**Chapter 13. Common Transformation Challenges**

Now it is time to look at some typical cloud native transformation challenges and scenarios. These are situations commonly found when a migration initiative has stalled or otherwise gone wrong. In fact we find these same situations so often, and across so many different types and sizes of enterprises, that we developed profiles to speed up analysis and enhance understanding. It is important to note that every organization has its own unique set of needs and circumstances, such that every transformation path differs in the details. Thus, these scenarios represent broader “big-picture” profiles, not analysis on a granular level.

Each scenario contains sample a Cloud Native Maturity Matrix graph representing how results would typically look when that situation is found. Remember [Chapter 6](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch06.html#tools_for_understanding_and_using), where we introduced the matrix as a tool for assessing organizational state and readiness for migration? If you have been playing along to assess and graph your own status, you can compare your results to the scenarios described in this chapter. Really, though, you would only find a match if you’re reading this book to find out why your cloud migration initiative is not going as planned. If you are at the beginning of the process, let this be a lesson in “what *not* to do when planning a cloud native transformation.”

After we explore each problem, we name a set of patterns that can be applied as part of a design solution. Bear in mind that these address only this specific problem at this specific stage. The set of patterns we describe then fits into an overall design—a larger set of patterns that holistically addresses the entire process and serves as the transformation design. See [Chapter 11](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch11.html#applying_the_patterns) and [Chapter 12](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch12.html#applying_the_patterns_a_cloud_native) for a start-to-finish roadmap.

**Too-Early “Lift & Shift” Move**

When we see the “lift and shift” mistake, the enterprise suffering the consequences is almost always taking a traditional Waterfall approach to software development (a company very much like WealthGrid, in other words). The organization may have adopted some Agile processes, such as work sprints, but the hierarchy and bureaucratic way of doing things that become ingrained in large companies are still in charge.

Companies like this tend to be comfortable with large initiatives and making big bets, once identified as a proper strategic goal for the business. They are willing to invest the resources to make a cloud native transition happen, but they are operating under the mistaken belief that it simply means migrating their existing system and processes onto the cloud. Unfortunately, there is little cloud native understanding or experience inside the company to guide more informed decision making. This is so common that it’s earned the name “lift and shift” and it categorically does not work if you try to do this at the beginning of a transformation. An organization has to change not only its technology but also its structure, processes, and culture in order to succeed with cloud native.

The “lift and shift” decision can come from the company’s engineers, who have been to some conferences and are eager to try out this cloud native stuff. They likely have set up an account on one of the big cloud providers—after all, it’s free to get started!—and have played around with setting up a container or two. From there they may have even built a small experimental app. It all goes pretty well, so they are confident they can manage a full-scale transition. Based on this, the engineers seek and get approval from management to move forward with a full-fledged cloud migration. (Sound familiar? This was exactly how WealthGrid undertook its first failed transformation attempt.) They’ll move to cloud infrastructure and split their existing monolith into some (but not too many—typically, four to six) components so that they can justify playing with Kubernetes.

As previously discussed in [Chapter 1](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch01.html#what_is_cloud_native_question_mark), companies with existing cloud native expertise or who have hired outside guidance for their transformation have gained real advantage from moving a monolithic system or application to the cloud first, before re-architecting everything to optimize for the cloud. The intent behind such a move, however, is to jump-start your journey to cloud native as a starting point for further transformation such as moving to a true microservices architecture. Yes, you get to shut down your on-premises data centers. But there is still serious work to be done in terms of evolving your processes, organizational structure, and culture to suit your new cloud platform.

The point is, the situations where lift and shift works are when it is done as a tactical move, not a strategic one. For example, sometimes a lift and shift could be exactly the right thing to do as an experiment, on a small scale, to understand more about the way a cloud vendor works. It’s not automatically a wrong move, it’s just wrong as a goal or an end move. Intent is key: lift and shift should not be a strategic direction, done at the very start as the core of your transformation strategy.

Or, alternatively, perhaps the transformation drive comes from the top down, when the CEO and board decide the company needs to go cloud native and hand it off to the middle managers as a large but not particularly difficult technical upgrade. The mistaken expectation in this case is that the company can throw money at the project and simply purchase a full solution from a major cloud vendor. After all, that was how it always worked in the past when it came time for a technology overhaul. It’ll be a rough two weeks while everything gets installed and people get used to the new tools, but after that smooth sailing.

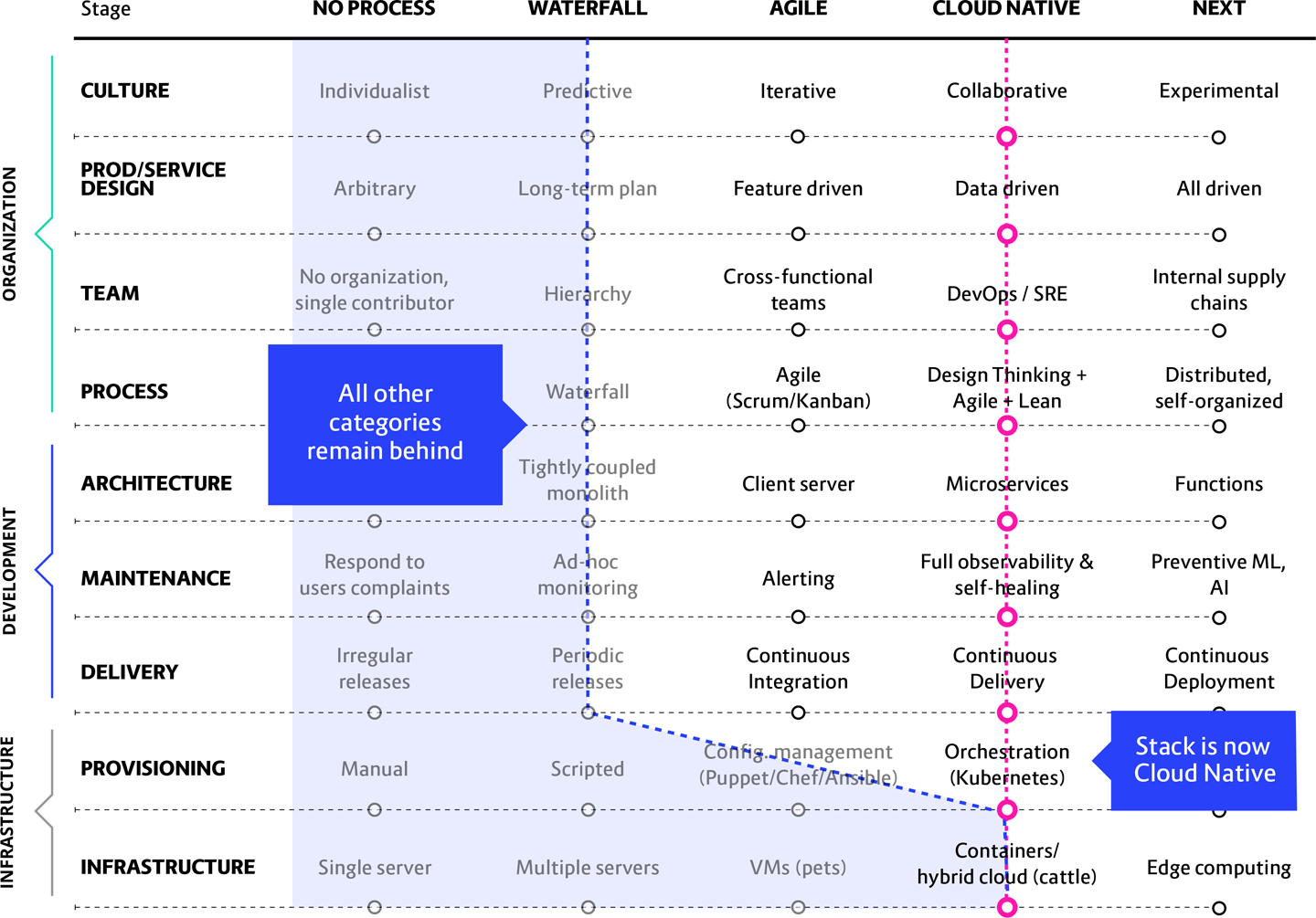
Both of these scenarios are based on the same understandable but fundamentally flawed assumption: Since their current system works well, they should just be able to re-create it on the cloud, right? (Vendors may even be promising that this indeed how the process will go. More or less. Emphasis on the less, but please sign here, thanks!)

In both cases the migration plan is essentially to take the old system (the “lift”) and move everything onto the cloud more or less intact (the “shift”).

In both cases, everything else, however—organizational and cultural aspects such as planning, design, development processes, etc.—remain unchanged. The technology goes cloud native, but the company itself is the same as it ever was.

In both cases, after struggling for months to move the old system—and, possibly, a couple million dollars or pounds or Euros later—the company is baffled to realize that, yes, they’ve got Kubernetes and cloud infrastructure…and nothing works. Or maybe things are even mostly sort of working, but not any faster or better than they did before.

Take a look at [Figure 13-1](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch13.html#lift_and_shift_the_whole_system) to see how this plays out when graphed on the Maturity Matrix.



**Figure 13-1. “Lift and Shift the whole system at the beginning” scenario as graphed on the Maturity Matrix**

**In this scenario we see:** A company with Waterfall or Agile culture, architecture, design, and processes that has attempted a “lift and shift” strategy for moving their current system (and comfortable way of doing things) onto the cloud. On the Maturity Matrix, Infrastructure has moved all the way to Containers/Hybrid Cloud while all other categories remain rooted around Waterfall/Agile status.

**What will happen:** Operating costs will go up. Few, if any, benefits will be gained. Time and money will be wasted: you now have a bunch of expensive complex tools on top of the same system you always had—but you are not faster or cheaper. Worse, you have missed the opportunity to use this motivation and momentum to make real, lasting changes to truly transform your organization and its processes.

**A better approach:** Focus on progressing all the points toward their respective cloud native states. For most organizations this likely means starting with Architecture and Process while simultaneously working to gradually evolve Culture.

So, again, doing a lift and shift is itself not a completely flawed approach. Early on, pursued as an open-eyed tactic rather than a strategy, it can have value. But doing a lift and shift as your *entire* transformation strategy is setting the stage for later disaster. In many migrations—toward the end, once the new platform is working in production—there may be parts of the original system that are still necessary and in place. If these legacy components are stable and seldom changed, they can indeed be moved to the cloud using lift and shift at the *end* of the initiative. See “Lift and Shift at the End” in the [Appendix A](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/app01.html#library_of_patterns) patterns library.

**Practical patterns solution:** An honest and thorough assessment of your current state is essential before moving forward in any aspect of a cloud migration. Once that is in hand, start with Vision First. (Presumably you already have buy-in and support from the CEO/board, but if not, do the Business Case and Executive Commitment patterns first). Next, name your Core Team. You will emerge from this initial set of patterns with a clear vision, a high-level architectural direction, and a team that is equipped to implement both.

**Treating Cloud Native as Simply an Extension of Agile**

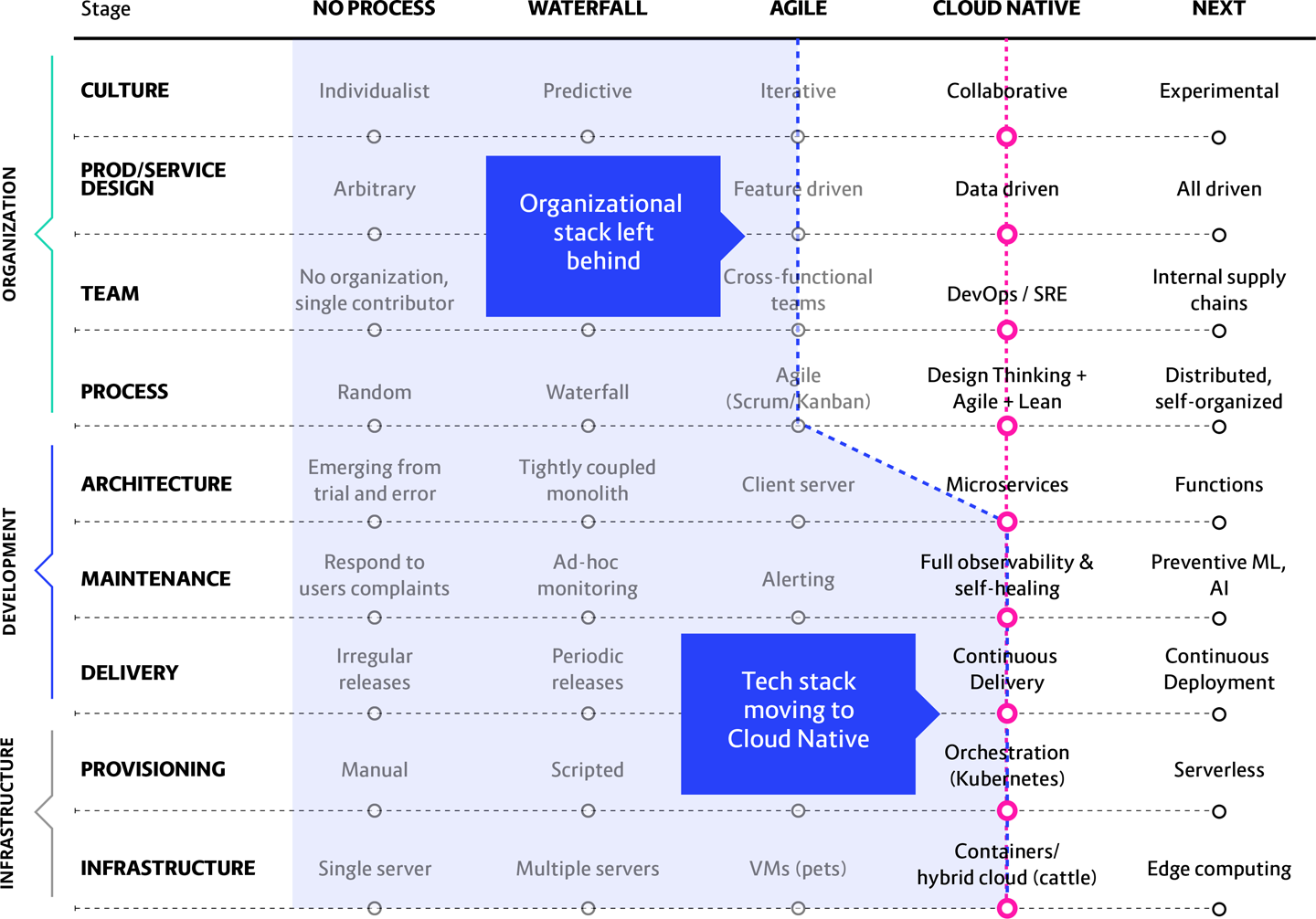
Changing your software development practices from Waterfall to Agile is a huge paradigm shift. Most companies understand that to remake themselves as truly Agile organization, they would need to alter a lot of the ways they do go about developing and delivering software, from building processes all the way down to where people sit. Unfortunately, they don’t extend this same understanding when it comes to moving the company from Agile to cloud native practices.

This particular scenario happens when a company that did a reasonably good job of transforming between Waterfall and Agile practices fails to recognize that moving to cloud native is a second true paradigm shift. Instead, they believe cloud native to be “just a new way to do Agile.”

Instead of being treated as a major and serious change in its own right, the cloud native transformation is treated as just a bunch of tech-related tasks that get added to the existing development backlog. As we saw in WealthGrid’s first failed attempt to deliver a new cloud native system, this approach doesn’t work.

This is similar to the Lift and Shift scenario, but not identical. In this scenario the organization makes more real progress toward building a cloud native platform but is still not able to deliver software on it. Worst case, pretty much nothing works well at all. Best case, things are basically working, but the real value isn’t there—the company isn’t getting the product velocity and data-driven design they just spent all that money to create. The overall feeling is that the new system is not much better or even worse than the old one. (And then they say, having no idea how much they have just missed out on, “Those containers aren’t such a great idea as we were thinking in the beginning.”)

[Figure 13-2](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch13.html#this_is_what_it_looks_like) nicely illustrates the somewhat subtle but significant difference between attempting “Lift and Shift” and treating your cloud native transformation as though it is just another technical upgrade.



**Figure 13-2. This is what it looks like when a system moves ahead of the people who must work within it**

**In this scenario we see:** The cloud native portion of the transition gets placed within the scope of Agile coaches and thus gets led by them. This is a genuine paradigm shift, but Agile sees it as only a simple platform iteration. Install Kubernetes, two sprints, done!

**What will happen:** This results in cloud native tools such as Kubernetes, CI/CD (Continuous Integration and Continuous Delivery), and Microservices being treated simply as normal extensions of the Agile approach, which they are not. You may be building small pieces faster, but if you are still releasing them all at the same time every six months as part of a tightly coupled monolithic app, you are completely missing the point of cloud native.

Waterfall’s predictive culture dictates long-term planning, where you don’t take steps unless you understand exactly where you are going: this is how you avoid risk. Companies that have adopted Agile manage risk by delivering more frequently and reducing the risk of failure to smaller pieces of work. However, even though you are now creating it faster, you are still creating something very predictable—from within a hierarchical organizational structure, with all of its bureaucracy and handoffs.

This is why Agile processes and culture are still not fast enough, nor responsive and flexible enough, to take best advantage of cloud computing. If Agile was able to do this, we would not have needed to invent a new/different approach through cloud native practices!

**A better approach:** This is a major shift for a culture used to a tightly coupled approach where everything is developed in parallel for delivery all on the same day. Doing a contained experiment first helps teams coming from an Agile background understand how they will need to think, and work, differently once they are no longer operating within a hierarchy-driven monolith. It then helps them evolve into a collaborative and distributed cloud native way of working.

**Practical patterns solution:** Start with *Vision First.* Name a *Core Team* and a *Platform Team.* Take small steps for a gradual transformation, starting by working on one small contained side experiment that may be very basic but encompasses the major elements of cloud native (*Exploratory Experiments; PoCs; MVP).* The culture of the platform team likely will begin naturally evolving to reflect the new platform, but this needs to be ensured.

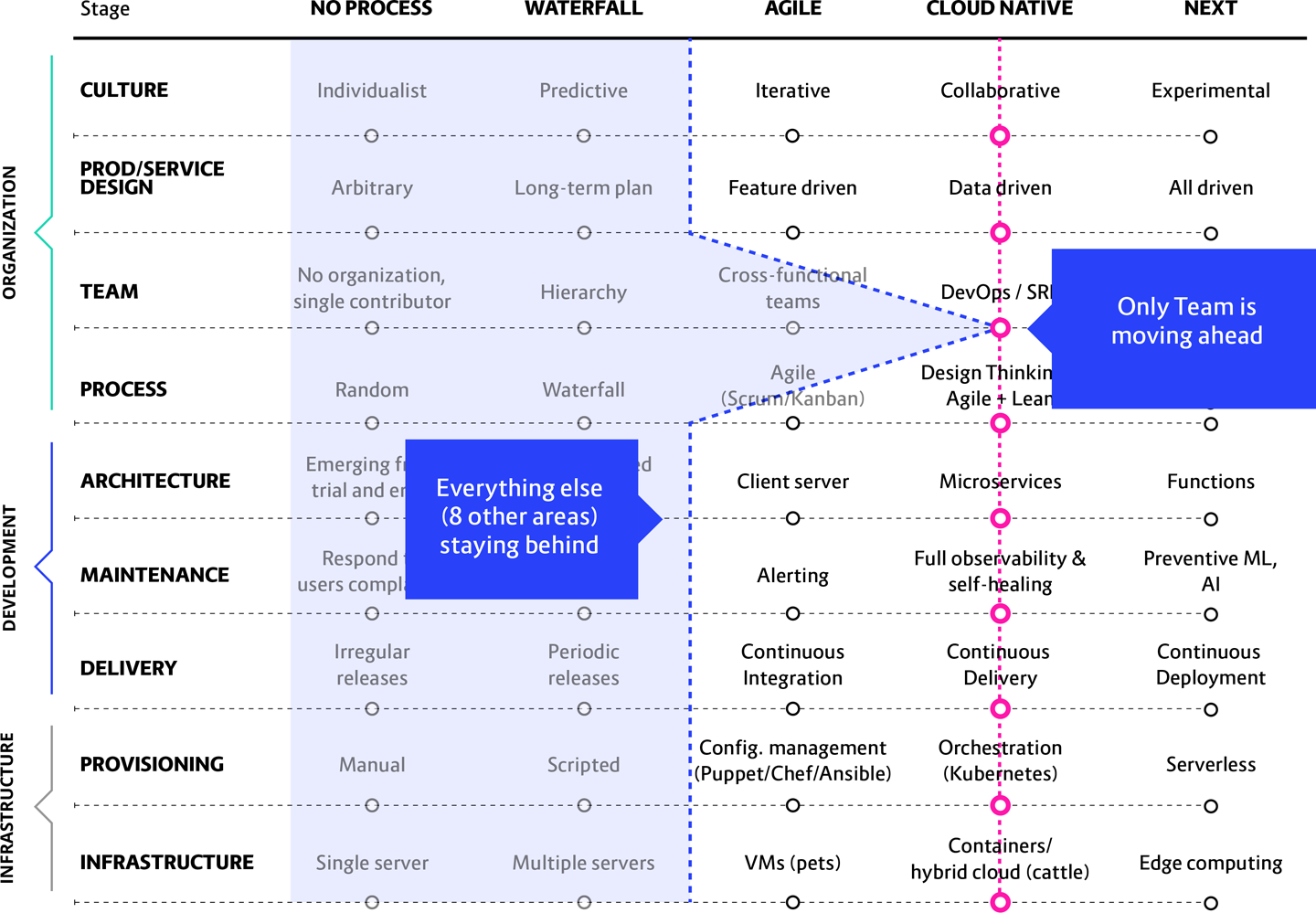
**“Spiking” Cloud Native Transformation via Unbalanced Approach**

This is actually the most common scenario we see—or, rather, the same situation caused by one of three factors. An unbalanced approach can happen at any point on the Maturity Matrix. When we are called in to help with a migration gone wrong, however, we typically find an organization that has moved significantly forward in only one of three areas—Microservices, Orchestration/Kubernetes, or DevOps—without also moving forward in the other related areas crucial for keeping a cloud native transformation on track.

In this scenario, when you draw the Maturity Matrix line connecting all the axes, we find a single sharp “spike” in an otherwise uniform line. The particular problematic “spike” area can vary, but the final outcome is the same: the migration will bog down and is likely to ultimately fail.

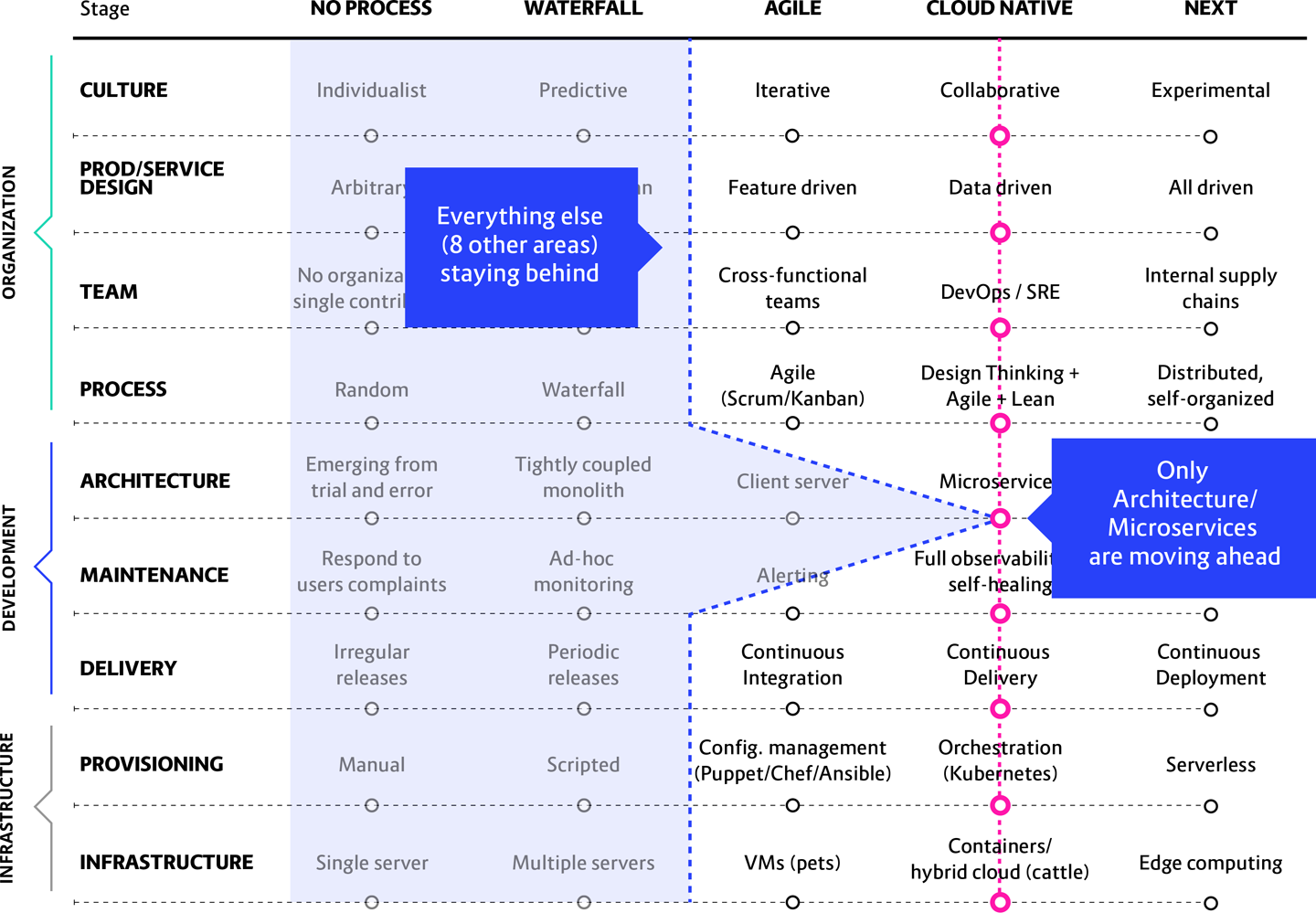
This almost always happens when the initiative is started in, and driven from, a single faction in the company. No other groups are participating, and so the migration progresses happens only in the area that affects the originating group. The members of that faction unilaterally proceed with the migration, doing only what is important from their point of view—without taking into consideration any other stakeholders or the role of every other division in the organization.

If the transition is top-down, with the CEO and board deciding the company must go cloud native, you are going to see teams moving to DevOps. Adopting a DevOps approach is a good thing, but in this case they don’t really understand what it means. Management will simply restructure things to add Ops people to the Dev teams and be done with it. Other than the seating assignments, nothing else really changes—process, delivery, and product/service design practices continue as of old. And once this one arbitrary idea has been enacted, they have no strategy for the future…and still no idea what a cloud native transformation means or how to get there. [Figure 13-3](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch13.html#when_an_unbalanced_migration_is_driven) illustrates this.



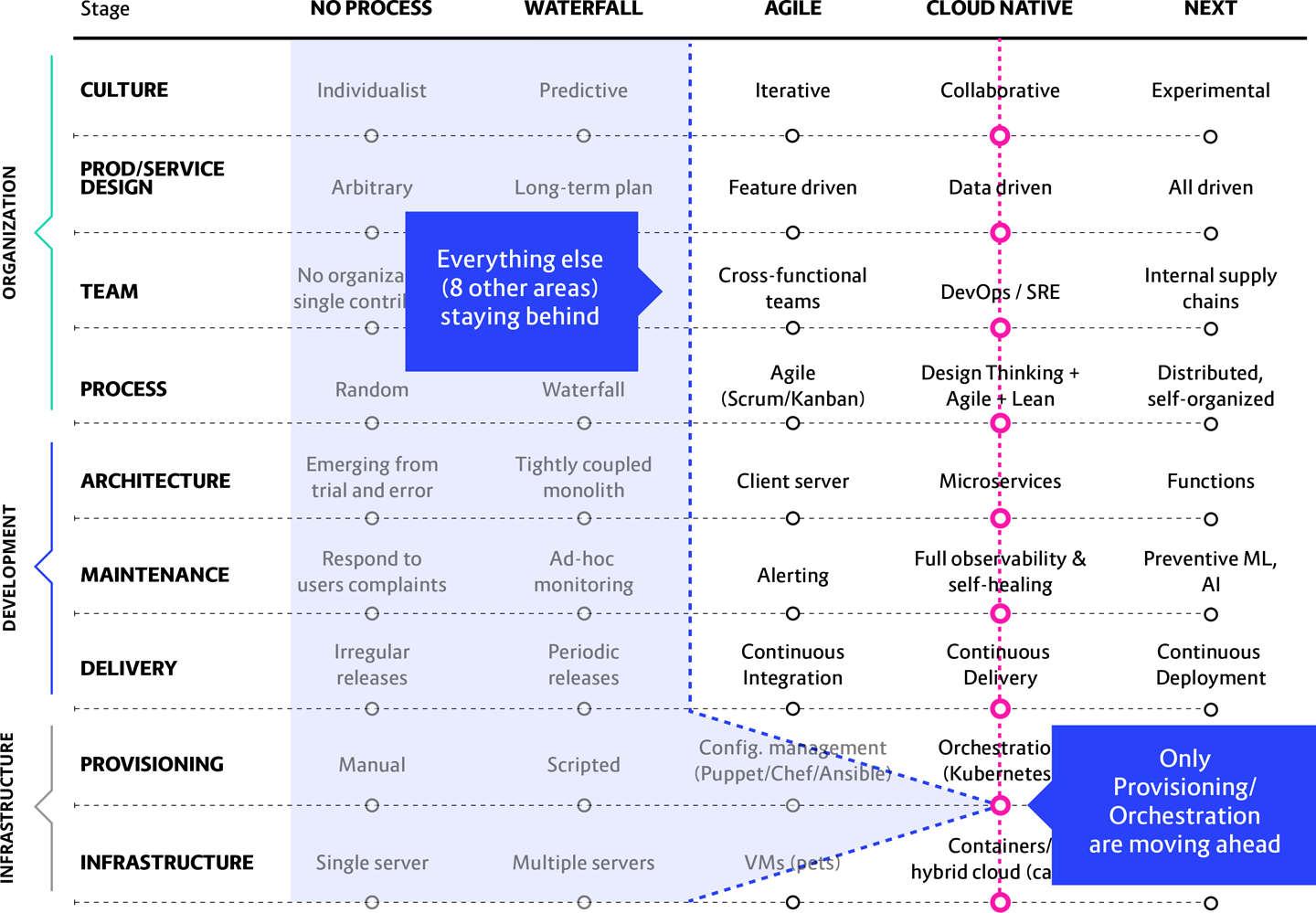
**Figure 13-3. When an unbalanced migration is driven from the top down, and only concerning organizational structure, the Teams are told to start doing DevOps while everything else stays unchanged**

If the move to cloud native is being driven by developers, however, [Figure 13-4](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch13.html#if_a_migration_gets_initiated) shows that we see a surge ahead to Microservices while the rest of the company stays behind.



**Figure 13-4. If a migration gets initiated by the developer teams, the devs surge ahead to building with microservices architecture while all the areas that need to support them remain unchanged**

If that group is Ops, they say, “We need Kubernetes” and the Provisioning/Orchestration axis leaps ahead of where the whole rest of the company finds itself left behind on the Maturity Matrix, as seen in [Figure 13-5](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch13.html#when_an_unbalanced_migration).



**Figure 13-5. When an unbalanced migration is driven initially by the Ops team, they embrace Kubernetes while the rest of the company keeps on building the way they always have**

**Even when the isolated initiative in question is actually going well, this is still a problematic scenario.** For example, consider a situation where the development teams decide to try out microservices without engaging or even informing other important stakeholders, like the operations part of the company. Things eventually become problematic because, while the engineering team may understand microservices and are implementing them properly in their own limited setting, the rest of the organization is not involved. And the engineers don’t know enough about cloud native architecture to consider the impact of doing microservices on the rest of the organization.

What they fail to consider is that they will have nowhere to deploy those microservices. Since they were operating in isolation, there are no Ops teams deploying and maintaining the target clusters. So, either it will stall or the devs will provision poorly configured clusters without talking to Ops. No one will be able to maintain such a mess.

The main problem, though, applies no matter who causes the imbalance: If there is no reorganization, then Conway’s law dictates that the way microservices get divided up will inevitably come to mirror the company’s organizational structure. Which will lead to the eventual and wholly unintentional re-creation of a monolith exactly like the one they have abandoned.

**In this scenario we see:** A company has decided to move onto the cloud and adopts one new practice or technology (Microservices, DevOps, Kubernetes) without considering the impact upon all other areas in the organization. This one area looks like a spike on an otherwise uniform matrix line.

**What will happen:** These initiatives rarely lead to success because teams very quickly run into all kinds of problems they never considered. Gridlock ensues: the teams know they have a problem, but they don’t know enough to even begin to understand what to do about it.

**A better approach:** Senior managers or executive/board leadership needs to intervene and define the cloud native transformation as a company-wide priority. Bringing all branches on board to work simultaneously on migrating their particular area of responsibility.

**Practical patterns solution:** The first need is to establish full support for the transformation, including dedicated resources, a defined vision, and a (high-level, not too detailed at first) plan for proceeding, so turn to the *Executive Commitment*, *Vision First,* and *Reference Architecture* patterns, plus the *Core Team* to execute everything. From there, we apply patterns for establishing the architecture and all major design decisions: *Exploratory Experiments, PoCs, MVP*. The neglected stakeholders also need to be engaged, so this calls for the *Involve the Business, Transformation Champion* and *Internal Evangelist* patterns.

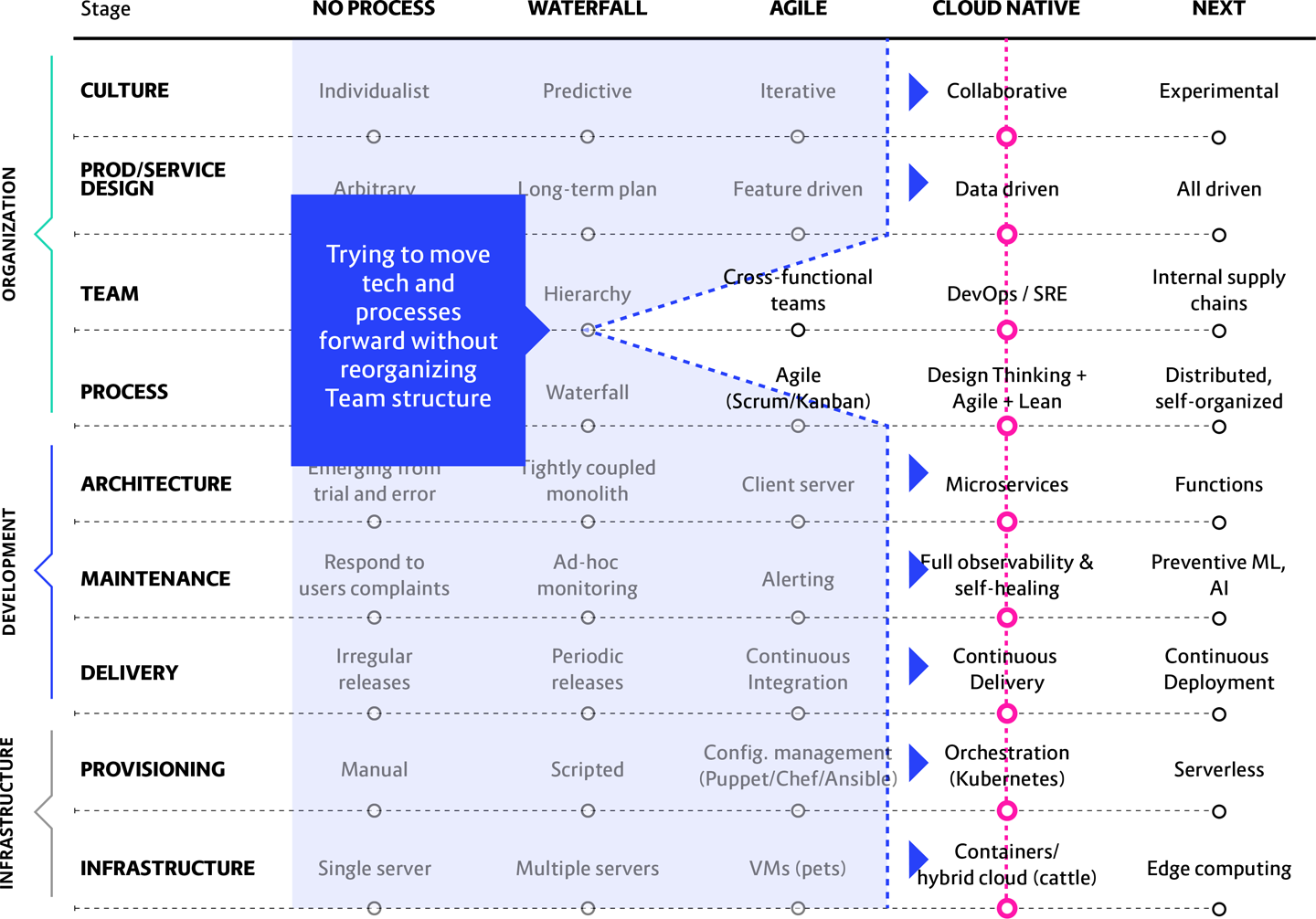
**New System, Old Structure**

Sometimes a company tries to go cloud native in every area except Team. We mean “Team” in the Maturity Matrix sense: how responsibilities are assigned within your organization and how communication and collaboration happen—or maybe don’t. Unfortunately, scoring even eight out of nine on matrix axes achieved is not a passing grade when it comes to a cloud transformation.

This particular challenge arises when, during the transition, your culture fails to evolve along with your new cloud technology. Typically it happens in one of two ways, both of which are artifacts of Waterfall culture wrongfully kept in place. Either you are retaining a very strong organizational hierarchy or keeping your teams specialized—grouped by skill instead of according to which piece of the system they deliver. Cloud native requires restructuring teams to become cross-functional, “you build it, you run it” groups capable of developing, delivering, and supporting a service from creation to production. In other words, capable of building microservices-based applications, and also defining infrastructure needed for those microservices in the process. (Think of it as microservices and infrastructure as code.)

You can have all the other areas in place and running smoothly, even the previous scenario’s problem areas of Microservices, Kubernetes, and DevOps. Unfortunately, no matter how well the rest of the system has been built, you are holding onto the mistaken belief that you can completely modernize your tech stack on the cloud, yet still produce things exactly the same way you’ve been doing for decades.

This scenario is the inverse of the previous one. Here we see a kind of “reverse spike” where all areas of the company have moved forward in the cloud native transformation process except for one single area, Team, remaining stubbornly behind. [Figure 13-6](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch13.html#shows_a_quotation_mark_reverse_spike) shows how striking this looks.



**Figure 13-6. A “reverse spike” of a Waterfall organization that has actually succeeded in moving toward cloud native in every area except how their people actually work to build things: Team.**

If your matrix graph looks like this, it means either you are attempting to deliver microservices using a fully hierarchical Waterfall process or that your process still requires handoff between specialized teams rather than each team delivering fully and independently.

Attempting to apply microservices (also true of DevOps) in a hierarchy might actually look like it’s working at first. After all, you are probably successfully delivering something on the new system. But all you’re really doing is streamlining a sub-process! You have in no way shortened the end-to-end release cycle, and you are definitely not delivering faster to the customer. Attempting this is an example of siloed thinking, a mistaken belief that optimizing the parts automatically means optimizing the whole. But your enterprise won’t find transformative business value unless you optimize *all* the areas on the Maturity Matrix.

Likewise, keeping your teams highly specialised—divided by skills like database administrators, QA, backend, front end, etc.—creates dependencies during the delivery stage. So attempting to build a distributed system of many small independent services using highly interdependent teams of specialists, none of which can deliver without handoffs to all the *other* specialist teams, goes completely against cloud native principles.

Unless that is you actually want to harness the advantages of this cloud computing system you just spent so much time and effort to create, in which case you’re going to need to restructure your teams.

**In this scenario we see:** Classic Conway’s law in action, i.e., the principle that system architecture will come to resemble the structure of the organization that contains it. In other words, Conway’s law describes how your system architecture will eventually and inexorably come to resemble your org chart: if the teams are organized based upon their specialized functions, eventually some sort of layered (and likely hierarchical, to boot) structure will emerge.

What this means is, if you attempt to introduce microservices without changing the organizational structure, over time the microservices will tend to drift together. Inevitably tight coupling will be introduced in one (or more) of the develop, build, or deploy stages. This will lead to more coordination and reduce the independence of the teams and—hey, look, a monolith!

**What will happen:** Basically you just re-created your previous monolith, only now on the cloud. Yes, it has microservices and containers and Kubernetes, but you used them to build a hierarchical system with very strong dependencies at the lower levels, single-specialty teams, and potential bottlenecks at each point in the process. There must be a handoff at each stage of progress, slowing things down even more.

**A better approach:** If you’re going to go cloud native, it means completely restructuring your codependent, highly specialized Waterfall teams into collaborative DevOps teams. Each fully functional team includes planning, architecture, testing, development, and operations capabilities for whatever service it builds. It does not mean that they are responsible for the cloud native platform itself—that, of course, is built and run by your platform team.

**Practical patterns solution:** Apply the Build-Run Teams (“cloud native DevOps”) pattern. This ensures that each team has its own planning, architecture, testing, development, and operations capabilities and that there is collaboration both within the same team and across multiple teams. Most importantly, there needs to be a Platform Team building the platform to run microservices and containers, and separate teams of developers building the application(s). This way, Devs can deploy directly and automatically to the platform without any need for handover to Ops.

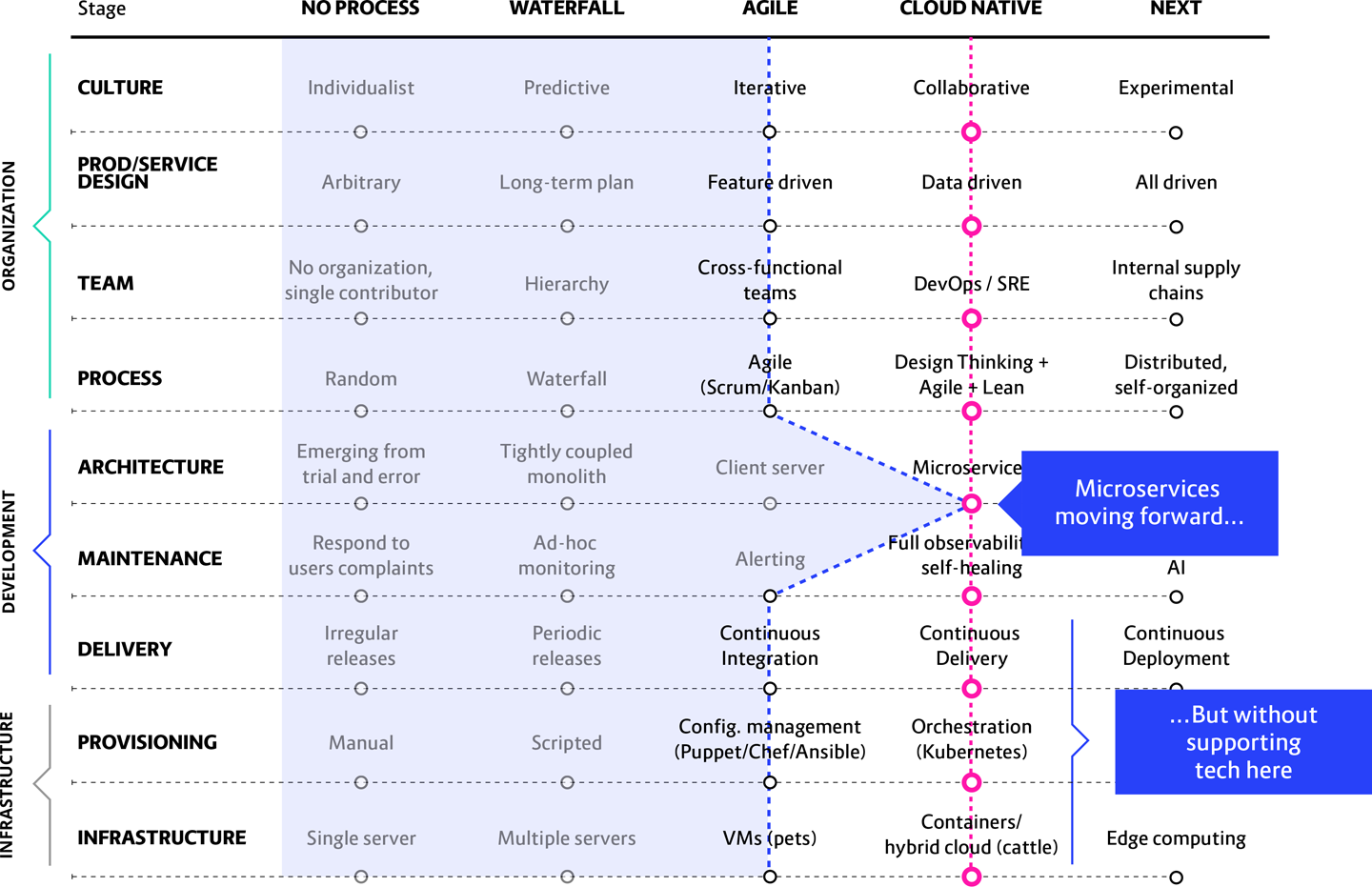
These teams will also work best when they have Personalized Relationships for Co-Creation and the patterns for managing Co-Located Teams (or at least how to optimize Remote Teams) are applied to make this happen.

**Wrong Order of Implementation**

Waterfall organizations are pretty good with order of implementation in general, in large part due to their predictive culture. Unfortunately, many enterprises moving to the cloud don’t consider the interactivity between cloud native elements—again, they don’t know what they don’t know! What they *do* know is they want microservices and so they move ahead on that sole axis, unaware of the havoc this can bring when implemented without also doing containerization, orchestration, automation, etc.

This can happen in a number of different ways, but the most common ones we see—as well as the most damaging—are when microservice architecture is launched either before CI/CD (Continuous Integration and Continuous Delivery) are in place or when companies try to run microservices without packaging them in containers.

When this happens, the result is literally hundreds of mini-monoliths running simultaneously—all of which need to be deployed manually. A nightmare, basically, which is happening because once again a monolithic team structure is expressing itself, via Conway’s Law, as a large number of strong inter-team dependencies. [Figure 13-7](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch13.html#what_happens_when_you_try) does not fully depict the chaos.



**Figure 13-7. What happens when you try to implement cloud native application architecture without also adding cloud native tools, technologies, process, and culture**

**In this scenario we see:** A typical non-cloud native company struggles to deliver a few components to production every few months, or maybe every few weeks if they have been moving toward Agile. They decide to move to microservices without ever considering the need to have other cloud native components in place to manage or run them.

**What will happen:** The number of components (microservices) goes up dramatically (from 5–10 to 100–300). The time required for delivering them, meanwhile, drops dramatically (down from months or weeks to days—or even hours). Juggling this many components in a greatly reduced timeframe means both quality and system stability are significantly affected. Trying to fit hundreds of microservices to a non-containerized platform without a high level of automation will be at best highly inefficient, and at worst possibly even disastrous.

The main result is that releases will inevitably end up becoming coordinated and scheduled as a natural consequence of needing to manage this myriad of microservices. Then, as the delivery process grows ever more painful and unwieldy, these coordinated and scheduled deliveries will start to happen at longer and longer intervals. First you just release at the end of the day, to keep things tidy; then, once a week because interdependencies seem to be growing and those take time to sort, and then even less frequently. This utterly defeats the purpose of cloud native, namely, reducing dependencies to go faster. Effectively, without the ability to deliver each microservice independently, quickly, and frequently, the application will over time develop into a monolith.

**A better approach:** With so many moving parts, reducing dependency on the infrastructure is essential. In such circumstances containerized packaging is essential, as well as implementing full automation around it. Simultaneously you need to cleave any dependencies between teams so that each one is able to move to delivery independently as soon as their code is ready.

**Practical patterns solution:** Choose one small application as an experiment (*Proof of Concept)*. Assign a *Core Team* to do the research necessary (*Exploratory Experiments, Research by Action)* to steadily progress the entire application across every axis of the Maturity Matrix. Not just Architecture/Microservices, but also putting in place containers, orchestration, CI/CD, and the kind of non-hierarchical *Build-Run Teams* culture that supports working in such a vertical environment. After establishing these, you should be in good shape to step back into a typical transformation design as outlined in [Chapter 11](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch11.html#applying_the_patterns) and [Chapter 12](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch12.html#applying_the_patterns_a_cloud_native).

**Platform Not Ready for Production; Going to Production Anyway**

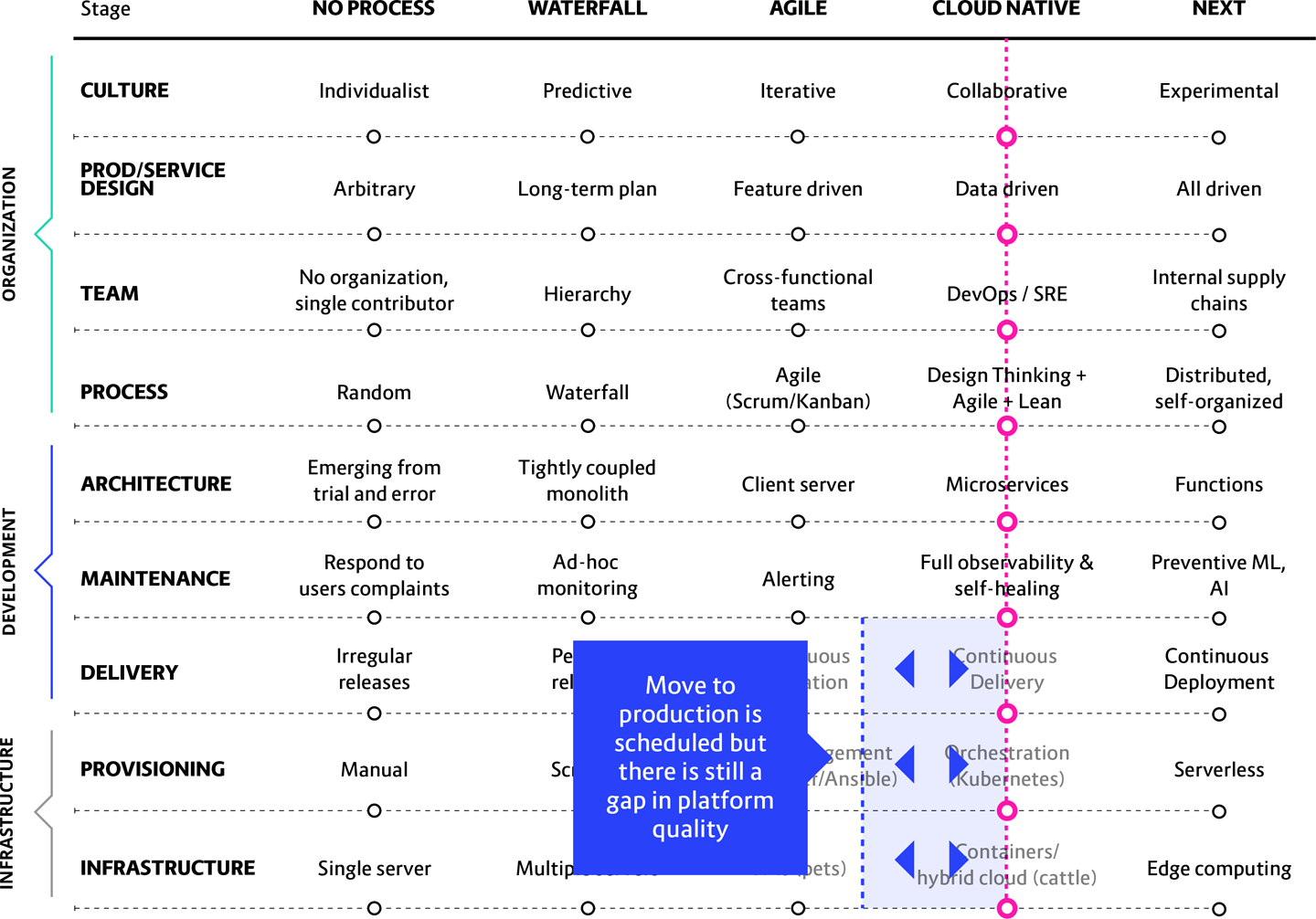
Public cloud platforms like AWS and Azure, as well as on-premises platforms like OpenShift, are presented as full end-to-end solutions, but beware: they are far from plug-and-play. Companies used to the full functionality of VMware (or any other fully mature tech solution) are unpleasantly surprised to find that configuration is partially manual, and not all the necessary pieces are even present.

Recall our metaphor from [Chapter 1](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch01.html#what_is_cloud_native_question_mark) where you buy a TV, bring it home, and open the box to find there are no cables, no power cord, no remote. None of the peripherals that are essential for operating the television set: it’s literally just a screen. You grab the directions only to read steps like, “Now go buy this piece here, that piece there, and assemble your TV—eventually, it will be fine!” So it is with cloud platforms: they claim to be full solutions, but half of the necessary stuff is missing. Because you believed your vendor that its platform was a full solution, you haven’t allocated people or budget or time to build or buy the missing or incomplete pieces.

The worst example of this scenario is when teams get ready to run microservices in production—or even already have them in production—but fail to prioritize strong observability and monitoring. They likely have some sort of event-based alerting set up, because that is how things have always worked before. In the past, when things went down, people would go into the servers to fix it locally. Once you move to MS and containers, however, you no longer have access to local storage. Containers go up and down in seconds, leaving no trace of themselves. Now you need dynamic monitoring with anomaly detection. If you are relying on traditional alerting triggered only when an actual problem occurs, by the time you log in, there will be zero evidence of what just happened.

Microservices, which run as many independent units, in particular must be well monitored and fully logged. Manual maintenance of microservices is costly and deeply impractical and of course you’re not going to do it. However, configuring your out-of-the-box platform to take care of maintenance tasks for you automatically is complex and beyond the abilities of inexperienced engineers.

[Figure 13-8](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch13.html#build_your_system_all_the) is a bit bare, but this is for a reason: no matter what the rest of your matrix looks like, if there is a gap in platform quality, your cloud native system is not going to launch.



**Figure 13-8. Build your system all the way to completion before you try to run it, OK?**

**In this scenario we see:** You have a bare platform with limited monitoring, security, storage, networking, etc., that is not ready for production. Or, interestingly, we also sometimes find that a company does install the proper tools—but then never checks to see if they work. The real truth is that nobody cares about monitoring until two days before going live with production. It’s the same with security and other functional requirements.

**What will happen:** Things go wrong. Many, many things. No one has any understanding what happened, or why. In many cases an automated response to a detected anomaly gets executed before the system’s human handlers are alerted, making it even harder to track down what went wrong.

**A better approach:** Harden your tools. Build in dynamic monitoring and anomaly detection—not just event alerting—from the beginning. Do a security review and implement load testing. Create full observability, including collection of all logs in a fully accessible central repository and status dashboards visible to everyone. Take advantage of cloud native’s “self-healing” abilities, with constant health checks offering auto-restart in the event of failure. MONITOR. ALL. THE. THINGS.

**Practical patterns solution:** Bring the platform to a stable and production-ready stage in one complete prototype before full rollout to the entire organization.

The approach we recommend is to have a dedicated *Platform Team* that builds an *MVP* first, incorporating all major components at a basic functional level. Once the platform is ready, then it is time for *Gradual Onboarding* of a couple (literally, one or two) advance teams and improve the system based on their experience to build your *Developer Starter Pack*. The final step is to gradually roll it out to the entire organization while running an extensive educational program to make sure that developers can use the toolset effectively—and incorporating feedback as they learn so as to further refine the platform for long term use.

**The Greenfield Myth, or, the “All or Nothing” Approach**

In this scenario we see a Waterfall organization with a deep legacy codebase—say, for example, a mainframe running COBOL. An enormous amount of work and organizational change are required to refactor any monolithic system into a flexible, functional cloud native one. The understandable temptation is to simply scrap the old system completely in favor of building a brand new one from scratch. A complete greenfield project must be cheaper, faster, and more efficient than trying to work with the existing one, right?

The COBOL example is a bit dramatic, though we have certainly seen it. The point is, though, that lots of people think this way. There is also talk of having a cloud native greenfield advantage being the reason that disruptor companies are so successful, so fast, when they pop up.

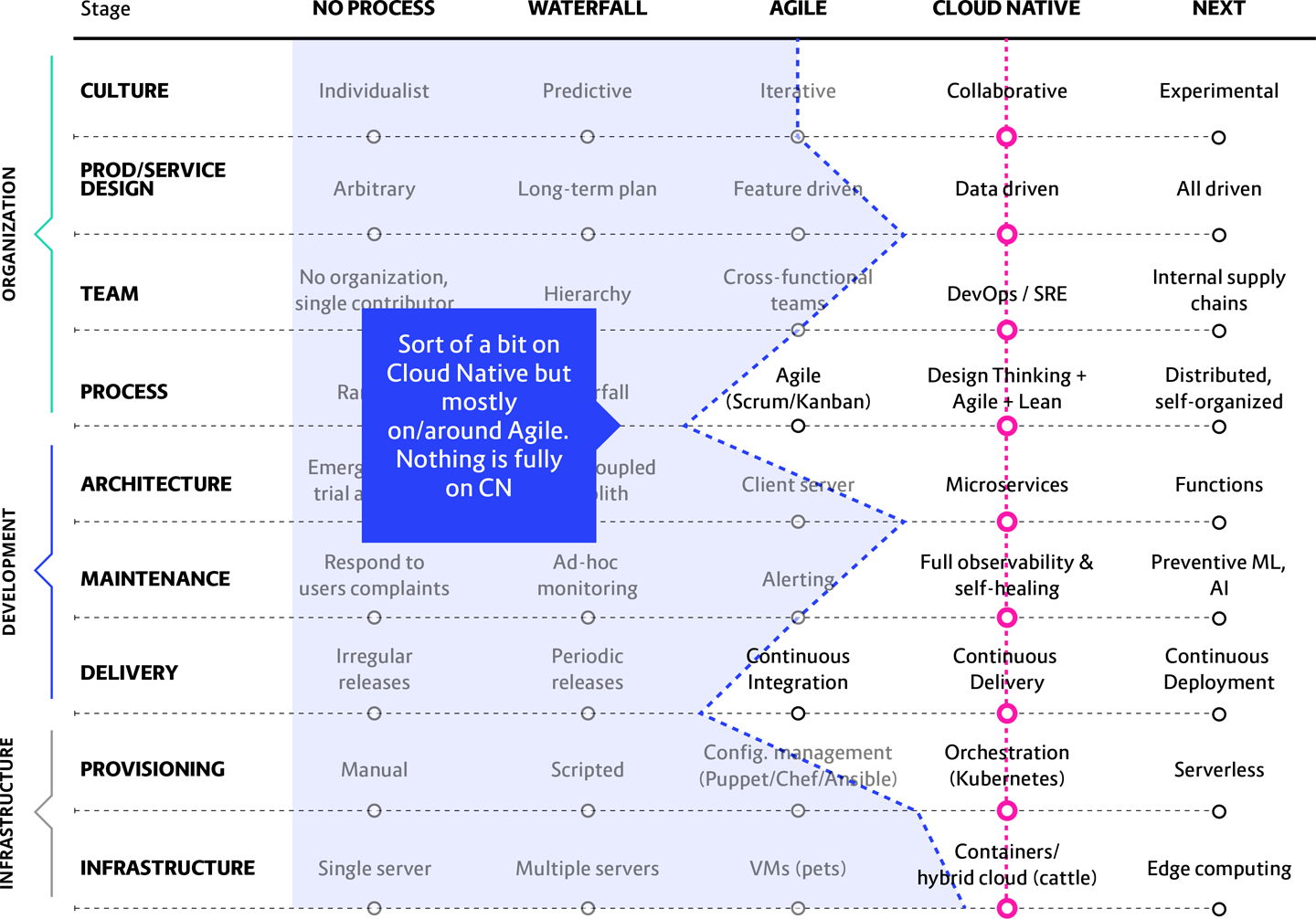
There are multiple problems with this “all or nothing” approach. Your legacy codebase does not automatically need to be abandoned. Yes, maybe it’s slow, changes are slow, the code itself is slow, and it definitely needs to be replaced someday—but so long as it is in place and working, *you should not touch it*.

This was exactly the mistake that WealthGrid made in their second attempt to go cloud native: setting aside the old system, essentially leaving it on life support with a skeleton crew keeping it alive while everyone else scrambled to build the new cloud native platform.

However, as WealthGrid discovered, in order to shut down the old system, the new one must first be working perfectly while re-creating 100% of the functionality provided by the existing one—all before you can ever begin rolling out new features. We have over and over seen real-world companies that, like WealthGrid, attempt to re-architect their monolith into 20 microservices and attempt to deliver them in a completely new greenfield system. Of course they built no connections to the old monolith—that is about to be abandoned. Of course they invest no effort, over the year or longer that this new platform takes to be built, to extend or upgrade the old one.

What they eventually realize is that this means they must build all 20 pieces and have them fully cross-functional, talking to each other and working together perfectly, before they can ever put the new system into production. This will rarely happen.

[Figure 13-9](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch13.html#Note_how_the_initiative_apostrophe_s) shows how an “all or nothing” approach can lead to a disorganized and uneven migration attempt.



**Figure 13-9. Note how the initiative’s progress line is all over the place, indicating an uncoordinated transformation with multiple bottlenecks.**

An “all or nothing” attempt—rebuilding a monolith into microservices from scratch—is not categorically a bad move. Greenfield can be a great way to go but only *if* you know what you are doing. Most companies don’t. Cloud native is simply too new.

This is actually a nice illustration of the Pareto principle, which states that the first 80 percent of a project is pretty simple and will take 20 percent of the total time to achieve. Unfortunately, the last 20 percent is the most difficult and complicated, so finishing takes the other 80 percent of the time. In our scenario what the Pareto principle is saying is that the delivery of microservices to the new platform will be delayed for a very long time.

**In this scenario we see:** Companies with a large, venerable codebase that decide it’s time to scrap that and just build a brand new cloud native system. After six months to a year of dedicated all-or-nothing effort, all they have is a stalled attempt to build a completely new system. None of it is ready to run anything. Meanwhile there has been no development at all on the old system.

**What will happen:** Confusion, chaos, and a deeply frustrated and unhappy team. You can’t just go all in, assign all your COBOL engineers to shift to Kubernetes, and expect it to work out. You may as well ask them to fly your mainframe to the moon. It’ll be just as effective.

**A better approach:** You did read the transformation design in [Chapter 11](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch11.html#applying_the_patterns) and [Chapter 12](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch12.html#applying_the_patterns_a_cloud_native), didn’t you?

**Practical patterns solution:** First of all, if any part of your existing legacy system is stable and requires few or no changes, just keep it. At least a first, anyway. In many cases it isn’t worth moving to microservices or cloud native in general as the value might be very low and the cost of change significant. A prime example of “if it ain’t broke, don’t fix it.” We even have a pattern for this: *Lift and Shift at the End.*

Meanwhile, you can follow the transformation design we outlined in [Chapter 8](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#patterns_for_organization_and_culture), starting with a small dedicated *Core Team* in charge of building your new cloud native platform while the rest of the engineers continue delivering on the existing system until the new one is production ready.

**IF IT AIN’T BROKE...BUILD A BRIDGE**

An important note: if you do have a deep legacy system that does what it needs to do, rarely needs fixing or changing, and is completely stable—don’t try to rebuild it. There could be five to six million lines of code in the old system, produced over years if not decades of work. You can’t rebuild that. You probably shouldn’t even try. It’s expensive and unnecessary to rebuild from scratch an entire monolith that is actually working pretty well for you, just because you bought into the greenfield myth. Rather than starting over, why not keep the legacy codebase in place and build a custom bridge to the cloud? It’s cheaper, it’s faster, and it’s much, much easier.

The goal of a migration is to increase the speed and frequency of delivering value to the customers—*not* to have all the code in microservices. Therefore you want to focus on the parts of the system that are changing the most and where most of the engineering effort will be focused. This gets that boost in productivity without a massive, and not very necessary, rewrite of any stable and functional legacy parts of the original codebase.

So, package that piece (or pieces) so that it sits on one server and works forever. After that, you can design and then create brand new segments that talk to the old code. Build a tiny greenfield platform and then slowly move one piece over at a time, refactoring and rebuilding your platform as you go. It’s a very slow, iterative long-term process, not a lift and shift of your whole legacy system to the cloud. But it will get you there.

**Lack of Platform Team**

It is extremely common, especially in larger companies, to find that before an “official” cloud native initiative is ever launched multiple teams have gone off on their own to experiment. This happens when they come back from tech conferences all excited about trying out cloud native tools and approaches. The company IT team has nothing to offer, so they go off to the side and start trying things on their own. This is such a common phenomenon that it even has its own name: shadow IT. This is all too easy because all you need is a credit card to be up and running on AWS or Google Cloud Services or Microsoft Azure in a basic way. At this stage, though, this is a good thing to happen. Teams are learning by doing, and the cost of experimentation is greatly reduced because they are not waiting for permission.

Eventually, though, if no organized company-wide initiative takes over, you end up with too much of a good thing. Seven different teams have each proudly produced their own version of a cloud native platform—but you can’t maintain seven different platforms. It is too expensive and makes no sense. The teams will still struggle to make it work, but thanks to a hierarchical organizational structure and the fact that each team has its own specialty, they don’t really know each other. They have no process for working together, or even talking with each other, for trying to figure out an effective solution (not that one exists). Often they are completely unaware that anyone else in the company was doing the same thing they were. All of that means that the organization has no way to capture or learn from the experience these separate teams gained.

This unfortunately is a pretty difficult problem to solve. You need to find a balance between flexibility and standardization. Strong standardization is great because you limit the support and maintenance costs, but this dramatically limits experimentation and research, which are crucial aspects of cloud native. Unfortunately, going fully in the direction of flexibility means you can’t support all the varied and random things everyone comes up with. So a good organization finds a way to standardize to a certain extent while still enabling experimentation and research into new things.

**In this scenario we see:** A company where multiple DevOps teams have separate, non-standardized setups, possibly on the same public cloud or (worse) multiple clouds. So now there are, say, seven different systems, all built and deployed in different ways, using different tools. Everything is random. Not much harm done—yet.

**What will happen:** If any of these independent projects goes to production with their custom setup, which was created without appropriate knowledge, and without involving other stakeholders, it will be difficult to refactor it to a consistent setup with proper consideration of operational, security, compliance, usability, and other concerns.

If that is difficult with just one rogue platform, consider trying to reconcile seven different versions to run on a single system. It is basically impossible to refactor these to all work on a standardized platform for production; such a platform simply does not exist. And of course each one of these teams believes their approach is the best, the one the company should adopt.

**A better approach:** In properly implemented cloud native architecture, the platform team is permanent. This is because its work does not end with the successful creation of the initial platform—that was simply its first task. Once the major parts of the transformation are up and running, the team continues to work on improvements to refine and optimize the platform. Only a dedicated and constantly improving team can consistently dive deeper into technical challenges.

**The practical solution:** Assign an official *Platform Team* of five to eight engineers and architects. This team is dedicated to building the platform and are given sufficient resources to research, experiment, and implement the technology. Meanwhile, *Internal Evangelism* alerts the rest of the organization that a unified platform is being developed. Any teams who have built their own skunkworks platform on the side are invited to share their experience and contribute to the new platform if possible through applying the *Learning Organization* and *Communicate Through Tribes* patterns.

**Lack of Education and Onboarding Plan**

Sometimes an organization starts out the right way with their cloud migration: they name a small transformation team to learn cloud native by experimenting and building PoCs and eventually an MVP. This *Core Team* then uses their new skills to expand the platform, get it production ready, and evangelize the transformation to the rest of the company.

Everyone is excited and ready to launch. The only problem being there is no plan for educating and onboarding the other 300 developers who are about to be working in the shiny new cloud native system. Many of them were unaware that it was even on the way!

This sounds implausibly short-sighted, but we actually see this happen shockingly often. So much focus is placed upon getting the new system designed and built that no thought is given to how it will actually be put into use once it’s in place and ready. Many times this happens because the transformation is led by engineers with a narrow focus on their own work and simply do not think in terms of the strategic company vision (which is exactly why the Dedicated Strategist pattern arose).

Unfortunately, the fix for this is not an easy matter of, “Here’s a wiki and some docs; see you in Kubernetes on Monday!” Preparing developers for working in a cloud native style in your new cloud native system carries a steep learning curve. Getting them properly onboarded requires dedicated training and proactive education that begins long before the platform is even fully built.

Lack of adequate education leads, as a start, to inefficient use of the platform and the tools, among other problems. (Another problem: unhappy developers feeling blindsided by this huge change.) Failing to adequately prepare and educate the developers who will be using the new system will dramatically reduce the overall value produced by doing the transformation in the first place.

**In this scenario we see:** Realizing a week before going into actual production that there are 300 engineers who don’t know how to do the work. That is when managers start frantically Googling “cloud native team training.” Consultants *looooove* to get exactly this type of desperate phone call.

**What will happen:** Either the go-live date gets pushed back to create a training window or the project proceeds with undereducated engineers who don’t know the best way to operate within the new system. Unfortunately this second outcome is what usually happens, because delaying the release date will harm the platform owner’s reputation.

Many mistakes get made as everyone scrambles to adapt to a steep learning curve for the new tech, the new processes, and the new platform all at the same time. They will make decisions based on their current knowledge and experience, because without good education or onboarding that is mainly what they’ve got to go on. This not only reduces the value the company will get from this transformation but eventually will start re-introducing monoliths, manual testing, delayed deliveries, etc.

**A better approach:** Prioritize a training plan as part of the migration strategy from the beginning. Gradual onboarding of teams means earlier adopters can help train later teams. Timing is important; obviously, waiting until the last minute to train can be disastrous. But training too early is also bad. It’s demotivating to get educated and then have nowhere to apply your new knowledge for a long time. Newly learned skills fade quickly if not put into practice.

It is important to say here that education is not just for onboarding, but needs to continue long after the platform and the apps are in production. Education is an ongoing and never-ending process that, once you start it, requires intentional support, organization, and planning to make sure your team’s skills stay current.

**The practical solution:** WealthGrid’s transformation design in [Chapter 8](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#patterns_for_organization_and_culture) shows how, optimally, training begins during the middle stages of a migration. First, the *Platform Team* gets the training necessary to begin the initial design and build. It takes four to six months to build a platform and get it fully operational. This is the point to begin *Gradual Onboarding* to train the first few teams onto the platform. This is not just functional “how to” training, but full onboarding with architectural principles and culture, including examples and tutorials in the *Developer Starter Pack*. Once this first wave of teams is fully grounded they can help the next teams to come. *Ongoing Education* ensures that teams keep their skills refreshed and current even once they’ve been successfully crossed over the new system. And all the while the *Dedicated Strategist* helps keep everything on track.

**Summary**

You may have noticed a lot of repetition in the patterns presented for solving each of these commonly occurring problems. Here is another Leo Tolstoy quote to help explain this phenomenon:

*“All happy families are alike; each unhappy family is unhappy in its own way.”*

Tolstoy was talking about how unhappy families seem to take the same basic set of existential building blocks everyone gets, only they assemble these in a skewed way to create their own particular family fortress of misery and dysfunction. Meanwhile, happy families are all kinds of uniform in their contentment.

It works the same way with cloud native transformations. We witness the nine common challenges we describe in this chapter happen all the time, even as part of unique migration projects in very different kinds of companies. And yet when it comes to handling these challenges we keep showing you the same patterns again and again—*because they work*.

Fortunately, if you come to this book early enough in your own transformation journey, you will be equipped with the patterns (and context for when and how to apply them) to make sure your company’s migration story has a reasonably happy ending. [Chapter 8](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#patterns_for_organization_and_culture) is your main map, though this chapter is still worth your time since it’s always worthwhile to see examples of what not to do.

And if not, well, late is better than never. Forget the sunk cost and use everything you’ve learned to rethink your strategy. We can’t resist a second quote here, this time a Chinese proverb, because it captures this perfectly: “The best time to plant a tree is twenty years ago. The second best time is now.”

So: If you are here because you’re stuck in the middle of a migration gone awry, we hope that you recognize yourself in one, or even more, of these scenarios, and use the suggested patterns—repetitive as they may be—as a way to find your way back on course.