This is the realm of multi-year projects that either underwhelm or are consistently late. As one manager at a large organization put it, "We did an analysis of hundreds of projects over a multi-year 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."

3 High-performing organizations can even achieve single-piece workflow, or 1x1 batches that focus on pushing just one feature through the entire cycle.
4 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. Indeed, as we'll see later in discussing the DevOps Reports, reducing WIP is a commonly found practice of high-performing organizations.

Become a Learning, not a Status-Checking Organization

> "When we were doing big design upfront, downstream changes had to go through a rigid change control process. We wound up being busy with our own process rather than delivering value, and either we didn't deliver or we delivered late."
>
> – Large European retail bank

Scaling up a small batch mindset in a large organization can be very challenging. At the middle and higher levels of an organization, most managers are focused on setting company goals and then monitoring progress toward achieving those goals. These are fine practices to have in place, but they too often leave learning and improvement by the wayside. This approach also tends to reinforce the practice of companies looking backwards at past performance rather than looking forward, let alone changing the corporate goals and strategies to match evolving markets and new capabilities. As a result, focusing just on monitoring progress rather than learning to evolve tends to calcify organizations. As disruption theory has painfully chronicled for over two decades, successful companies frequently failing to evolve their business are now falling prey to smaller companies.

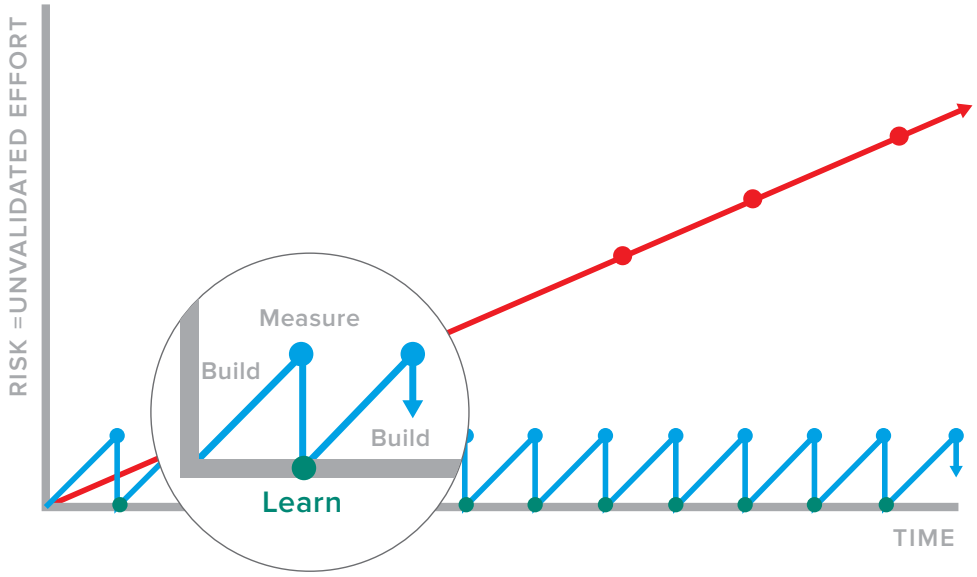
Most of the time, management spends a majority of time in what I think of as "Christmas tree meetings": reviewing slides with a list of projects that have either green, yellow, or red circles next to them. Teams or individuals report on the progress of fixed goals, not the progress of making useful, productive software. Management by Christmas tree meetings is not well suited for cloud-native organizations because it does not focus anyone on what has been learned from each release and how those learnings can be applied to improve software in the next release. Cloud-native organizations are learning organizations, not status-checking organizations.

Moving from failure management to risk management

> "In order to grow Citi, we first have to grow our own perspective, skills and capabilities… Our curiosity, our openness to learning and trying new things, our ability to adjust and adapt quickly and our willingness to fail fast and fail small are the essence of a culture that innovates and exposes new value to our clients in real time."
>
> – Stephen Bird, CEO, Citi Global Consumer Group

One of the major process changes then is for management to become much more interested in what has been learned and how it has been applied to make their software better. To do this, management—and the organization—has to shift from thinking of failure as career cataclysm to thinking about it as a synonym for learning. As you use a small batch approach, you'll find that you fail often: how you chose to code up the hypotheses in your current small batch will often fail, or your theory may be hopeless and need to be rejected outright. The point, though, is not to slowly boil in a stew of failure, but to start the process over again with the benefit of failure-driven learning, getting closer to success next time.

Thinking about the build up of risk through a project's life cycle is a good way of illustrating how failure, in small batches, is actually beneficial. The chart below shows the negative effect of large batches building up risk compared to how small batches better control risk.



The blue line shows how a small batch approach drives down risk, while at the same time steadily delivering customer value in a predictable, linear fashion. These are attributes that generally make people feel all warm and fuzzy. In contrast, the orange line shows how risk builds and builds until the very end when the software is finally released. Also worth noting is that a small batch approach delivers value very early in the process with incremental releases of feature to production. This contrasts to a large batch approach that waits until the very end to (attempt to) deliver all of the value in one big lump.

You can hopefully start to understand how this two-steps-forward, one-step-back approach can begin to transform your software delivery capabilities into a consistent, reliable, and highly risk-managed process. This approach also dovetails with one of the unspoken secrets of software: it's really hard, if not impossible, to get it right the first, second, or even twentieth time. You have to keep thinking, making, and checking until you figure it out. And then after that, your users' and customers' behavior so often changes that the software needs to quickly adapt and change as well. Many other corporate life events feed into the eternal struggle to define what your software must do: new requirements from business partners, new compliance demands from regulators, and post M&A integration tasks.

The only way to truly figure it out is to deploy your software to actual users, which may cause you to fall on your face from time to time, but will result in high-quality software in the long run. As Reid Hoffman, LinkedIn co-founder and former PayPal COO, is known for saying, "If you aren't embarrassed by the first version of your product, you shipped too late." With the right small batch mindset in place and an organization that supports a failure-driven learning approach, you can create software that's actually useful, one small batch at a time.

Case Study: "What Have You Learned?"

Faced with the need to become a learning organization, the CEO of a large retailer knew the organization had to change what happened in its own weekly Christmas tree meeting. Instead of only going through the status of projects, the CEO started asking, "What did you learn with this release?"

As with anything The Boss asks, this question prompted teams to focus on having an answer, forcing them to ask themselves what they had learned. Of course, the goal was also to then prompt a discussion around the question, "So, what are you going to do next to make our business even better?"

Shift to User-Centric Design

What, indeed, are you going to do next? If a small batch approach is the tool your organization now wields, a user-centric approach to software design is the ongoing activity you enable. There's little new about taking a user-centric approach to software. What's different is how much more efficiently and rapidly good user experience and design can be done thanks to highly networked applications and cloud-automated platforms.

When software was used exclusively behind the firewall and off networks as desktop applications, software creators had no idea how their software was being used. Well, they knew when there were bugs when the users reported them, but users never reported how well things were going (or just barely good enough) when everything was working as planned. This meant that software teams had very little input into what was actually working well in their software.

Little feedback was accompanied by slow release cycles that required a tremendous amount of staging, release planning, and other operations work not only before deploying to production, but even to give software teams the servers needed to start development! Resources were scarce and expensive, and the lack of comprehensive automation across compute, storage, networking, and overall configuration required much slow, manual work.

The result of these two forces was, in retrospect, a dark age of software design. Starting in the mid-2000s, the ubiquity of always-on users and cloud automation removed these two hurdles.

Because applications were always hooked up to the network, it was now possible to observe every single interaction between a user and the software. For example, a 2009 Microsoft study found that only about one-third of features added to the web properties achieved the team's original goals—that is, were useful and considered successful. If you can quickly know which features are effective and ineffective, you can more rapidly improve your software, even eliminating bloat and the costs associated with unused, but expensive-to-support code.

By 2007, it was well understood that cloud automation could dramatically reduce the amount of manual work needed to deploy software. The problem was evenly distributing those benefits beyond Silicon Valley and companies unfettered by the slow cycles of large enterprise. Almost 10 years later, we're finally seeing cloud efficiencies spreading widely through enterprises. For example, Pivotal customers like Comcast realized a 75% lift in velocity and time to market when putting cloud-native technologies and practices into place.

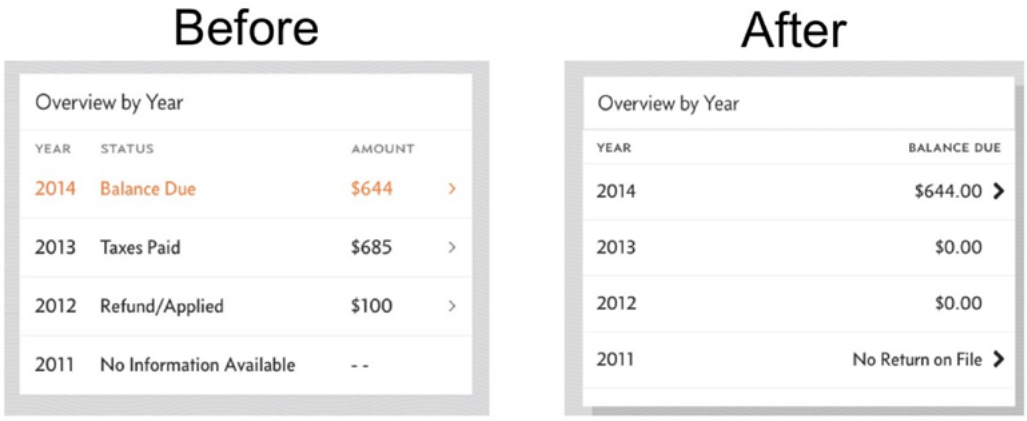
When you can both gather, and thus, analyze all user interactions as well as deploy new releases at will, you can finally put a small batch cycle in place. This allows you to create even better user interaction and product design than possible, at any scale.

Good user design practices are numerous and situational. Most revolve around talking with actual users and figuring out ways to extract what their real challenges are and then iteratively working on ways to solve them. As the infamous tire swing cartoon reminds us, the road to solving user problems is paved with well-intentioned failure.

In addition to diving into those resources, I find that a good, simple case study best illustrates all of this.

Case Study: No One Wants to Call the IRS

How much do I owe the IRS?



Source "Minimum Viable Taxes: Lessons Learned Building an MVP Inside the IRS," Dec 2015.

You wouldn't think big government, 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 user-centric approach to software. The IRS needed to improve the process taxpayers used to look up delinquent taxes. When you fail to pay your taxes on time, you're fined more daily. So, you're motivated to pay off your tax debt as quickly as possible. You would like to find out how much you owe, to the day. To do this, you traditionally would have had to call the IRS. This was not only inefficient for you, the citizen, but very costly for the IRS. And, as you can imagine, the IRS is always forced to cut budgets. Thus, the IRS wanted to create software to solve two problems: improving the experience for users and shrinking the IRS call center budget.

In building out software for this previously manual, expensive, phone-driven drudgery, the design team at first thought it should create a user interface that presented the complete history of payments and transactions with the IRS. When this approach was tested with actual users, it proved to be overwhelming. Many taxpayers ended up wanting to call the IRS. By following a small batch approach, the IRS finally winnowed down to a more streamlined interface that users validated as more useful.

In contrast, imagine how a long batch approach would have turned out. After deploying the initial version, the IRS would have had to wait something like 6-9, or even 12 months for another release window, meaning the agency would not have solved the users' original problems and would have had to keep costly call centers operating.

DevOps as the Overarching Process

If a user-centric approach describes the process of designing software in small batches, DevOps describes the process of running it. Now, the intention of DevOps is to unite all of the software life cycle into one bucket—hence the amalgamation of development and operations. However, for purposes of explanation, I hew closer to the original thinking of DevOps, back when it was called "agile infrastructure." As with all practices, once you achieve an intermediate level of understanding, everything becomes everything. But, at the beginning, it's good to have stark separation. Let's look at what DevOps is then, first, by starting with my version of how it came about.

It'd be nice if this software worked

Sometime in the mid-2000s, consumer-centric websites were battling tooth and nail for "eyeballs" and user loyalty. While the business models were not always well understood, one thing was clear: grab as many users as possible and keep ahold of them. We'll figure out how to "monetize" those people later (advertising, it turned out in many cases, or being purchased by larger companies that then monetized through advertising). To grab user attention, companies had to compete by pushing out new features constantly. Even when they eventually built up a stable base of users that refused to leave because of the network effect of social media, companies had to keep competing on features.

Now, any old salt of IT knows that the surest way to bring down a system is to frequently change it. Deploying new changes to production carries risk that the new software will have bugs previously hidden in development and exposed in production, negatively dinging uptime. Worse, changes in software may not work well with other services not under your control, whether at your own company or provided by a third party.

So if these businesses are proposing to deploy multiple releases a week, if not a day, we'll surely have zero uptime. Nothing will work. For example, in the late 2000's, Twitter famously suffered from bad uptime, giving juice to (now forgotten) alternate services like FriendFeed. Meanwhile, in other industries, stories like Knight Capital's $400m loss in 45 minutes show how enterprises can suffer from downtime.

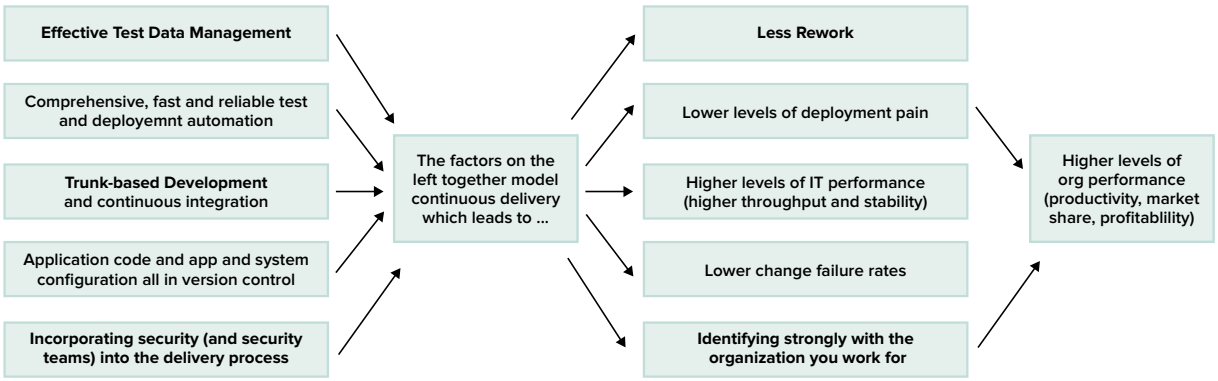
There's a paradox here: we need to deploy frequently to production to stay in business, but when we deploy to production, nothing works and we damage our business. DevOps launched as a way to solve this paradox. The grand theory was that if you could get developers to be more aware of operational concerns and operators to be more involved in the development process, the software might stay up longer.

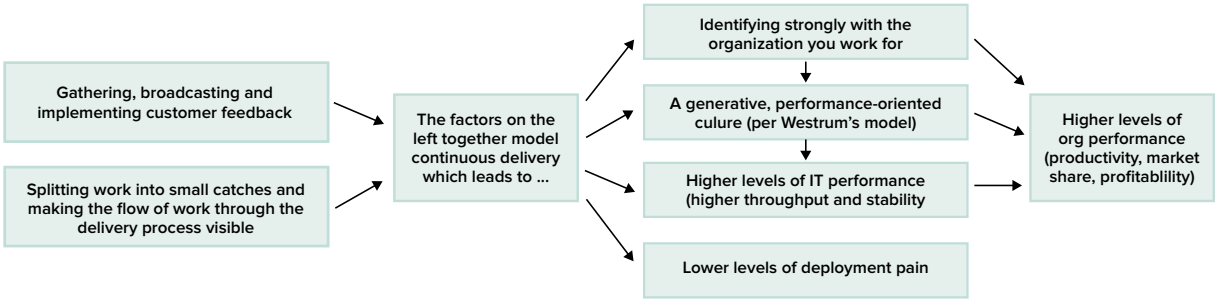
One of the first principles to emerge was putting both roles on the same team and giving everyone on that team responsibility for keeping the software up. This gave rise to the delightful cliché of developers carrying pagers. As this happened, developers started writing their software much differently to avoid being woken up past the witching-hour, while operations staff started adding in more automation and process standardization, melting snowflakes and slaying pets. The two groups started actually working together instead of battling with each other.

In addition to using sleep-driven incentives to improve code resiliency, new practices, processes, and culture were soon joined to this body of thought that became known as "DevOps." Much was borrowed from agile software development and lean software development, with a hefty sprinkling of common sense. All of it, by my reckoning, helped enable small batch thinking.

Discovering the practices of high-performing organizations

The exact processes of DevOps have been shifty for many years. There's long been an emphasis on what feels like hippie corporate think: DevOps cultists say "culture" a lot. In recent years, the DevOps Reports have done an excellent job of characterizing the practices of DevOps that lead to "high-performing organizations." The core practices are represented in the following charts:





I won't go much more into detail about what DevOps is. After years of being poorly documented, there are now numerous books and papers on the topic. In particular, for a good, brief overview I'd point you to Chapter 8 in The Practice of Cloud System Administration. Three recent books can also serve as a trilogy of DevOps manuals: The DevOps Handbook, Effective DevOps, and Start and Scaling DevOps in the Enterprise. Also, Ernest Mueller's "What is DevOps?" though published in 2010, still reads well. The point is this: high-performing organizations are finding success with DevOps, so it warrants significant investigation and use in your organization.

Put it All Together to Enable Small Batches

Process change requires walking through the following thinking:

- Because we live in an era of transient advantage, our business must continually innovate.
- Custom-written software is a major enabler of innovation, but it must follow a small batch process to put out the best, frequently innovated software and services. This is accomplished by:
  - Focusing on a user-centric approach to continually study and improve how users interact with the software.
  - DevOps to ensure that the software is operational (i.e., it works).

Let's next look at how your organization and staffing should change to align with all this.

ORGANIZATIONAL CHANGE

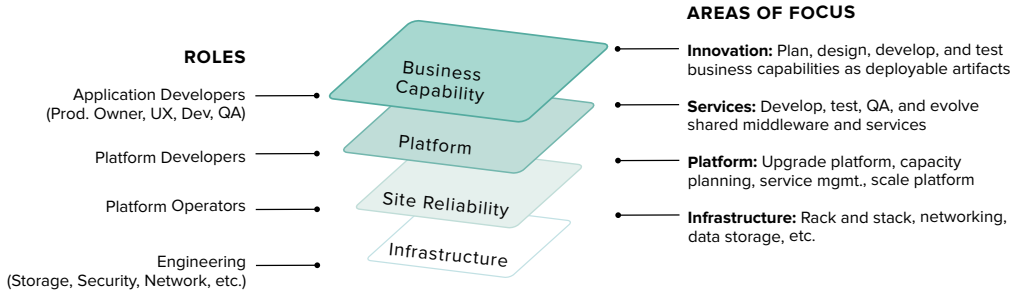
Changing your process is, perhaps, the most difficult part of becoming cloud-native. Putting the needed organizational changes into place is a close second. Teams operating in a cloud-native organization contain every role, and thus person, needed to deliver software from inception to production on the same team, with their time fully dedicated to that task. This method of organizing is often called integrated teams or balanced teams. While product teams are the organization's primary focus, there are a handful of teams that support the infrastructure—or "platform," as we tend to say—used by product teams; these are the operations teams.

Let's first look at the general structure of a cloud-native organization and then explore each of the types, teams, and what they do.

The Shape of a Cloud-Native Organization

In recent years, cloud-native organizations have been shaping into something like the structure in the following diagram:

The Emerging Cloud Native Organization Model



As depicted crudely in the size of each layer, the number of staff in each layer dramatically reduces as you go down the stack. The greatest number of people, those in the business capability layer, are working on the actual custom-written software and services. Next, the platform developers work on creating and customizing capabilities of the organization's cloud platform (e.g., a service to interact with a reservation or ERP system that is unique to the organization). Even further down the stack are cloud operations staff (the platform operators and platform engineering), working to keep the cloud

platform up and running, ensuring it's updated, and handling the hardware and networking issues. Numbers wise, the quantity of people dedicated to product teams will only be limited by the amount of products that you want to support. As discussed in the next section, between 6 and 12 people is a good rule of thumb for each product team. In the lower levels of operations (site reliability and infrastructure), however, the number of people is dramatically reduced because of the massive amount of automation and standardization done at the platform level; there's simply less manual work to be done. Examples from the Pivotal Cloud Foundry customer base show one large insurance company manages 1,500 applications with just two operators, while a large bank manages 145 apps with just two operators.

Exact numbers across each team and each layer will vary, but look for your approach to have the most people working in the business capability layer. This makes sense, if your goal is perfecting user interaction and innovating new software. You want to put most of your resources as close to the user as possible.

Product: Balanced Teams

> "The best way to do this stuff is to get a multi-disciplinary team of people in house—designer, user researcher, developer, content person—you're talking a team of about 12 people."
>
> — "Why Britain banned mobile apps; an interview with Ben Terrett, former design chief at the GDS," GovInsider, June 2016.

A small batch approach to discovering, and then perfecting, your software requires a lot of trial and error experimentation. You're operating in an incredibly chaotic, information-poor environment, where decisions need to be made constantly to keep moving ahead. In complex systems like that, speed and quick access to information are important to make decisions. You should always be exploring and looking for the best fit for your software to address the customer problem you're looking to solve.

When you're executing on a weekly, if not daily, deployment feedback loop, you don't have a lot of time to synchronize across teams. You also don't have time to continually "go up the chain" to ask permission to try something new.

Thus far, the organizational practice of balanced teams seem to be the best staffing approach to maintaining focus on that overall outcome. Because of the huge degree of automation in cloud technology, organizations are now able to free up resources from the bottom of the stack to focus almost all of their resources on the more valuable application, layer on top, the actual applications being used to run the business.

Although the exact implementation of a balanced team can vary, in general, it's one team of people per application with responsibility for and authority over the software you're creating and delivering. This team is dedicated full time to the application for which they have ownership.

An innovative, product-centric approach is what's needed in an exploratory process like software creation, where people don't know what they want and are often (at first) completely wrong about what they want. Teams in this setting should be more closely attached to the software being written rather than parachuted in as needed. The team's understanding of the problem being solved, approaches tried in the past, and the overall tribal knowledge will be invaluable in figuring out the right product to build.[5] Integrated teams are not only important for product management continuity, but also for ensuring that the product is resilient in production. It's vital to keep these teams small and they should have all the skills needed for the full life cycle of the product, including development, testing, design, and operations.

Staffing wise, this typically means teams of roughly 6–12 developers, operations folks, at least one designer, a product manager, and often a quality assurance (QA) member, giving the team all of the skills needed. In larger organizations, there will likely be shared resources that may start to look dangerously like traditional teams in silos (e.g., security testers or domain experts). Ideally, you truly want every role and person on the same team, but that's not always possible.[6]

Worked in my silo

> "Typically our developers, or business analysts, or designer, work on five projects at the same time, and if you are good you work on 10. That is tremendously inefficient. So the first task was to get them out of their traditional environment and put them into the garage, concentrating on one project. That is mandatory that the team members work co-located—all disciplines from market management to analysts, designers, and developers."
>
> — Dr. Andreas Nolte, CIO, Allianz

Balanced teams are pretty much the polar opposite of how companies traditionally organize staff: teams of people are sliced up by function, both vertically and horizontally, into the data team, the QA team, the designers, the front-end team, product management, and so forth. Large organizations have, for many reasons and many years, organized themselves into functional silos, the most logical explanation being the scarcity of IT resources and skills, real or just perceived.

Lean-think people will quickly point out that dividing teams up functionally results in local optimization, which in turn damages the end-to-end goals of your software. This desire to cluster, and then control, resource allocation is usually done to get as much value out of individuals and teams as possible. As with factory thinking, if a database administrator's (DBAs) time is worth $500 an hour, your inclination is make sure they're always working so that you're not burning off cash when they're idle. The problem with locally optimizing comes into play when the silo team does indeed work as efficiently as possible, but to the detriment of the larger project. As Goldratt explained long ago in The Goal, the best approaches to process optimization focus primarily on removing bottlenecks, not just optimizing each step in the process. No matter how efficiently each step of a phase operates, the entire process will be held back, and thus governed, by the slowest component.

5 It's worth noting that putting rotating pairing in place can help ameliorate the negative effects of churn when it comes to team members. See the 2016 study "Sustainable Software Development through Overlapping Pair Rotation" which shows how Pivotal Labs successfully uses this practice.

6 "Agile in the Large" schools of thought like Scott Ambler's Disciplined Agile Delivery (DAD) spend a lot of time contemplating and advising around shared resources in large organizations.

In a traditional, "unbalanced" team approach, the communications overhead required between many different teams can also create waste in the system. A lot of information gets passed around—usually resulting in meetings and large documents—as software is handed off between business stakeholders, product managers, designers, developers, operators, auditors, and other teams that must get involved to ship and then run the software.

Worse, the "worked on my box" mentality quickly pervades: those earlier in the chain are responsible only for their deliverables—creating requirements, writing software, testing, etc. Individuals tend to think that if something goes wrong further up the chain elsewhere, well, it's not their fault. These issues introduce a huge amount of waste into the overall, end-to-end, small batch process of creating, running, and then refining software.

All of this handing off between teams can be the cause of slowdowns, not to mention the incredible number of errors introduced as software is passed between different groups. Often, numerous customer advisory boards (CABs) and auditor chains are formed as a sort of QA check on these highly manual processes, all of which creates a web of error prone-hand offs and "meetings to sync up and align." In contrast, a fully automated pipeline provides the same checks and assurances, but removes the slowdowns of manual process reviews.

Let's next look at some examples of how you might staff teams to avoid this silo approach.

Product Roles[7]

The composition of product teams will change over time as each team gels, learning the necessary cloud-native operations skills and mastering the domain knowledge needed to make good design choices.

As detailed below, the core team is composed of developers, operations, designers, and a product owner. There are also some supporting roles that come and go as needed—testers, architects, DBAs, data scientists, and other specialists.

Developer/Engineer

These are programmers or software developers. 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 front to back-end to get good exposure to all parts of a project. By using a cloud platform, like Pivotal Cloud Foundry, developers can also package and deploy code on their own through continuous integration and continuous delivery (CI/CD) tools.

Developers are not expected to be experts in all operations concerns. Instead, they rely on the self-service and automation capabilities of cloud platforms for the most common operations needs. This means they don't need to wait for operations staff to perform configuration management tasks to deploy applications. There will, of course, be operations knowledge that developers need to learn, especially when it comes to designing highly networked, distributed applications. Initially, prescriptive platform patterns help here, as well as embedded operations staff. In addition to relying on the cloud platform to automate and enforce (now) routine operations tasks, over time, developers often gain enough operations knowledge to work without dedicated operations support.

7 Much of this section is taken directly from Pivotal Labs manuals and guides written on the topic of staffing.

The number of developers on each team is variable, but so far, following the two-pizza team rule of thumb, we typically see anywhere from one to three pairs; that, is two to six developers, and sometimes more.

Operations

Until business capabilities teams in a cloud-native environment have learned the necessary skills to operate applications on their own, they will need operations support. This support will come in the form of understanding (and co-learning!) how the cloud platform works, as well as gaining assistance troubleshooting applications in production. Early on, you should plan to have heavy operations involvement to help collaborate with developers and share knowledge, mostly around getting the best from the cloud platform in place. As with development, using rotating pairing will help quickly spread knowledge. You may need to assign operations staff to teams at the beginning, making them designated operations staff instead of dedicated, as explained in Effective DevOps.

In many organizations, the operations role never leaves the team, which is perfectly normal. Indeed, the desired end state is that application teams have all of the development and operations skills and knowledge needed to be successful.

As a side note, it's common for operations staff to freak out at this point, thinking they're being eliminated. While it's true that margin-berserked management could choose to look at operations staff as waste, it's more likely that following Jevon's Paradox, operations staff will be needed even more as the amount of applications and services multiplies.

Product Owner/Product Manager

This role defines and guides application requirements. It is also one of the roles that most varies in responsibilities across products. At its core, this role is the owner of the software under development. In that respect, product roles help prioritize, plan, and deliver software that meets requirements, or stories, as they're commonly called. Someone has to have the final word on what happens in high-functioning teams. The amount of control versus consensus-driven management is the main point of variability in this role, plus the topic areas in which the product owner has knowledge.

It's best to approach the product owner role as a breadth-first role: these individuals have to understand the business, the customer, and the technical capabilities. This broad knowledge helps them make sure they're making the right prioritization decisions.

In organizations that are transitioning to cloud-native, this role also serves as the barrier between the all-too-fragile new teams and the existing legacy teams. The product owner becomes the gatekeeper that keeps all 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. While nice-looking UIs are, well, nice to have, design in software is so much more than looks. The designer takes responsibility to deeply understand the needs and challenges that users have, and how to create solutions to overcome these challenges. You might think of designers as the empathizers in chief.

As discussed, the small batch mentality of learning and improving software afforded by cloud platforms like Pivotal Cloud Foundry gives designers the ability to design more rapidly and with more data-driven precision than ever.

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 may include completing the information architecture, user flows, wireframes, visual design, and high-fidelity mock-ups and style guides. Most important, designers have to get out of the building and not only see what actual users are doing with the software, but get to know those users and their needs intimately.

Testers (Partial/Optional)
Although the product manager and overall team are charged with testing their software, some organizations either want, or need, additional testing. Often this is exploratory testing where a third party (the tester[s]) is trying to systematically find the edge cases and other bugs the development team didn't uncover.

Some Pivotal customers have reported that they've been able to dramatically reduce their QA staffing and thus, overall IT spend. While this may not always be the case, if you find yourself with a lot of QA staff, it's worth questioning the need for separate testers. Much routine QA is now automated (and can be done by the team through automated CI/CD pipelines), but you may want exploratory, manual testing in addition to what the team is already doing to verify that the software does as promised, and functions under acceptable duress. Yet even that verification can be automated in some situations, as the Chaos Monkey and Chaos Lemur show.

Architect (Partial/Optional)
Traditionally, this role has been responsible for conducting enterprise analysis, design, planning, and implementation using a big picture approach to ensure the successful development and execution of strategy. While those goals can still exist in many large organizations, the role of an architect is evolving to be an enabler for more self-sufficient, decoupled teams. Too often, this role has become a Dr. No in most large organizations, so care must be taken to ensure that the architect supports the team, not the other way around.

Architects are typically more senior technical staff who are domain experts.[8] They may also be more technically astute, and in a consultative way, help ensure the long-term quality and flexibility of the software that the team creates. They may also share best practices and otherwise enable teams to be successful. This last point is crucial for, yet often ignored by, large organizations as we'll discuss in the section titled dealing with legacy.

Data Science (Partial/Optional)
If your application requires a large amount of data analysis, you should consider including a data scientist role on the team. This role can follow the dedicated/designated pattern as discussed previously with the operations role.

8 Highly related, if not done by the same role, the notion of a business solution architect as described by Brett Beaubouef in his book on managing large commercial off-the-shelf (COTS) projects is helpful framing for how enterprise architects can help cloud-native teams.

Data science today is where design was a few years ago. It's not considered to be a primary role on the product team, but more and more products today are introducing a level of insight not seen before now. Mobile notifications surface contextual information to buyers about flash sales nearby; users are offered deals on movie rentals tuned to their viewing behavior; GE uses fast modeling and analysis to tune wind and jet turbines; and trucking companies are using analytics to program their fleets. These features help turn "dumb," transactional products into "smart," differentiated products.

Other Roles

There are many other roles that can, and do, exist in IT organizations. These are roles such as DBAs, security operations, network operations, and storage operations. In general, as with any tool, you should use what you need when you need it. However, as with the architect role above, any given role must reorient itself to enabling the core teams rather than governing them. As the DevOps community has discussed at length for nearly 10 years, the more you divide up your staffing by function, the further you move from a small, integrated team, and achieving your goal of consistently and regularly building quality software will become harder.

Agile Operations: Managing and Running the Platform

At the lower levels of the organizational stack, roles focus on operating, supporting, and extending the cloud platform in use. For this discussion, each role is described in term of roles and responsibilities typically encountered in Pivotal Cloud Foundry installs. These can vary by organization and deployment (public versus private cloud, the need for multi-cloud support, types of infrastructure as a service [IaaS] used, etc.), but are a good reference.

Application Operator

These are typically the operations people described previously. They serve as a supporting and oversight function to the business capabilities teams, whether designated or dedicated to the actual product teams. Typical responsibilities are

- managing life cycle and release management processes for apps running in Pivotal Cloud Foundry;
- responsible for the continuous delivery process to build, deploy, and promote Pivotal Cloud Foundry applications;
- ensuring apps have automated functional tests that are used by the continuous delivery process to determine successful deployment and operation of applications;
- ensuring monitoring of applications is configured and have rules/alerts for routine and exceptional application conditions; and
- acting as second-level support for applications, triaging issues, and disseminating them to the platform operator, platform developer, or application developer, as required.

A highly related, sometimes overlapping, role is the centralized development tool team. This team creates, sources, and manages the tools used by developers all the way from commonly used libraries and version control and project management tools to maintaining custom-written frameworks. Companies like The Home Depot and Netflix maintain tools teams like this, often open sourcing projects and practices they develop.

Platform Operator

This role is the typical "sysadmin" for the cloud platform itself:[9]

- Manages IaaS infrastructure that Pivotal Cloud Foundry is deployed to, or coordinates with the team that does.
- Installs and configures Pivotal Cloud Foundry.
- Performs capacity, availability, issue, and change management processes for Pivotal Cloud Foundry.
- Scales Pivotal Cloud Foundry, forecasting, adding, and removing IaaS and physical capacity as required.
- Upgrades Pivotal Cloud Foundry.
- Ensures management and monitoring tools are integrated with Pivotal Cloud Foundry and have rules/alerts for routine and exceptional operations conditions.

Platform Engineering

This team and its roles are responsible for extending the capabilities of the cloud platform in use. What this role does by organization can vary, but common tasks of this role for organizations using Pivotal Cloud Foundry are to

- make enhancements to existing buildpack(s) and build new buildpack(s) for the platform;
- build service broker(s) to manage life cycle of external resources and make them available to Pivotal Cloud Foundry apps;
- build Pivotal Cloud Foundry tiles with associated BOSH releases and service brokers to enable managed services in Pivotal Cloud Foundry;
- manage release and promotion process for buildpacks, service brokers, and tiles across Pivotal Cloud Foundry deployment topology;
- integrate Pivotal Cloud Foundry APIs with external tool(s) when required.

Physical Infrastructure Operations

While not commonly covered in this type of discussion, someone has to maintain the hardware and data centers. In a cloud-native organization, this function is typically so highly abstracted and automated—if not outsourced to a service provider or public cloud altogether—that it does not often play a major role in cloud-native operations. However, especially at first, as your organization is transforming to this new way of operating, you will need to work with physical infrastructure operations staff, whether in-house or with your outsourcer.

Transition the Organization

How you roll out these changes to your organization and the exact timeline will vary, depending on your plans and needs. As we discuss in the Getting Started section to come, doing everything at once is probably a bad start and will result in a lot of big, up-front spending that may need to be corrected. Indeed, you can use a small batch approach to carefully, but steadily transform your organization, starting small and growing as you figure it out.

9 For another brief take, see Anthony McCulley's talk at SpringOne Platform 2016 about how The Home Depot thought through this topic.

Christopher Tretina put forward a simple maturity model for creating DevOps teams that emerged at Comcast. Over the course of what I'd estimate was 6-12 months, they first seeded operations people into the development teams, then did ongoing cross-training between the developers and operators on the teams. Once everyone on the teams (read: developers) was able to deploy to production, Comcast considered the transition complete.

PART 2:
Cloud-Native Transformation—Doing the Work

Having described the business motivation, the process mindset, and the organizational structure needed to become a cloud-native organization, let's look at common barriers, hurdles, and challenges that organizations encounter on their cloud-native journeys. Large organizations, in particular, find it hard to transform and are full of unique, and little discussed, problems. We'll look at some of these issues next.

MANAGEMENT WILL BE THE FIRST TO FAIL

Much of what we're talking about when it comes to digital transformation is, to put it in developer terms, programming the organization. In studying how change happens in large organizations, I've found that the most sustainable change begins and ends with management. This work is largely the responsibility of management. A bottoms up, #OccupyCube revolt might happen in the movies and in Fast Company cover stories, but it never seems to happen in real life. Instead, change starts with leadership.

For managers leading the change to cloud-native, I've found three main areas that require close attention:

- **Creating the game:** setting goals, context, and the strategy.
- **Creating a continuous learning organization:** changing from Christmas tree managers to learning organizations continually using a small batch approach to "program the organization."
- **Building trust:** genuinely changing how you, leadership, behave for staff and marketing to the rest of the organization, why it's good, and how it works.

These are all beginning, bootstrapping areas that will evolve over time. Let's look at each.

Creating the Game

> "Management needs to establish strategic objectives that make sense and that can be used to drive plans and track progress at the enterprise level. These should include key deliverables for the business and process changes for improving the effectiveness of the organization."
>
> — Gary Gruver, Leading the Transformation

The whole reason you're shifting to a cloud-native approach is to better align IT to the business, so a natural question is, what does the business want and need? While we'd like to think that individual staff members are clamoring to know this and ensure their daily work matches the corporate goals, that's rarely the case in large organizations. Instead, it's the job of management to figure out and then tell staff both the organization's vision and strategy, and how IT supports the execution therein.

In search of strategy

What passes for strategy in many organizations is actually what I'd call plans. That is, details about how to execute a strategy. While defining strategy is one of those tasks that you immediately get wrong once you try to pin it down, let's take a stab at it. To me, a strategy is the continual discovery and definition of (1) what an organization wants to do and what it doesn't want to do, (2) who it wants to serve or sell to, and (3) the assets available and working constraints.

More than likely, the strategy of your organization has already been defined. Your job is to make it actionable to your IT department, mapping your organization's goals to the capabilities IT can provide. In our era of transient advantage, this means providing innovation as a service, driven by the creation of custom software and services that rapidly evolve. With this approach, instead of just being departmentally inward looking, you'll need to work closely with the rest of the organization outside of the IT department to help others understand how a cloud-native approach provides new capabilities and removes previous constraints[10].

The first step is defining the vision, or mission of your organization. To some extent, an existing business has an easier time coming up with a strategy because it's already been defined: we sell insurance to individuals, we sell hardware to individuals, we sell network access to other enterprises. Public sector organizations often have much of their strategy mandated: collecting taxes, for example.

Next, you want to understand how IT is driving this strategy. That is, you need to see IT's part in the overall business process. There are tried and true (although seemingly new to the IT department) techniques like value-stream mapping: take any given business process and map out all of the activities that happen from end to end, questioning if each is needed. Most people creating such maps are shocked at how much "stupid" is going on, so it's a great technique for finding and removing bottlenecks. The trick to a value stream map is that it forces people to see everything that's actually going on and understand why. To borrow from a lean and agile notion, I think of this as making the strategy visible.

Regardless of your organization's strategy and how it maps your organization's strategy and how it maps to the business process, management's job is to get all staff members to understand that strategy and how they can take action. How this is done depends on the organization, the people, and the strategy. More than likely it means clearly and pragmatically describing the strategy (over and over again) as something more than to be the best in class or delight our customers. For example, the strategy could be increasing margin by driving repeat business from customers or optimizing the operation and servicing of turbines to increase revenue and lower costs. In both cases, as with most businesses, the goal is to make more money and be damned sure that you don't make less.

Testing your game creation

One way to test how well you're communicating strategy is by cultivating squeaky wheels. When change happens, individuals often pipe up asking, "Why are we doing this? Why is this valuable to the customer?" More than likely, they're seen as troublemakers or sand in the gears, and are shut down by the group, five-monkeys style. At best, these individuals cope with learned helplessness; at worst, they leave, kicking off a sort of *Idiocracy* effect in the remaining organization and helping seed competitor talent pools.

10 This is usually called The Business.

These complainers are actually a valuable source of data for testing how well employees understand a company's goals and strategies. You want to court these types of people to continually test how effective the organization is at setting goals and strategy. One fun practice, as mentioned by Ticketmaster's Jody Mulkey, is to interview new employees a month after starting to ask them what seems "screwy around here" before they get used to it.

Creating a Continuous Learning Organization

Now, if you remember how small batch thinking works, you should take a small batch approach to listening to the complainers: you had a theoretical idea for how to explain your strategy, you explained it, and then you collected and analyzed feedback to see if it actually worked. You're creating a continuous learning organization.

Just as we can't deliver great software without continually creating, deploying, observing, and refining that software, we can't create great organizations without taking a small batch approach to designing the company. This may seem obvious, but when you look at how most large organizations actually work, processes are rarely adaptive and more likely set in stone. When you ask, "Why do we do it this way?" the answer is usually, "Because we always have!" Every IT process going back to the abacus has pondered adaptability and change, but it tends to fall by the wayside. One of management's key jobs is to make sure that doesn't happen.

Not only do organizations have to initially change to a cloud-native mentality, but they have to keep continually learning and changing. To do this, leadership will need to create a culture of learning and experimenting, closely addressing the organizational friction that prevents it: this can be anything from speeding up how long it takes for a developer to get a server to opening up previously closed office space to facilitate better collaboration.

One tactic, as discussed earlier, is to shift from management by Christmas tree to management by learning. Instead of just asking for the status of a project, management should ask what has been learned and how those learings are being applied to improve the software in the next release. When it comes to the organization itself, management would be wise to regularly ask the same questions about process, continually reassessing and improving process.

A complementary tactic to this is to put less emphasis on metrics in the first year of your transformation. As your organization is changing and learning, things will seem to go screwy. Remember, learning is largely about failing over and over again until you get it right. Even a mild obsession with key performance indicators (KPIs) and other metrics will likely result in staff returning to making sure their project status is green instead of making sure it's improving.

Building Trust and Defibrillating Staid Corporate Culture

> "Nothing will kill your culture like hypocrisy."
>
> – Matt Curry, Allstate

Finally, management need to closely monitor how they, themselves are changing. Leaders have to prove to their staff that they're walking the talk. Staff will smell hollow transformation right away; quickly identifying this year's "Vision Baloney" is a core skill to surviving in a large organization. Individuals won't actually change how they operate if they believe management is just flipping through the latest slides from the consultants over a crackly conference line.

Small things are a big deal

Matt Curry recounted how Allstate's IT leadership team navigated these challenges in an excellent talk at the 2016 Cloud Foundry Summit. Here are just a few examples of how Allstate leadership walked the talk:

- **Branding**. Naming the transformation effort and even coming up with a logo—then putting it on swag like t-shirts and desk toys—was an important way to signal the reality and tribe. Allstate even went so far as to create a distinct group and company within a company, called CompoZed.
- **Facilities.** One of the first tasks of management will likely be to change cube farms into more open, collaborative settings. With the high velocity of collaboration required in a small batch approach, even the thinnest beige walls will prohibit innovation, let alone the ability to pair effectively. Also, because this change to the physical environment is highly visible and often at odds with status quo-minded facilities staff, getting this change through will show an early victory that demonstrates management commitment.
- **Working to change silly rules.** This usually starts with changing facilities, as mentioned, but can also include addressing unhelpful rules and regulations—for example, absurd topics like restrictions on using scooters inside the office.
- **Casual dress.** Speaking of silly rules, Matt tells the story of how executives went from wearing ties to t-shirts, having a profound impact on building trust, and thus, encouraging change. He said, "I can't tell you what having a leader stand up in front of an organization with a hoodie and t-shirt does to cultural change. It all of the sudden makes it OK for everyone within that organization to participate in change."

Transformation propaganda

The rest of the organization, too, needs to build trust that the IT department is changing, and for the better. IT has always been tasked with helping the business, but as we showed at the beginning, IT teams have been falling on their faces in recent years, despite even fancier and more "executized" slides ruffled outside of the IT department. Management will need to do a hefty amount of internal marketing to sell change to the rest of the organization. Tactics like picking a series of small, but growing, projects as outlined in the next section help with this, but good, old-fashioned corporate propaganda—boasting in newsletters, making sure you show up in all-hands slides, and otherwise driving a sense of success—are also required.

> The voice in your communications should change to be more natural, as well. Stilted, corporate speak will put people off, making them think again that this is all a bunch of "Vision Baloney." Matt Curry offers some more advice about that here: "There is a tendency to take personality out of internal blog posts by sending them through approval committees and making sure they represent the 'corporate message.' Blogs are meant to be someone's opinion—allow them to be authentic and transparent. They should be respectful, but also be allowed to voice concerns over things they dislike and really highlight the things that they like. Humor is a powerful tool. It really eases the tension. We should make an effort to introduce humor and personality into our communications, so that they come across as authentic."

However you end up fueling your change propaganda, make sure you pay close attention to it. An important part of management's job during transformation is to cultivate support from the rest of the organization, which you'll need no end of as you undertake the difficult, often counter-status quo changes needed to become a cloud-native organization.

INTEGRATE YOUR CODE REGULARLY AND MAKE THE WORK VISIBLE

Many organizations are not getting 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 a 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. 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.[11] Despite this, industry surveys reveal that CI isn't as widely practiced as you'd expect with such a good, quality-inducing process.

If you're not currently doing CI, drop everything and put it in place. CI is necessary for CD, which will get you the fully automated delivery pipelines needed to be cloud-native. These pipelines are, for many, the core enablers of the small batches process, allowing you to create better software. Gary Gruver goes into this thinking and the importance of CI in-depth in his short book, Start and Scaling DevOps in the Enterprise. He explains how putting CI and pipelines in place is key to scaling agile and DevOps up in large, multi-team organizations.

Gruver wrote an article on the topic, stating:

> For these organizations, implementing DevOps principles (the ability to release code to the customer on a more frequent basis while maintaining or improving stability and quality) is more about creating a well-designed deployment pipeline that builds up a more stable enterprise systems on a regular basis so it is much easier to release the code on a more frequent basis. This is done by creating a deployment pipeline that integrates the code across the enterprise system on a much more frequent basis with automated testing to ensure that new functionality is not breaking existing code and the code quality is kept much closer to release quality.
>
> 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.

As one last recommendation, consider running your CI system in a cloud environment. The "bursty" nature of these pipelines is a perfect match for the metered pricing of cloud versus the much higher, fixed costs of traditional infrastructure.

Finance and ROI hurdles

Running the numbers to show how you're improving software is impossibly vexing. Until you're an expert at software, you won't be able to predictably measure and estimate ROI. How could you show progress if you don't know exactly what the software-driven solution is, let alone the actual

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

problem being solved? Worse, ironically, is that one of the major benefits of improving your software capabilities should be a large business transformation that completely wrecks your models, in a good way, blowing out spreadsheets. Still, ROI is often unclear. What's the ROI for online banking, driverless cars, or simply surviving when Google, Amazon, and Facebook decide to play around in your market?

Disruption hyperbole should always be tempered with some deep breathing, a strong drink, and a good night's sleep. The point here is that finance for software is fraught with unanswered questions and itself requires some innovative thinking. I spend a lot of time searching for easy answers but have come up with little beyond that tactic for dealing with auditors. My advice is to go get friendly with your finance people by understanding what they're saying and what they need, and buying them lots of lunches and 20-year-old bottles of scotch. Once you've befriended your finance folks, you'll start getting along swimmingly and might actually learn some Excel tricks.

Andrew Greenway describes the core of how to deal with finance, using his experience in the UK government as an example:

> Business cases are really about trust. When a policy team submits a business case, it gives a senior official the chance to consider whether they trust them. That trust will be shaped by a view on the quality of their thinking, yes, but also their ability to play the game, follow due process, write well, and make up plausible numbers. Once trust is built with the purse strings, greater deviation from the process is allowed for. But all of those qualities are measured according to the team's competence to deliver something which is business case-shaped. It rewards means, not ends.

Building up trust, as ever, will make all your finance dreams come true. Coupled with the graciousness to follow finance department rules, you'll find success.

The Three Finance Questions You Meet in Drab Conference Rooms

When finance and management interrogators ask about ROI and business cases, I find that they're mostly asking three questions:

- Will this fit in the budget?
- Are we paying too much?
- Will this actually work?

Sometimes they're asking all three questions; sometimes just the first two, but it's usually some combination of these three.

Will this fit in the budget?

Of all of the ROI questions, this is the easiest to answer. If you know the budget, you just need to figure out how you'll meet or come under it. When looking at the change to cloud-native, this means you'll first establish the baseline cost of following the old way, including staff pay, tooling, and the expected cost of fixing screwups. Then model how cloud-native concepts such as "two-pizza teams" and reducing release cycles will lower your costs.

If you don't know your actual budget, or if that's not "enough" to survive the finance Hunger Games, think about estimating savings in the costs of communications between disparate teams. If your teams spend less time communicating with other teams, there's less time in meetings, getting the

meeting room projector to work for presentations, and coordinating what to do after the meetings. Communication is more effective and efficient if you're all on one, small team.

You want your product teams spending 90% or more of their time on product, but they're probably spending more like 20% to 30%.[12] Fewer siloed teams will result in fewer errors caused by hand-offs between teams. Meanwhile, DevOps' smaller batches of code and weekly release cycles will increase the resilience of your applications (faster time to recover) and the productivity of your software (as you iteratively release, observe the use of, and improve your software's usability).

Health Care Service Corporation's (HCSC) use of pair programming is one example showing how agile practices mixed with a small batch approach can reduce staffing costs by 40%. For one of its initial projects, its traditional estimate put the project at 15 weeks. When the team switched over to approaches described here, they delivered the project in six weeks. Using back-of-the-envelope reasoning, that's a 40% savings. Put another way, in the traditional approach, they were overpaying by 40%.

Now, that kind of estimation is like doing engineering calculations without accounting for friction, yet it shows the general direction to look toward when putting business cases together. You should expect to pay less in time, and thus staffing, with a cloud-native approach.

Cutting yourself to greatness

In addition to becoming more efficient, there are often classic cost-cutting measures to take. If you want to pull out the trimmers, look at staff reductions. Several large organizations I've spoken with have drastically reduced their operations and QA staff after modernizing their software development and delivery approaches. Many organizations are still doing a tremendous amount of manual testing using scripts. They're often forced to rely on testers because they haven't automated much of their release cycle and testing.

Nowadays, much of that testing should be automated and more of the testing burden falling on the product teams. They should be relying on CI and exact replications of the production environment—if not production itself, with tactics like canary deploys—to handle many of the traditional testing tasks. Additionally, by focusing on resiliency over uptime, product teams can ensure easier rollbacks when errors do make it into production. All of this means how you think about QA changes and requires you to rethink your financial assumptions about QA staffing and funding.

Management doesn't speak much in big, glitzy keynotes about cutting QA. But in dark bars, over drinks, most of them will offer up stories of cutting QA organizations significantly. There are other savings as well, several in pure technology. As the raw price for IaaS continues to decline and new technologies optimize the actual cost, you can find savings if you're not locked into your infrastructure. For example, Pivotal Web Services and Pivotal R&D have switched from Amazon Web Services (AWS) to Google Cloud to take advantage of Google's different pricing model and container packing optimizations for significant reductions in annual costs.

12 This is based on Allstate's experience.

Similarly, by switching to Pivotal Cloud Foundry to support a cloud-native approach, one manufacturing company achieved 30% in hardware savings because it no longer needed to use virtualization on beefy servers.[13]

Are you paying too much?
We all like a good deal and can agree that getting fleeced is a poor outcome. You'd like to know you're not overpaying. With a process change like DevOps, the tough question is: paying for what? There are costs associated with modernizing your software approach, like buying new tools and hiring consultants (or in the agile world, coaches) to help change your organization. There are also soft expenses in time and money, like your internal debates over which reference architecture to use, procurement negotiating, proofs of concept, and training. How to account for all of these costs is befuddling.

There are no easy answers, just models and competitor matrixes to gnaw on. The raw tools to use when evaluating costs and alternatives are standard technical tests to prove out the options and gaining an understanding of the track records of other users and customers, good and bad.

You might also ask if an outsourcer can do it cheaper than your organization. Answering this question requires more of an assessment of your organization's willingness to change its process to match the ideal cloud-native state. That is, be careful when comparing your legacy organization to an outsourcer that you believe can do it better.

The outsourcer may indeed be more expert at execution than you, and thus, more productive and cheaper. However, what if you changed how your organization worked to be more like the outsourcer that you regard so highly? Organizational change like this is hard—and the specialty of Pivotal Labs—but with a couple years of up-front investment in actually changing your organization, insourcing is likely to be an attractive option that doesn't carry the risks of outsourcing.

Will this actually work?
You've crafted up numbers for a business case, spent the wee hours of December 31 horse-trading your way to a good deal, and ensured that your people can pull it off. Then unbelievably, true to Larman's Law, people keep insisting on more justification.

Other than table-flipping your way into a new job, I've found three useful tactics for handling this situation:

- **Discover other people's success, first hand.** Talking with peers outside of your company who've had success can help win over doubting Thomases and provide the raw models and figures for sorting through business cases, if not getting permission to just try it out and see if following new methods of software development can be trusted.
- **Hide.** Creating a skunk works is a tried-and-true method to bootstrap a new process, ignoring the finance gatekeepers. You're hiding out to create an internal success that can be used to prove the new approach. If you fail, there's massive risk. If you succeed, you've demonstrated that the new way is effective and to be trusted.

13 For more ROI whiz-bang, see my recent round-up of before/after numbers.

- **Start small.** Do a series of small projects to prove out the new process. These can't be science projects; they need to be small, yet important to your organization. In doing these little projects, you're building up credibility for the new process and also learning how to do it.

All of these, to varying extents, rely on building up trust with the financial gatekeepers and management. They each require some amount of executive support to simply get off the ground, though one could do skunk works completely bottoms-up with the hope that you'll avoid punishment if you're successful.

Success is the Best ROI

Sustained success makes funding discussions much easier. Once your organization trusts the new methods of operating both a priori and because of a track record, the roadblocks around up-front business models will soften. For example, to continue one of the prior stories, after delivering an initial project in half the usual time,[14] HCSC found that others in the company worried much less about funding questions and were instead clamoring to adopt the new cloud-native approach.

Case Study: IRS

Returning to the IRS case described previously, we can see the financial dynamics at play.[15] Recall that the problem to be solved was reducing call center costs associated with citizens wanting to know their delinquent tax and penalty amount.

Before the project, the only way to do this was over the phone. A recent study by a group called Taxpayer Advocate found that of all calls made to the IRS during a recent filing season, only 37% of them were ever answered. A reported 8.8 million people were "courtesy disconnected," which means the switchboard was overloaded and the IRS just hung up on them without warning. Before you even get to the money part, you know they're failing. These call centers cost millions of dollars a year to staff and maintain. Meanwhile, the IRS's budget has been cut by over $1.2bn over the past four years. The agency has had to consistently find better, cheaper ways to provide services.

Following the approaches described here, a team of Pivotal and IRS employees conducted an experiment that took less than a week. The team got concrete evidence that its solution would not prevent phone calls, and then refined the software more and more over subsequent releases. In 12 weeks, the team was able to prove what the IRS could not in two years: online account access would not generate more phone calls.

By fixing the process and approach that was being used to create software in the agency, the IRS was able to achieve excellent business results, fitting to a desirable business case. Getting the permission to try out new methods takes some trust, but new ways can have such dramatic success and savings results that finance people should be more than happy to try.

14 This provides a good example of properly selecting initial first projects with an eye toward building credibility for internal marketing and scaling up your approach to the larger organization.
15 Details here are from Pivotal's Lauren Gilchrist.

AUDITORS: YOUR NEW BFFS

There have obviously been culture shocks. What is more interesting, though, is that the teams that tend to have the worst culture shock are not those typical teams that you might think of: audit or compliance. In fact, if you're able to successfully communicate to them what you're doing, DevOps and all of the associated practices seem like common sense. Auditors say,

> "Why weren't we doing this before?"
>
> – Manuel Edwards, E*TRADE, Jan 2016

In large organizations, there's always a laundry list of hurdles from audit, compliance, security, and other governance, risk management, and compliance (GRC) gates. These gates have fortified over years in organizations as they comply with law and attempt to better manage risk. When moving to a cloud-native approach, the traditional approaches to GRC, 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 not to say that some of those things aren't still important, but the nature of that risk is changing.

Ehringer says that many of these things 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.

First, make sure you're only doing what's truly necessary. 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. Sometimes this is due to specific compliance requirements no longer being necessary, or not having been applicable in the first place. For example, PCI contains several requirements that deal with client devices only and don't apply to servers at all. There's folklore about government agencies purposefully skipping compliance with archaic compliance policies to see if anything happened; when nothing did, they proved that the policies were no longer needed.

Target's Heather Mickman recounts her experience with just such an audit abyss clean-up in The DevOps Handbook:

> 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.

Second, you not only want to avoid paving cow paths, but you also want to rely on your new set of highly automated tools to replace manual audit processes. In a cloud-native organization, everything is tracked in a proper CD pipeline down to the "who did what" line of code and "who deployed what" configuration to production. As Liberty Mutual's Ehringer puts it, "Having everything go through a continuous delivery pipeline gives 100% visibility in everything end to end." He goes onto explain:

> So that is actually very empowering when you go and talk to a security person, or you talk to an auditor, and it gives people trust in the process that you've put in place. And I think that's one of the biggest things that needs to be built up over time when you're talking about, kind of, deep ingrained cultures around managing risk by slowing everything down. To speed things up, you need to have trust and complete transparency through that process which you get through enforcing continuous delivery pipelines.

Case Study: "It Was Way Beyond What We Needed to Even be Doing."

HCSC's cloud-native journey provides a great example of working with auditors. HCSC is the United State's fourth largest health insurer, employing more than 22,000 people, and serving nearly 15 million members. Founded in 1936, the company is better known as the Blue Cross Blue Shield provider in Illinois, Texas, Oklahoma, Montana, and New Mexico. A multi-state healthcare company like this is up to its eyeballs in regulations and compliance.

Initially, HCSC felt like getting over the audit hurdle would be impossible. Mark Ardito recounts how easy it actually was once auditors were satisfied with how much better a cloud-native approach was:

> Turns out it's really easy to track a story in [Pivotal] Tracker to a commit that got made in git. So I know the SHA that was in git, that was that Tracker story. And then I know the Jenkins job that pushed it out to Cloud Foundry. And guess what? I have this in the tools. There's logs of all these things happening. So slowly, I was able to start to prove out auditability just from Jenkins logs, git SHAs, things like that. So we started to see that it became easier and easier to prove audits instead of Word documents, Excel documents—you can type anything you want in a Word document! You can't fake a log from git and you can't fake a log in Jenkins or Cloud Foundry.

Automation makes auditors happier and removes huge, time-sucking bottlenecks.

Risk Management with Small Batches

As more examples of rethinking GRC in a cloud-native organization, let's look at some new ways of thinking about risk when you apply a small batch approach:

- **Bug swarms:** If I have a week's worth of code versus half a year's worth of code and something goes wrong in production, there's a much smaller set of code to diagnose and fix. This also speeds up your ability to deploy security patches.
- **Useless software:** The biggest risk in software development is creating software that users don't find valuable, but that's otherwise perfect. With small batches, because you deploy each iteration to users, you can easily figure out if they find the software useful. Even when you get it wrong, you've only lost a week (though, I'd argue you've won in gaining valuable learnings about what does not work). As an example, recall the IRS case.
- **Stymied innovation:** Coming up with new ideas can take a very long time if you have to wait six months to try them out and see how your users react. Instead, if you deploy a series of small

batches, you can experiment and explore each week, hopefully getting into a virtuous cycle of steadily discovering new ways to delight users.

- **Budget overruns:** A small batch mentality avoids "big-bang bets" that require a massive capital outlay at first and then a white-knuckling 12–24 months of waiting before shipping the code. If you're only focused on the next few releases, finance can adjust funding either up or down as needed. The existence of government IT projects going over budget serves as an example here (though, I assure you, private industry can be just as bad; they're just better at hiding failure).
- **Schedule elongation:** Projects that don't force shipping can often find themselves forever stuck with just a few more weeks left before shipping. There are always new features to add, more hardening to do, and then suddenly it's the holidays, and you've got a good month of a windless sea adrift in the HMS Your Project...which is just long enough to think of still more new features to add. Without an emphasis on shipping every week, you eventually slow down.

Security: Same Old Story, New Spiffy Tools

Security controls present another circle of GRC hell, but of course, they matter a great deal. Here, the primary answer seems to be getting security people more involved in the cloud-native process. While there are strong attempts at this with concepts like DevSecOps and Rugged Ops, there's still a paucity of security expertise involved in helping organizations adopt—rather than stall!—these new approaches to innovation. As Gartner's Neil MacDonald and Ian Head put it recently:

> Based on hundreds of discussions with clients, we estimate that fewer than 20% of enterprise security architects have engaged with their DevOps initiatives to actively and systematically incorporate information security into their DevOps initiatives.
>
> — Gartner, DevSecOps: How to Seamlessly Integrate Security Into DevOps; September 2016

They suggest 12 practices to follow to achieve DevSecOps. Meanwhile, Pivotal's Chief Security Officer, Justin Smith, has outlined a three Rs (Rotate, Repair, Repave) approach to security that takes advantage of the highly automated, cloud-native capabilities of Pivotal Cloud Foundry:

> Its idea is quite simple. Rotate data center credentials every few minutes or hours. Repave every server and application in the data center every few hours from a known good state. Repair vulnerable operating systems and application stacks consistently within hours of patch availability. Faster is safer. It's not a fantasy—the tools exist to make most of this a reality today. Do it, and you'll see a dramatic improvement in enterprise security posture.

With security, once again, the answer is to revisit your assumptions and use the new tools and processes to solve the same problems, but solve them with much more efficient and efficacious methods.

ARE YOU REALLY DOING AGILE?

> "If you're doing 30-minute weekly 'standups' while sitting down, you're not doing agile."
>
> — Robbie Clutton, Pivotal

Of all of the topics to understand in cloud-native, the exact software development skills needed day to day are the most straightforward. Although there are significant operational skills added to the team by taking on DevOps practices, the software development practices have been honed and studied for almost 20 years now in the form of agile software development. These practices are mature and proven. As Forrester's Jeffrey Hammond says, "I think from a tactics perspective, agile is increasingly a 'solved problem.' We know many practices that work, and that have been well proven in the field."

While agile practices are well understood; it turns out, those practices aren't widely followed.

While easier practices such as unit testing are commonly used, wider use of agile practices drops off incredibly fast. Pair programming, in particular, is being ignored by around 70% of organizations, surveys often find. This is despite studies and long experience showing that pairing increases software quality, team resiliency, and overall improvement of the software development cycle.

Rather than review each practice and how it's being neglected, I'd encourage you to benchmark yourself against these results, but more importantly, verify that your organization is actually doing them. As a start, Pivotal's Robbie Clutton offers a simplified list that focuses on the goals and results of following agile.

If you're doing agile, you should be:

- Reducing the cost of change for your product
- Getting continuous feedback about how your software is used
- Continuously improving your team, leading to improving your software
- Empowering the people on teams to do the above

If you're like most people, after comparing these four goals and individual practices to how your organization is performing and operating, you'll find there's much room for improvement. During Pivotal Labs engagements, we frequently find that organizations claim to have been doing agile forever, but upon closer inspection, we learn they follow the practices piecemeal, at best. Indeed, when you look at industry surveys, about 45% of respondents admit they're still developing using a waterfall approach.

There's a lot less agile out there than you'd think. So, it's always good to verify what you think is happening. As an old journalist principal says, "If your mother tells you she loves you, check it out."

OUTSOURCING AND CONTRACTORS

One of the most common cloud-native inhibitors I hear people complain about in all types of organizations, across the globe, is outsourcing. We all know the story. Around five or ten years ago, a new CIO came into the organization and achieved huge cost savings by outsourcing much of IT, including development and operations. It was great for several years, so great that the CIO got an even better job at a new company. In the present day, these organizations are finding themselves stuck with a huge bill and results they usually don't like. And now in the era of transient advantage, when creating custom-written software is mission critical, these organizations have little to no ability to produce high-quality software.

Government contractors often are mythologized to be the worse of all outsourcers. This story from Steven Levy is a good example of what typically goes wrong with large outsourcing deals. The US Federal government had been trying to digitize the process for green card renewals, but it wasn't going well:

> **Steven Levy:** Do you have a metric that shows the difference between that immigration program before and after your small team revamped it?
>
> **Haley Van Dyck:** Well, the metric before was that the integration system was entirely paper-based. To actually apply to the system it cost about $400 per application, it took end user fees, it took about six months, and by the end, your paper application had traveled the globe no less than six times. Literally traveled the globe as we mailed the physical papers from processing center to processing center. This transformation process existed well before we showed up [but wasn't succeeding]. There was actually a billion-dollar contract that was out at one point to start this modernization process. At the end of the five-year contract, which included [an additional] two-year requirement-gathering phase, zero code was delivered that worked. They were in the second year of another five-year long contract when we showed up. And I-90 is the first functioning release that has existed on this project in almost seven years. [After the interview, a government spokesperson clarified that the first contract was a seven-year deal for $1.2 billion, and the result was "behind schedule and slower than paper."]
>
> **Steven Levy:** What did I-90 cost when you folks did it?
>
> **Haley Van Dyck:** The salaries of five people.
>
> **Mikey Dickerson:** They worked with an existing organization, but what we added to the project was five people. Not even measurable against a $1 billion contract.

While the scale of this project may not exactly fit what you've experienced, the general shape and pattern of this story matches what many large organizations tell me is wrong with contractors, be they in government or the private sector. Outsourcers too often do exactly what the contract (from five to ten years ago) says instead of helping you innovate and keep the business growing. It's little wonder that in a recent study, more than 75% of senior executives said they want to replace their legacy outsourcers because those providers are so unwilling to change to new models.

Among many others, Citi provides one example of how to address this need to change how outsourcing and offshoring is done in large organizations. Currently 80% of Citi's developers are contractors, while 20% are employees. Citi wants to invert this model by the end of 2017, and is striving for 80 percent of developers as employees with only 20% remaining contractors. Its hope is to drive a culture of ownership, leading to better product-level thinking, greater alignment to the business's needs, and more innovation.

Further, while remote work is possible, Citi has found that having co-located workers pays off. It is reducing 26 different development locations down to just four. Going through the process of co-locating teams and working with stakeholders is driving a 57% increase in the speed of delivering its software. Industry wide, we see similar initiatives. A recent Cloud Foundry Foundation survey found that "[b]y a nearly 2:1 margin, [respondents] are choosing training over hiring or outsourcing as the preferred method for addressing a shortage of skills in their own companies."

Again, what color badge team members wear won't prevent them from becoming cloud-native, but the traditional relationships with outsourcers will probably be an inhibitor. There's no way around it. You have to start insourcing more or have such a special relationship with your outsourcer that it's virtually insourcing.

That special relationship essentially means having outsourcers follow the core principals of your cloud-native approach. For example, one of the most useful requirements you should institute is that the outsourcers work on the same code base, use the same build pipeline, and follow the same integration requirements that your insourced teams use. As Gary Gruver writes:

> Ensuring the different [outsourcing] vendors don't go off track and create code that won't work together in production is critical and a well-designed deployment pipeline is a critical tool for coordinating development across different outsourced organizations in traditional tightly coupled systems. For these types of outsourced organization, DevOps principles while different are almost more important than they are for companies that do all their development internally.

One of the key enabling principals of the cloud-native approach is reducing as much variability up and down the stack. Automate as much as possible and all use the same code base that's integrated multiple times a day, as well as follow other practices that strip out variability that cause errors and slowdowns as different processes sync-up. By the very nature of being from another company, outsourcers typically create a silo that must be continually synced-up, introducing variability. From a process perspective, eliminating that variability is top of the list if you're doing any kind of outsourcing.

Arguably, the lower levels of your cloud platform could be outsourced, since there are clearly defined contracts between the application and platform layers. However, care must be taken to avoid introducing scarcity into the system. At The Home Depot, for example, after several rounds of trying to speed up the release cycle, managers found that product teams were still not doing releases

frequently. Upon investigating why, managers found that teams were charged in internal money for each release, making teams leery of blowing out budgets when they released frequently. This sort of a la carte fee schedule occurs in outsourcing arrangements and creates negative incentives that will inhibit rapid release cycles, slowing down the innovation engine for your business. I've been in several meetings with management teams that were trying to figure out how to optimize the ticketing process for automating server and networking configuration: an absurd situation in an era of endless cloud automation driven by an outdated approach to outsourcing, enforced, no doubt, by a now anachronistic contract.

DEALING WITH LEGACY

> "Digital transformation is about rethinking those legacy business processes and doing stuff in a new way."
>
> – Marc Geall, SAP

I like to think of legacy software as any software you're afraid to change, but must change. The exact age or technology of the system is less important than that fear of change. You fear changing it because you don't have a reliable and/or trustworthy enough way to test if your changes broke the software or other services that rely on the legacy software. If you have software that you have to change, but are happy to change, you usually just call that software. Of course, if you don't have to change the software, who cares?

When people talk about dealing with legacy, there are typically four buckets of concern:

- **Transforming legacy process:** Changing your approach for software creation and delivery.
- **Revitalizing legacy code:** Fixing issues in your software—the code and the architecture—that prevent you from making changes as quickly and cost effectively as you'd like.
- **Living with legacy:** Integrating with legacy services that you cannot change, at least change as quickly as your core software evolves.
- **Avoiding legacy pitfalls:** Putting effective portfolio management in place to prevent getting saddled by legacy in the first place.

The first bucket is what much of the discussion is about in this paper—changing how an organization thinks about, organizes, and then executes on software as a whole. Let's take a look at the rest of the issues.

Revitalizing Legacy Code

One of the more popular definitions of legacy code comes from Michael Feathers' classic in the field, Working Effectively With Legacy Code. He writes, "Legacy code is simply code without tests." Most code will need to be changed regularly, and when you change code, you need to run tests—to verify not only that the code works, but that your new code didn't negatively affect existing behavior. If you have good test coverage and good CI and delivery processes in place, changing code is not that big of a deal and you probably won't think of your code as legacy. Without adequate, automated testing, however, things are going to go poorly.

Thus, one of the first steps with legacy code is to come up with a testing strategy. The challenge, as Feathers points out, is going to be testing your code without having to change your code to make testing possible. Or, as Feathers summarizes:

> **The Legacy Code Dilemma**
>
> When we change code, we should have tests in place. To put tests in place, we often have to change code.

Feathers' book is 456 pages of strategies for dealing with this paradox that I won't summarize here. What I want to emphasize is that until you have sufficient test coverage, you're going to be hampered. In other words, this is one of those pesky prerequisites for being a successful cloud-native enterprise.

If your code lacks good test coverage, and especially if you don't have build pipelines in place, you likely won't be able to fix all of the problems at once. This is especially true in a large organization. Gary Gruver, in Start and Scaling DevOps in the Enterprise, describes a process to incrementally tackle these problems while still moving toward the goal of thorough testing and build automation. Once you've introduced proper CI/CD, you will have a policy in place that stops failing builds from progressing down the pipeline: product teams that are responsible for making sure the builds don't break, the tests pass, and their code integrates with the rest of the system. When bringing in new code, you can't test everything at once, so instead choose some key build acceptance tests (BATs) as a starting point for increasing test coverage. As you find failures in the build and integration, write more BATs around those failures. These failures may be in the code, the infrastructure, or elsewhere. The point is, as you discover problems, stop everything, write a test, and fix the issue.

Coupled with the code-level changes Feathers describes, this long, trying process of building out your build pipeline tests will allow you to discover what needs to be tested and strengthen the confidence in your test coverage to start making changes more fearlessly. This type of work is tedious and can be disheartening. However, if you want to improve how you do software, proper test coverage of your code and automation in your pipeline are table stakes.

Living with Legacy

In many cases, you have no control over legacy software and services. You're forced to use and rely on them. Think of external systems like airline and hotel booking platforms, or at the infrastructure layer, as networking and content delivery network (CDN) configuration. When you can't directly or quickly modernize these legacy systems, you need to put in a scheme to quarantine these systems and, longer term, a scheme to modernize or replace them.

Application replatforming: forklift with caution

People often dream of lift-and-shift schemes wherein software is simply boxed up and moved to newer, better environments. This may be the case with simple, well-written, and low-priority applications, but in many cases, simply lifting and shifting is a chimera of improvement. As Forrester's John Rymer points out, among the many legacy coping options, the life-and-shift approach looks the easiest but has the worst long-term payoff. This is because simply changing how you manage the life cycle

of the application without changing the application itself can limit the benefits of a cloud-native approach, namely, the ability to quickly add new features while maintaining a high level of availability in production. Evaluate these so called forklift fixes carefully. They could be exceedingly easy, or deceptively disastrous.

Pivotal recommends that applications to be forklifted have minimum code changes, likely none at all, and instead recommends telling teams to do just minor configuration and packing changes to get the applications running on Pivotal Cloud Foundry.

Application modernization
Some applications may actually be good candidates for major architectural and code changes that modernize them to run on cloud platforms like Pivotal Cloud Foundry. These should be applications that, for business reasons, you want to run well on your cloud platform, and thus, can invest proper resources in for doing real development work. More than likely, these will be more recent applications that have already been written in a highly componentized fashion. You should also have a relatively strong understanding of the architecture and project as you'll be rearranging the guts of the application and adding in new capabilities.

Application rewrite
Finally, it may be time to rewrite the application. If a legacy application has high business value, yet is impossibly risky to change, it's worth spending the effort to rewrite it. This may take time and even carry risk, but it might be less risky than doing nothing. One company I spoke with has been doing just this. Part of its portfolio, consisting of about five core applications and services, had grown from in-house development and acquisitions. The portfolio was more than 10 years old and showing signs of legacy drag. A third party analyzed different options and concluded that rewriting the applications over the course of two years would actually be more cost effective and less risky than refactoring the existing code.

Strangling with APIs
When migrating legacy applications, you'll often find dependencies on services that you either can't change now, or at all. You'll want to isolate your application as much as possible from these immovable services so that you're not dependent on their own release schedules and idiosyncrasies. If you're lucky, you can slowly rewrite them as well.

The first step is to properly hide these legacy services behind APIs, shifting all of your code over to using those APIs that you control. At first, and perhaps forever, these will just be pass-through calls, but they give you an important architectural option you can selectively replace when those APIs are called. Now, this is a pattern as old as time in software development. What's important is to actively garden that option rather than thinking that one day you might just magically swap out the implementation.

The strangler pattern describes this gardening over time. You create APIs as a front end to legacy services and processes, then slowly start replacing various subcomponents of those legacy systems. At first, all of the actual work will be done by the legacy service, but just like a strangler vine, your new code will take over slowly until the legacy code is no longer used, rotting out and leaving just the new code. This slow and steady pace generally makes this a safe, time-conscious approach to working

with and then modernizing legacy services. This pattern is covered numerous places, including Matt Stines' book on migrating to cloud-native applications, which contains numerous other approaches for modernizing legacy architectures.

One variant of this looks to convert your ESB and SOAP-driven SOA services over to new approaches, like microservices, slowly but surely. Rohit Kelapure's paper on this topic describes a general approach and key tactics for this type of conversion. Comcast's Vipul SavJani and Christopher Tretina describe an approach they off-handedly call two layers of trickery to deal with similar issues.

Finally, dealing with data is often the most difficult process. How do you move from rigid, risky, and slow to change relational databases? The book Refactoring Databases describes several tactics. Kenny Bastani has described a clever approach that focuses on slowly changing your data over time, even moving it to new data stores.

In all of these cases, the first step is to isolate and hide the legacy services, while still using the service. This quarantining allows you to better manage how you interact with the legacy service. If you can swing it—or need to!—you can then focus on replacing, or simply augmenting, the legacy service and even data.

Avoiding Legacy Pitfalls

It's good to use the pain of dealing with legacy applications as a reminder to actively manage your portfolio and ensure overall code hygiene; that is, to avoid creating legacy problems through neglect. The best time to start flossing is yesterday, barring that, before all of your teeth rot out. Today is a pretty good time to start.

How did you get in this dilemma in the first place? Acquisitions are a common path, especially for large organizations. You end up not only with another organization's legacy software, but software that's so different and incompatible with yours that it has many of the same productivity and quality drag effects as legacy software.

More commonly, though, organizations have failed to put proper portfolio management in place that helps prevent legacy problems. If you're stuck in legacy, now's a good time to fix it. As in the discussion of agile software methods above, you should start by verifying that you're actually following any legacy management strategies you have in place.

Prioritize your portfolio

In the context of managing legacy, portfolio management means monitoring and managing the full life cycle of software. You should first have an inventory of all of the software in place—or the ability to create an inventory on demand. Next, you need to know who's the owner for that application and who's responsible for keeping it up and running. Finally, you'd like to know where the application is in its life cycle. Is it brand new, operating in a state of full usefulness, or sort of just puttering along? Knowing all of this—the software itself, the stakeholders,[16] the operators, and the current business value—will allow you to prioritize what you do with each piece of legacy software.

16 You could easily list dependencies as another required piece of knowledge. For example, you might be providing an identity and access management backend that has no end users, but is relied on by other applications and services in your portfolio. Here, I'd categorize dependencies as a mixture of stakeholders and business value, but if it's more effective for you to think of dependencies as its own thing, do that.

Obviously, software that sees heavy use should be well taken care of regularly. You should spend much preventative energy to keep the software as hygienic as possible (for example, keeping tests up to date so that integration builds run green and paying close attention to technical debt).

Lower-priority software should be treated accordingly as well. If it's of low enough value, ensure that you've fully virtualized the application, if not moved it to a managed service provider or a SaaS version of the application, if that makes sense. The goal with low-priority software in legacy management is to prevent it from slowing down your ability to create and evolve innovative software. It's just like paying down debt to free up money for investing in growth.

Turn it off and see who complains

If you're extremely lucky, there may also be applications that can be decommissioned. There may be applications that are kept up and running just to maintain access to the data locked up in their databases (for example, for regulatory reasons or historical analysis). Could these just be extracted to a general purpose database or even flat files, removing the need to maintain the actual application? In other instances, you may have software that simply isn't used anymore.

You could apply the tried-and-true pattern of monitoring for usage over a period of time (say, one to three months) and then, if there's no activity, turning the application off and seeing who, if anyone, complains. You could also follow a more mature approach and try to track down the owners, asking them if it's OK to turn the application off. For those owners that are reluctant to turn off applications, use another IT service management trick: charge an arm and a leg to keep running the out-of-date software.

Case Study: Avoiding Portfolio Paralysis Analysis

Unintended slowdowns, and thus, project failure, often come with the best analytical intentions. In legacy studies and projects that Pivotal has worked on with organizations, we've found that spending too much time studying and scoring your portfolio can actually be damaging and is a common anti-pattern. Instead, Pivotal recommends forcing yourself to operate at a quick clip at first, emphasizing getting things done; full cycle rather than eating the whole elephant of your portfolio.

Pivotal engagements are typically 10 weeks and their goals are to do the sort of on-the-job training that we discussed in the Getting Started section: to both get actual legacy applications migrated and train staff for the next round of migrations. These 10-week cycles are repeated as you eat through your legacy portfolio. Each time, there should be more organizational knowledge about how to migrate applications, and eventually, staff should become old hands at dealing with legacy.

As one example, a large financial organization quickly created a list of about 50 applications as candidates for migration to Pivotal Cloud Foundry, before narrowing down the list to 10 that were migrated during the 10-week project. During this time, Pivotal worked with staff members to familiarize and train them to use Pivotal Cloud Foundry, modern development techniques, and legacy migration tactics.

Principles for migrating apps to cloud-native platforms
After numerous engagements like this, Pivotal recommends the following six principles when tackling your legacy portfolio:

- **Plan just enough to start.** Don't do, or analyze and plan for everything at first.
- **Start with "one thing".** As ever, start small and build up over time as you learn.
- **Break big things into small things.** Decompose the monolith, as they say.
- **Automate everything you can.** Automation at all levels is the source of most cloud-native magic.
- **Build skills by pairing and doing.** Pairing an experienced staff member with a new one, and rotating them, is key to quickly learning and then diffusing knowledge.
- **Let real work inform strategy.** As you learn what's possible and gain confidence, you can go back and ensure that your overall strategy is realistic and pragmatic.

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 unknown 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 it takes to deploy just one line of code to production. 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. Some organizations find that it takes even longer and it's common to hear that teams going through this exercise just give up after waiting too long.

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. As discussed in other sections, gaining the trust of auditors, security experts, and other third-party gatekeepers, requires building up trust in a repeatable, standardized stack of software.

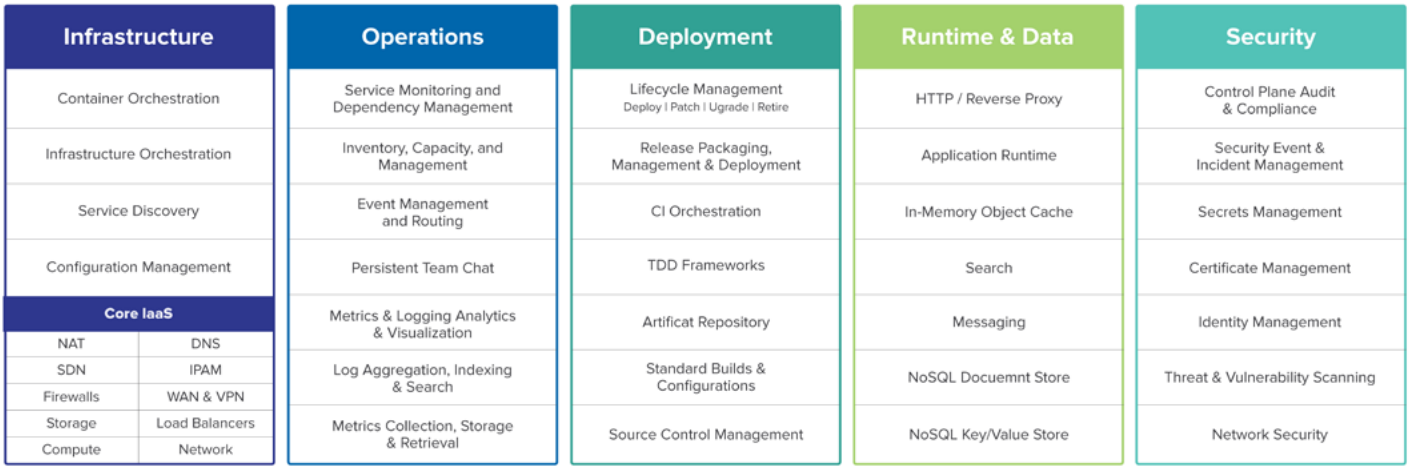
These are all the day one problems of getting the first version of your software out the door into production. After that come the day 2+n problems of managing that application and updating it with new releases.

17 This reference architecture and the description of each column is from Matt Walburn.

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. Below is one reference architecture based on conversations that we've had with organizations planning their cloud-native transformations.[17]

Reference Architecture for Cloud Native Plaforms



Any good cloud platform will have deep capabilities in these five domains:

1. **Infrastructure.** In the cloud era, 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 the platform and its applications can be run on and even moved across any provider.
2. **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.
3. **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.
4. **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.

5. **Security.** The notions of enterprise security compliance and rapid velocity have historically been at odds, but that no longer has 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 may 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 with 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 as such. Many Pivotal customers started off building their own platform, sometimes because 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 (as just described), or giving into 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 $14m, 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 resources (time and money) spent building and maintaining a platform the best use of those resources relative to, say, building the actual business applications?

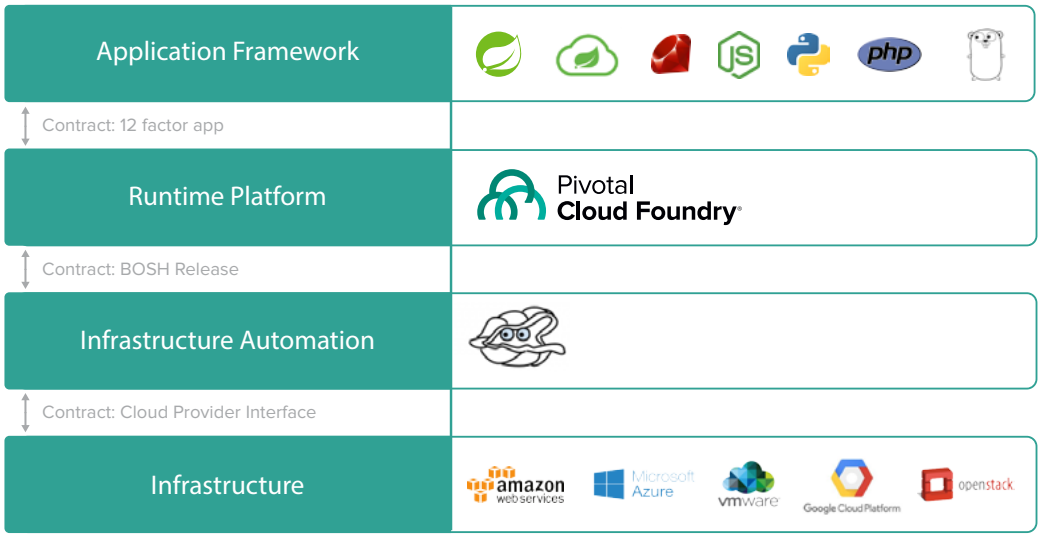
In my experience, organizations that decide to roll their own platform quickly realize a pesky truth: it's more expensive than they thought. Their investment in platform engineers grows faster and higher than projected. First, selecting and understanding which components to use for each part of the platform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

Typically 50+ People for Staffing a Platform Product

Each sub system demands multiple engineers and a product manager, and also staff to coordinate the whole effort—just like a real product! In working with numerous large organizations, we've found that even a minimal do-it-yourself platform team can consume two years of time and $14 million in payroll, across 60 engineers. Worse, these organizations will have to wait for two years to start their cloud-native transformation in earnest because they have to build the platform first, before they can get back to the original problem of building business applications.

Pivotal Cloud Foundry

Pivotal Cloud Foundry, Layers, and Capabilites



Instead of building and maintaining their own platforms, many organizations are using cloud platforms such as Pivotal Cloud Foundry. The Pivotal Cloud Foundry platform comes fully integrated and full of services and middleware that allows product teams to start in minutes once the platform is up and running. Because there's a full company's R&D force and the larger open source community around Cloud Foundry, updates and patches are frequent and new features are added regularly. For example, Pivotal Cloud Foundry supports numerous programming languages, comes with an ever-growing suite of integrated services and middleware suites, and runs on all popular infrastructure layers such as AWS, Google Cloud, VMware, Microsoft Azure, and OpenStack, whether in public or private cloud.

Most of the organizations discussed, including The Home Depot, Comcast, HCSC, Liberty Mutual, Allstate, Citi, and others, are using Pivotal Cloud Foundry as their cloud platform. This allows them to allocate the vast majority of their resources to what's actually valuable to their organizations; not to general purpose platforms, but rather to business-specific applications.

GETTING STARTED

Every journey begins with a single step, they say. 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.

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 want to quietly sweep it under the rug so no one notices. Few things will ruin the introduction of a new, proven 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, as we saw with the HCSC example, a string of small victories will make scaling success easier.

PROJECT PICKING PECCADILLOES

Your initial project, or projects, 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 and also technically feasible for cloud technologies. These shouldn't be science projects or automation of low value office activities—no virtual 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 migrate the .com site. Christopher Tretina recounts Comcast's initial cloud-native ambitions in this way:

> 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 only do that 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 roll-out.

"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 chosen were projects like managing (and charging for!) late returns for rented tools and running the custom paint desk in store.

A special case for initial projects is picking a microservice to deploy. 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. The microservice could be something like a fraud detection or address canonicalization service. This is one approach to migrating legacy applications in reverse order, a strangler[18] from within!

Picking Projects by Portfolio Pondering

There are several ways to select your initial projects following the criteria described. Many Pivotal customers use a method perfected over the past 25 years by Pivotal Labs called discovery. In the abstract, it follows the usual BCG matrix approach, but builds in intentional scrappiness to ensure that you can quickly 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.

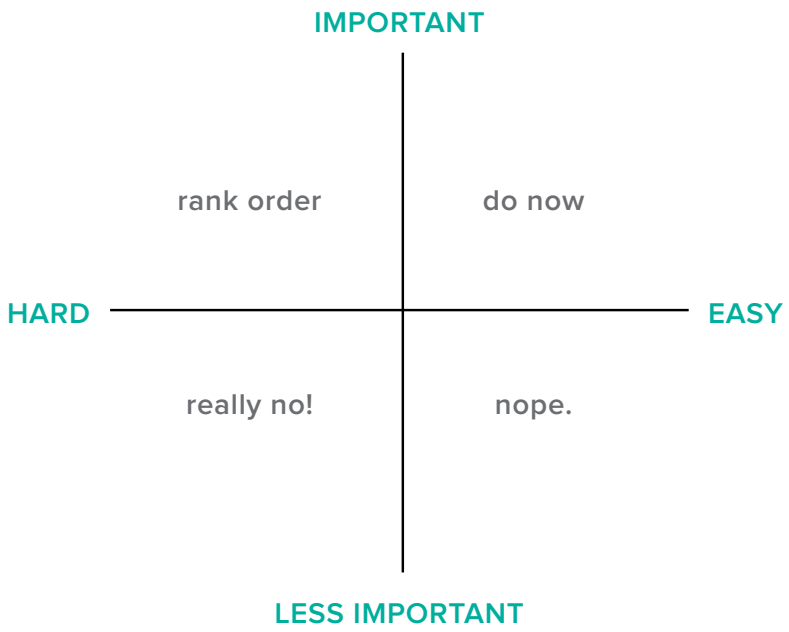
First, gather all of the relevant stakeholders. This should include a mix of people from the business and IT sides, as well as the actual team that will be doing the initial projects. A discovery session is typically led by a facilitator, usually a Pivotal Labs person familiar with coaxing a room through 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" is 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?"[19] Start from there, maybe with some "five whys" spelunking.

18 The strangler pattern and how it's applied to migrating legacy services is covered in the legacy section.

19 This is based on 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.

Once the stakeholders have written down projects on their sticky notes, the discovery process facilitator draws or tapes up a 2x2 matrix that looks like the following:



Participants then put up their sticky notes in the quadrant, forcing themselves not to weasel out and put the notes on the lines. Once everyone has done this, you get a good sense of projects that all stakeholders think are important, sorted by the criteria I mentioned. Are they 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.

At the end, you should have a list of projects, ranked by the consensus of the stakeholders in the room.[20]

Planning Out the Initial Project

You may 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 break down tasks into stories,[21] iterations, and releases. More than likely, following the general idea of a small batch process you'll:

- 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;

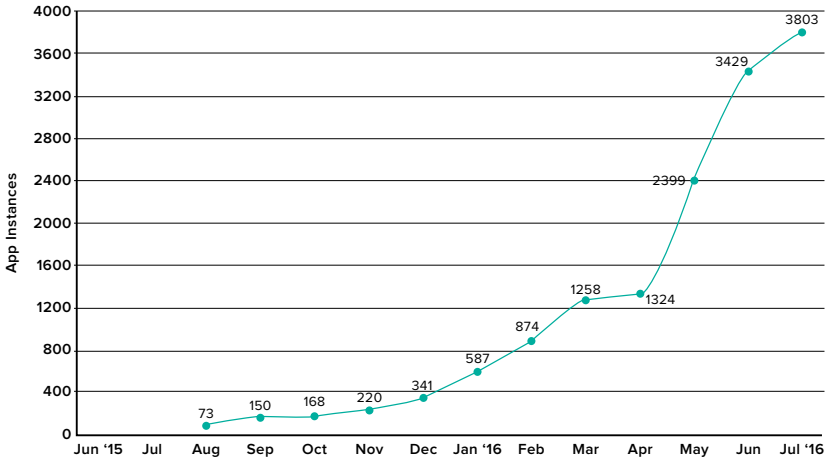
20 While written to help organizations pick advanced analytics projects to pursue, BCG has a good article that captures this general type of thinking, especially with respect to how leadership can think through the strategic parts of this process.

21 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 encapsulate 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.

- come up with several theories for how those problems could be solved;
- Distill the work to code and test your theories into stories;
- Add in more stories for non-functional requirements (like setting up build processes, CI/CD pipelines, testing automation, etc.); and
- 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).

Crafting Your Hockey Stick

Starting small ensures steady learning and helps 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. This chart 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 as The Home Depot became more familiar with the approach and the platform.

Another Pivotal customer, this one in the telecommunications space, started with about 10 unique applications at first and expanded to 100 applications just over half a year later. These were production applications used to support millions of customer account management and billing tasks, hardly science projects.

CONCLUSION: IT'S EASIER THAN EVER TO STOP HITTING YOURSELF

> "We are uncovering better ways of developing software by doing it and helping others do it."
>
> – The Agile Manifesto, 2001

A cloud-native organization's goal is to provide its business with an effective, sustainable means of innovating. This is accomplished by using cloud technologies and practices to fully automate the infrastructure layers of the application stack. With a huge chunk of resources and responsibility freed up, product teams can finally attain the focus and release speed needed to apply a small batch approach to development that results in continually improving software. A cloud-native IT approach in place enables organizations to meet the innovation needs of the business.

Although the idea of how to predictably and consistently craft cloud-native organizations is still evolving, the case studies and anecdotes we've covered provide ample lessons from the first few years of early adopters. These methods will keep evolving and all of us in the community would do well to keep sharing our successes and failures.

In contrast to the clichéd perception of IT—always late, always over budget, and always under delivering—the cloud-native approach puts IT at the center of the organization's operations and innovation cycle. That's a chance that shouldn't be wasted on doing things the same old way.

When discussing transforming to the cloud-native mindset, I am too often met with pleas of, "That all sounds great, but it can never happen here. We're too screwed up." That defeatist stance, while depressing, is understandable given the mismatch between traditional approaches to IT the new expectations of how easy it should all be and the business demands in the era of transient advantage. Everyone wondering, "If Facebook and Netflix can do it, why can't we?" Switching to a cloud-native approach isn't easy, but it's certainly possible. And the alternative of further, large organization failure certainly isn't a viable option.

If you've read to this point, hopefully you realize it's possible—very possible!—to change, and there's an actionable body of experience and work at organizations like yours to help you learn and get started. All of this tangible, doable opportunity is what makes me excited about cloud-native technologies and thinking. I hope reading this booklet has sparked some enthusiasm for moving to cloud-native by helping you understand: the goals and methods; how to get there from here; and how other real-world organizations are proving that it's all possible. Good luck!

Let me know how it's going—you can always email me at cote@pivotal.io or find me in Twitter @cote.