**Chapter 14. Building a Bank in a Year: Starling Bank Case Study**

*By Greg Hawkins*

*Greg Hawkins is an independent consultant on tech, fintech, cloud, and DevOps. He was CTO of Starling Bank, the UK mobile-only challenger bank from 2016–2018, during which the fintech start-up acquired a banking license and went from zero to smashing through the 100K download mark on both mobile platforms. He remains a senior advisor to Starling Bank today. Starling built full-stack banking systems from scratch, became the first UK current account available to the general public to be deployed entirely in the cloud. Voted the best bank in Britain in 2018—and again in 2019—in a survey of more than 27,000 consumers, Starling beat out High Street mainstay banks with hundreds of years of experience behind them a mere year after its launch.*

*As a startup Starling was able to do cloud native from the start, while most companies are looking to migrate existing operations to the cloud. It’s a complex and challenging process no matter where you’re starting from! We are including Starling’s story in part to illustrate what building a greenfield cloud native system looks like, but mainly how the same patterns can be used to compose many different kinds of transformation designs.*

*Greg Hawkins oversaw the creation of the cloud native platform as Starling’s CTO. He remains involved as a senior advisor, helping the bank grow and shape its cutting-edge services as it continues to disrupt the financial services industry. This is the story of how Greg and the Starling tech team built a cloud native bank in a year.*

So this is the tale of how we took advantage of our small scale to build a bank in a year, completely from scratch, completely cloud native. OK, so to be fair, before that year of intense building we did have a couple of prior years to lay the groundwork, mainly preparing the application for a banking license and getting the business side in place. There had also been a couple of technological false starts, meaning that when I joined Starling to begin building the tech stack in the beginning of 2016, we already had an idea of what didn’t work for us, which was helpful. And of course we didn’t stop after a year—many of the features that make Starling great today came later; the work is never done. But from a standing start we had a real live UK current account with payments and cards in place in a year, and we built it cloud native.

It was a bit nerve-wracking that, as we got started, we were having lots of meetings with regulators about how and why a bank should be allowed to deploy entirely in the cloud—it was far from certain that it would be allowed at all. We were right in there at the dawn of cloud banking in the UK, building all the software from scratch, before we knew it was even legal to operate as a cloud bank at all.

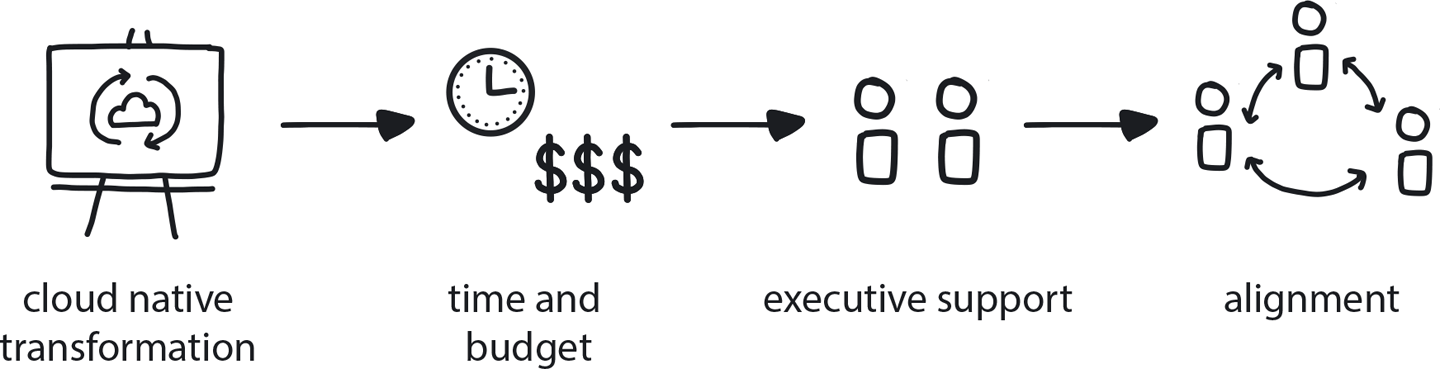
To a degree, the tight build period was actually imposed on us by regulation. In the UK once you have been granted your provisional banking license (or banking license with restriction, as it’s called) you have precisely one year to sign up your first customer, or else lose your license and have to start all over. So there was that driver behind our timeline, as well as competitive disadvantages that would arise from delay.

Even though cloud banking was in the process of being invented as we (and also a few other challenger banks starting up around the same time) went along, building for the cloud was already fairly well-understood. Primitives like auto-scaling groups were well known, and the associated architectural patterns were well-established by companies like Netflix. Kubernetes and Terraform were pretty new at the time. After some bad experiences we judged them too immature for our purposes back then, and so we built directly using a handful of Infrastructure-as-a-Service primitives—building blocks that Amazon Web Services (AWS) rests many of its other services upon. These were services like Elastic Compute Cloud (EC2) and CloudFormation that allowed us to build a collection of related AWS resources and provision them in an orderly and predictable fashion.

Fortunately, we were a well-funded startup, so we were able to make sure our devs had everything they needed instead of trying to keep costs down and save on the AWS bill. We kept an eye on costs but didn’t manage them too closely. The first decision was architecture, with a goal of building in such a way that we would be able to change quickly and remain resilient as the startup evolved.

*In Starling’s case, cloud native was a core part of their business plan. But in any organization it is essential to have Executive Commitment for the project to be successful, and Dynamic Strategy for adjusting plans and tactics in response to changing business conditions.*

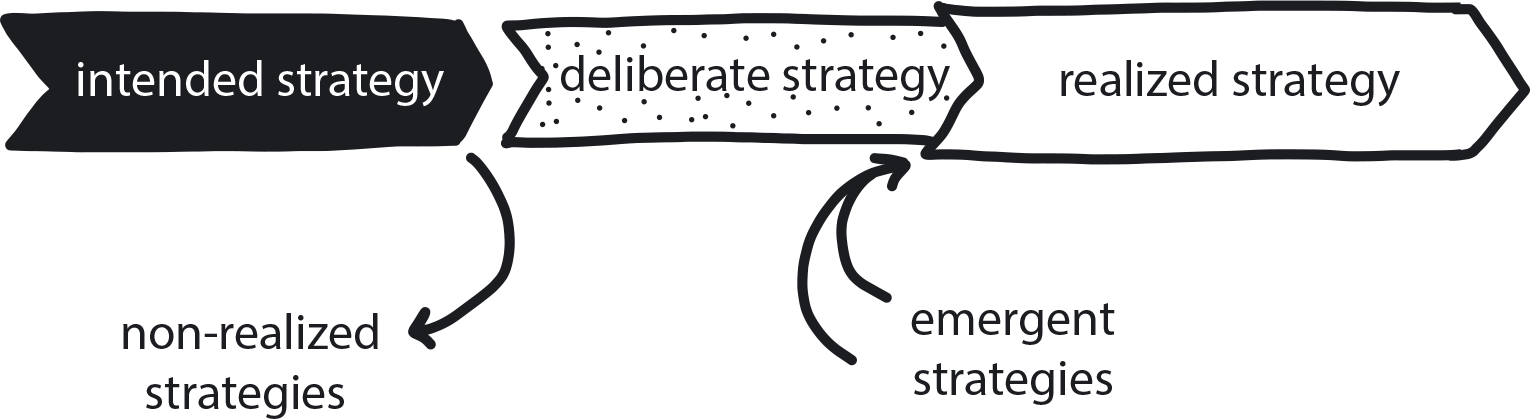
**PATTERN: EXECUTIVE COMMITMENT**



**Figure 14-1. Executive Commitment**

To ensure allocation of sufficient resources and reasonable delivery time frames, large-scale projects such as cloud native transformation require strong Executive Commitment.

**PATTERN: DYNAMIC STRATEGY**



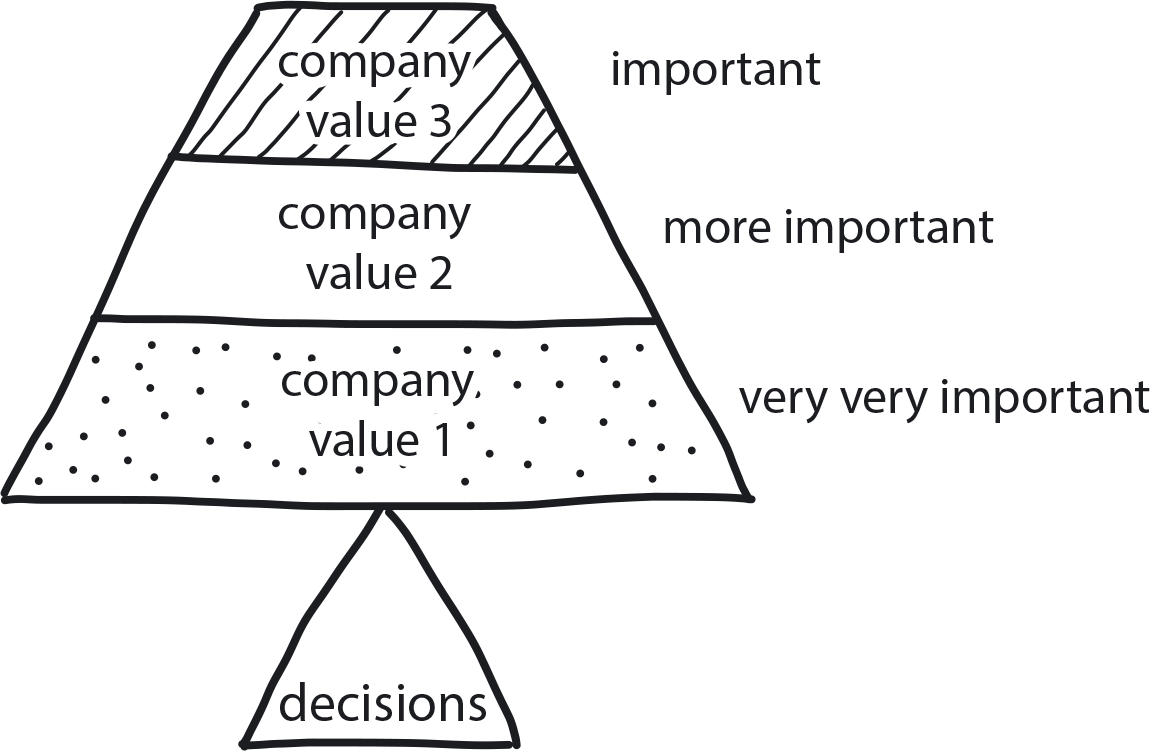
**Figure 14-2. Dynamic Strategy**

Today’s tech-driven marketplace is a constantly shifting environment, no matter what business you are in—so your game plan needs to shift right along with it.

**Acing the Architecture**

A key value hierarchy that guided our choices was security over resilience over scale. In reality, these values entangle and overlap, but having a clear hierarchy allowed us to make concrete decisions about how to proceed. Without some recognition of what’s important to you it’s hard to make the decisions that allow you to move forward in a complex and fast-changing environment. Then a secondary value hierarchy of velocity over economy further guided the decisions we made. We needed an approach around not just software architecture but also the organization itself that prioritized pace. And that is why in various areas we deliberately tempered our microservice architecture with some pretty monolithic practices.

**PATTERN: VALUE HIERARCHY**



**Figure 14-3. Value Hierarchy**

When an organization’s values are clearly stated and prioritized, day-to-day decisions can be made without seeking consent or permission/approval.

A monolith lets you go very quickly to start with, because you don’t have to manage all the interrelationships between all the different microservices and all the common things that they need to share. The problem is that monoliths slow down over time—a lot. Everyone’s all working on the same thing and stepping on each other’s toes; it becomes harder and harder to change anything in a tightly coupled system that has a lot of stakeholders. And on top of that it’s frighteningly easy for small errors to have enormous implications.

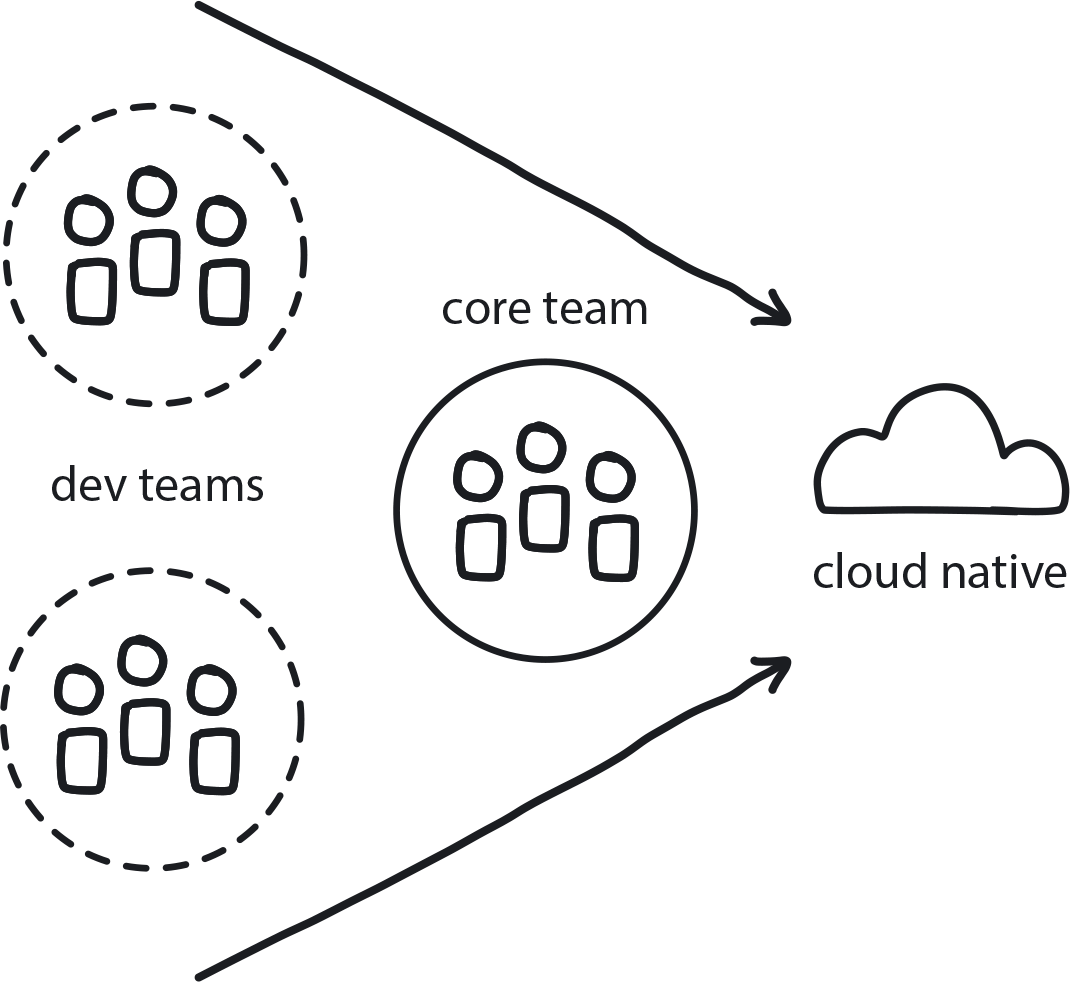
By contrast, microservices are very slow at the start because of all the complexity, but then you do get faster as you get more proficient at managing things. And you don’t slow down so much from the effect of everyone stepping on each other’s toes in the same codebase. So my plan, my big idea—though it’s probably everyone in the world’s big idea—was to try and get the best of both worlds. To start fast and stay fast.

Our approach to capture the benefits of both was to follow a sort of microservices, or maybe miniservices, architecture but to behave monolithically in many ways. For instance, even though we have something like 30 microservices, we tend to release them all at the same time rather than having separate release cycles for all of them. That will change over time, but we’re still working this way almost three years later, to be honest and just looking to kind of break it, round about now actually, when we’ve got 60 to 70 developers.

This might just be the inflection point where a microservices-style organizational structure becomes easier, more sustainable. And this makes sense because microservices solve the problems of large organizations quite elegantly. But when you are a startup, if you were to introduce all the necessary overhead of that when you have a team of two to five developers, then you just hurt yourself with no benefit. And that is just where we were, at the time: we started out with just two engineers. The team grew, of course—by the time we went live we were up to 20 or so—but during those earliest days we were in the low single digits.

**PATTERN: CORE TEAM**

This is an absolutely perfect illustration of the Core Team pattern in action: never doubt that you can build a cloud-native platform with a small team of five to eight engineers—Starling did!



**Figure 14-4. Core Team**

Dedicate a team of engineers and architects to the task of uncovering the best transformation path and implementing it along the way. This reduces risk embedded in the transformation while the team gains experience helpful for onboarding the remaining teams later.

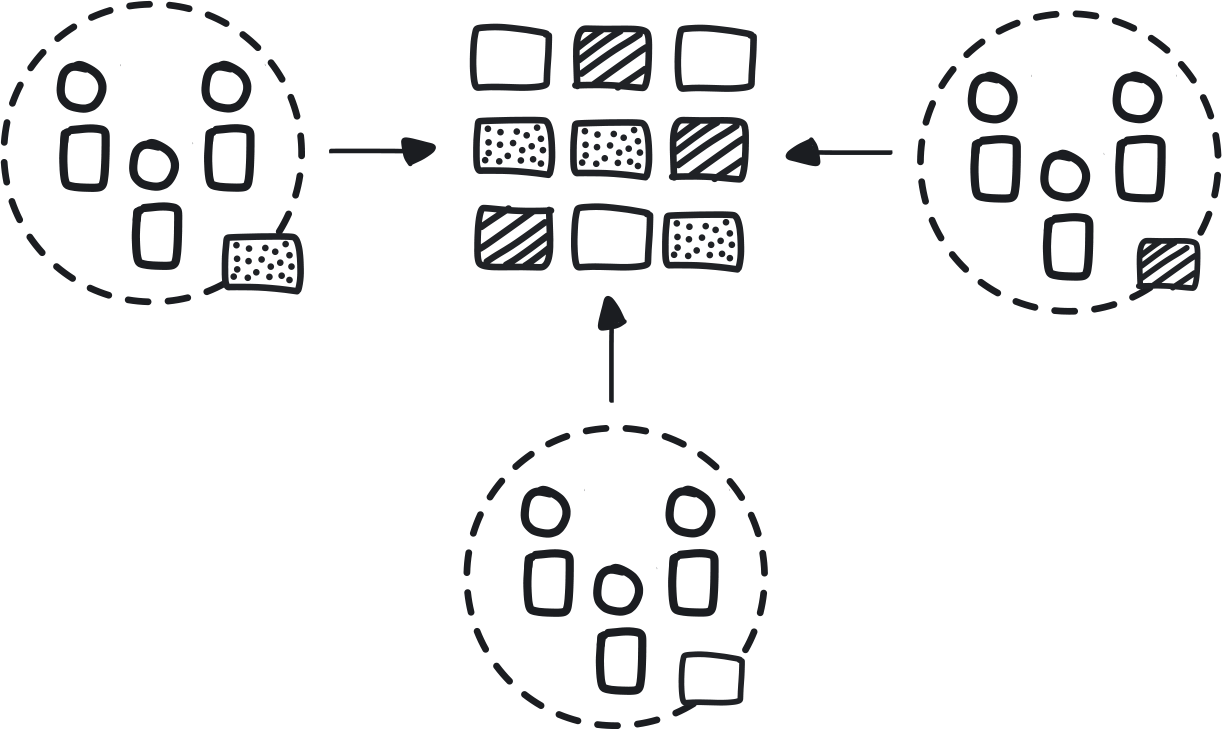
Although we were too small to organize ourselves along the same lines as a distributed microservice architecture, the idea was still to try and build in the ability to eventually make the organization more, well, microservice-y. So we architected for change. We decided we would build separate services that could be deployed independently and managed independently, but then for convenience, at least at first, we would behave monolithically—we would release them all together. That way you get the simplicity of a monolith in early days and the ability to move to independent management of microservices as the organization grows or as circumstances change or as the structure of the organization becomes clearer over time. The risk, of course, is that unexercised muscles atrophy, and so if you’re not actually releasing these things independently, how do you really know anymore that you could? These sorts of trade-offs are the bread and butter of delivering big projects like Starling. While this approach helped us to move fast at first, my definite concern was to stave off the sort of stagnation, or even eventual paralysis, that results if you go on long enough delivering a monolith.

In cloud native, microservices helps you stave off that paralysis by allowing different parts of your organization to act independently without being bottlenecked by others. It’s also fosters resilience: if all your services can tolerate the failure of all the others—if they can all live or die independently—then you’ve by default built in a lot of fault tolerance and natural resilience.

Microservices also give you tremendous flexibility to be compliant with Conway’s law, which states that software structure will come to reflect the organizational structure of the company building it. It’s easy to adjust service ownership to ensure that the system architecture is consistent with and reflects the structure of your organization. Unfortunately for a small startup growing at explosive rates, the structure of your organization is only half-formed, and probably extremely volatile. I often say that Conway’s law is a bit of a mess with startups. So devoting energy to complying with Conway’s law is best deferred by maintaining as much flexibility as possible and worrying about it later as the organization’s mature structure emerges.

So much of my decision-making was guided by trading off flexibility and simplicity against each other.

**MICROSERVICES ARCHITECTURE**

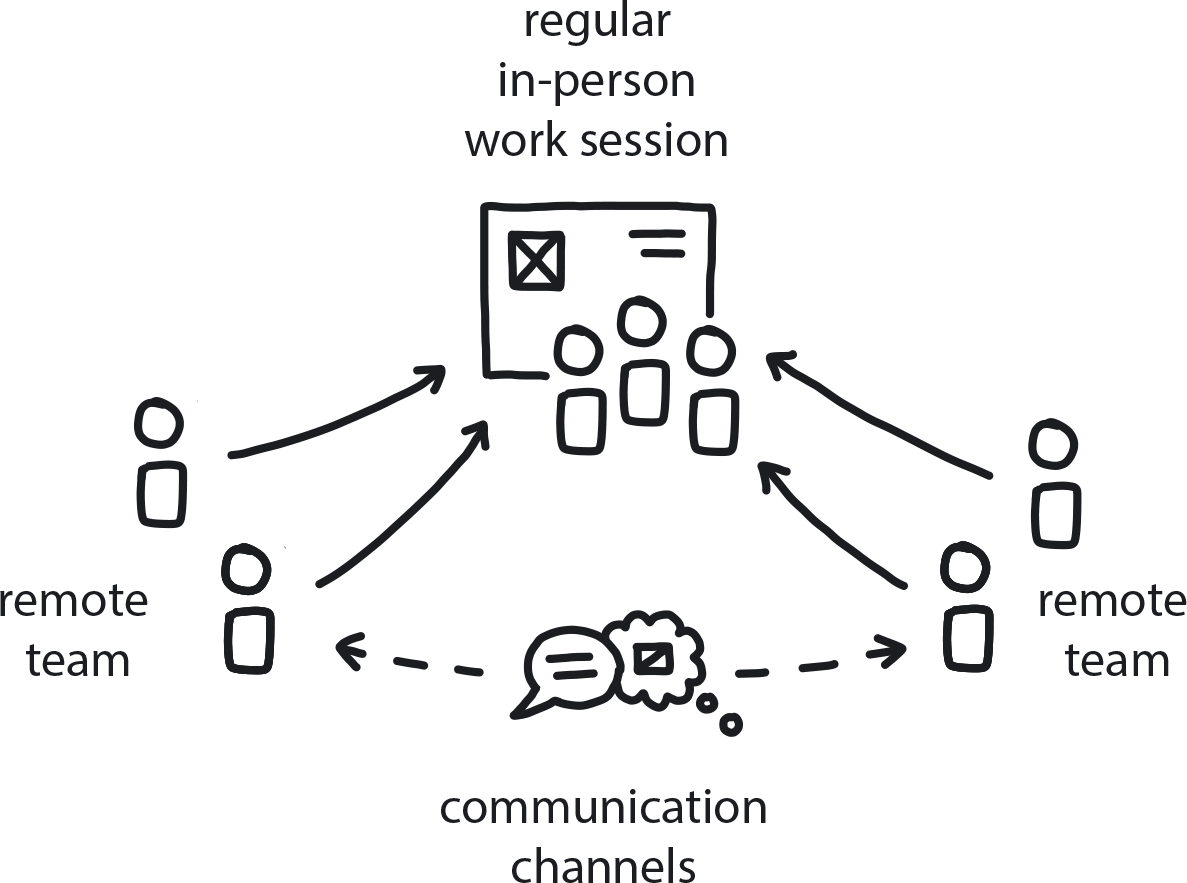


**Figure 14-5. Microservices Architecture**

To reduce the costs of coordination between teams delivering large monolithic applications, build the software as a suite of modular services that are built, deployed, and operated independently.

These were the principles I followed while architecting for change—for that point at which different teams are going to take ownership or responsibility for different components, going from the point where you have fairly few developers, they’re kind of all able to work on everything, and no one’s specialized around a single service because you need that flexibility. As the company becomes larger and split across different floors or different locations, etc., then you’re in a different world. The goal is that you’ve architected such that it becomes a case of just taking ownership rather than re-architecting the software to support the communication structure of your new organization.

**REMOTE TEAMS**



**Figure 14-6. Remote Teams**

If teams must be distributed, whether across an office building, a city, or the globe, build in regular in-person retreats/work sessions as well as robust channels for close and freely flowing communication.

How we built is the important thing. *What* we built, our tech stack, is rather beside the point, because things are changing so much and so fast and every company is different. The whole idea with cloud native truly is how you build, not with what, and so the architecture is what truly matters. But for those with a technical bent, we built Java services in Docker images, which we deployed 1:1 onto EC2 instances running CoreOS, along with sidecars for log shipping and metrics publishing. We used many cloud services like GitHub, [Quay.io](http://quay.io/), Artifactory, and Slack. We used Circle and then TeamCity for CI/CD. Our infrastructure code was CloudFormation and our rolling deployment tooling was our own.

**Building the Apps**

Once the initial platform architecture was decided, it was time to figure out what exactly we would run on it. As a mobile-only bank, this was our sole customer interface, and quite critical. We chose to have two mobile apps, one for the iPhone and one for Android.

We made no attempts to share code, for two reasons: one, we inevitably need quite deep native access to the hardware, and, secondly, we believe the best people to be writing for iPhones are those who eat, drink, sleep iPhones, love iPhones, and write in Swift for iPhones. Similarly with Android. Various technologies try to target both platforms, none of which super hot groups of engineers really love. So we’ve got two separate codebases there and we think it was the right decision.

That determined, we built our bank according to five principles I felt were important for cloud-native optimization.

**No IT Department**

At Starling, we wanted to be the first bank with no IT department. A lot of the people who came together to join Starling had a lot of experience of how banking traditionally delivers software, which is very badly. The main problem is the business/IT distinction: typically, you’ll get businesspeople writing specs; these get thrown over the wall to IT, who tries to translate them and deliver, and then throws it back over the wall to business, who will do some sort of acceptance testing, find they didn’t get what they want, have a bit of a moan, and the whole cycle goes round again.

We see all sorts of problems with this, not least efficiency, and we didn’t want to build our organization that way. We think of ourselves as a tech company with a banking license, not a bank that needs to try to do technology.

Some of the smarter incumbent banks have tried to fix this, but not very well. They see an Agile transformation as simply an IT initiative, which is merely tolerated by the business. So you might reorganize teams so they aren’t strictly specialized—instead of layers like a Unix team and a firewall team, you might have product- or feature-centric teams—but ultimately they’re still not with the core business. Communication with the business still has to be mediated by people who used to be called business analysts, who now might be called product owners. This still has many of the same problems as before.

Cloud native represents a genuine paradigm shift in modern software development and delivery. Previously most companies understood that, in order to remake themselves as a 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 and the way they talk to each other. Unfortunately, most don’t extend this same understanding to moving from Agile to cloud native practices.

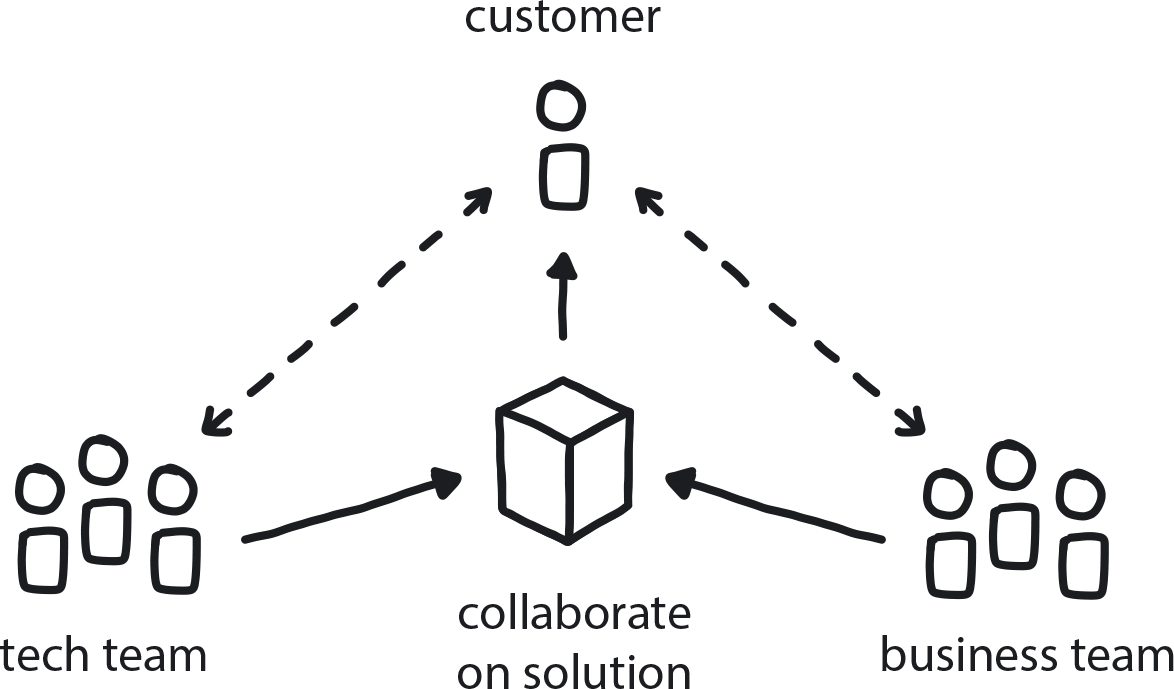
This particular scenario can happen any time in any organization, but most often arises when a company that has adopted Agile practices fails to recognize that moving to cloud native a second true paradigm shift. Instead, they believe cloud native to be “just a new way to do Agile”—seeing it as a technical, not pan-organizational, change. (See [“Treating Cloud Native as Simply an Extension of Agile”](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch13.html#treating_cloud_native_as_simply).)

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, it simply doesn’t work. And now Starling Bank confirms the same phenomenon, which they successfully avoided by observing what *didn’t* work when traditional banks tried to go cloud native.

So we went a completely different way.

We structured Starling so that our delivery teams, the people actually delivering new features and parts of the bank, were largely engineering and engineering-led but also contained members from what you’d normally regard as “the business” end of the company. So our cross-functional teams were truly cross functional. We would have infrastructure expertise, Java expertise, iPhone expertise, Android expertise, possibly UX, UI, and product expertise as well, and then non-tech people as well, to keep us grounded in building for customer needs. At one point, for instance, one of our delivery teams had the CFO on it!

**PATTERN: INVOLVE THE BUSINESS**



**Figure 14-7. Involve the Business**

The business teams and the tech teams need to collaborate to create an effective customer-feedback loop that drives product improvement.

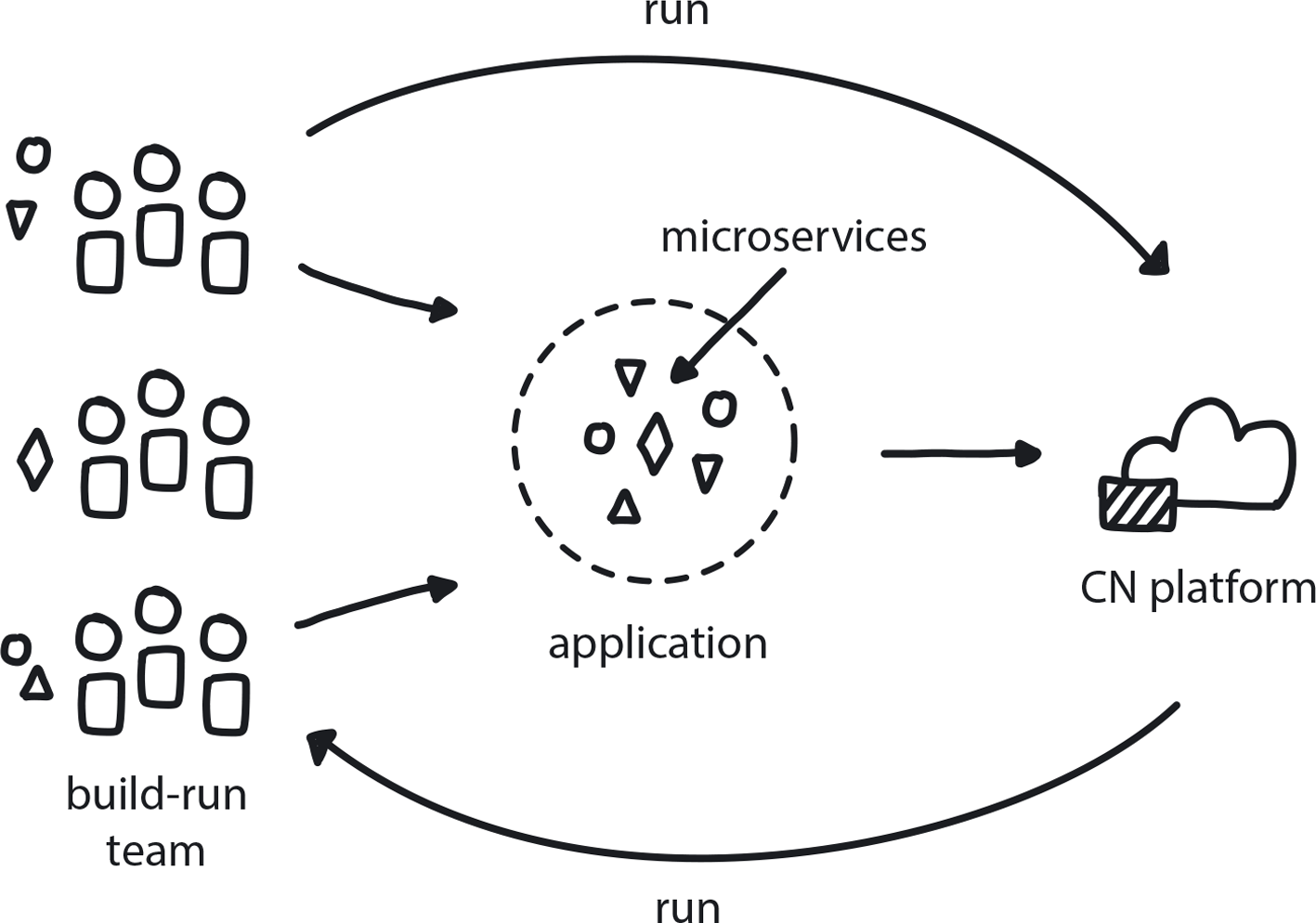
Not only are those mixed teams working together on the same goals, many things that they are doing come directly from those people in the business. If someone in payment operations is having difficulty administering some direct debits or there’s been a problem with a direct debit run the night before, there’s no one at any sort of high level making a decision whether that needs fixing. It gets fixed because the engineer’s sitting right next to the person who’s actually seeing the problem and just takes care of it. Ultimately, of course, the big priorities are set by the executive committee. But there’s quite a lot of latitude for the engineers in the delivery teams to set their own priorities and to manage their own backlogs.

Thus the distance between business and IT is dissolved to a very high degree. As far as possible, we try to have teams that focus on products, not projects—nothing that will be important one year and not the next. That way we don’t end up with all sorts of sins bequeathed unto the next generation. And this naturally means that teams are doing a lot of stuff, because not only are they delivering new things, but they’re also responsible for running all the old things.

**You Build It, You Run It**

Which leads to our next guiding principle: You build it, you run it, the true meaning of DevOps, but even more than that. Being an engineer at Starling isn’t easy: you conceive and you design it, you build it, you run it, you support it, you fix it, you enhance it, you work out how it’s being used, how it’s being abused, how to sunset it, how it’s going to be reported on, how the auditors are going to approach it—every aspect of it, because you are truly accountable and an owner of the system that you’re working on.

**PATTERN: BUILD-RUN TEAMS (“CLOUD NATIVE DEVOPS”)**



**Figure 14-8. Build-Run Teams**

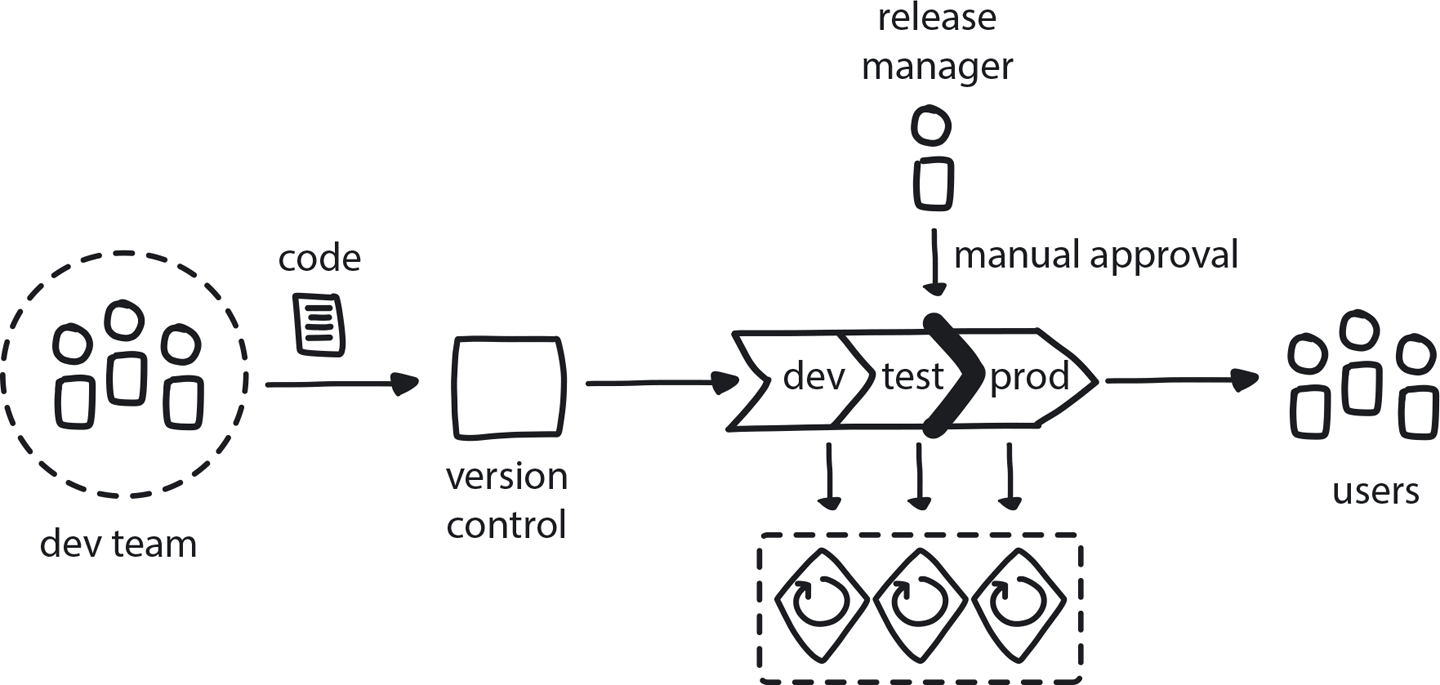
Dev teams have full authority over the services they build, not only creating but also deploying and supporting them.

The real point here is accountability: our engineers are on the front line of incident response. In traditional incumbent banks, you normally have no view of what’s happening in production. You might deliver something and maybe two months later it will go into production, but you’ve got no idea whether it’s working—you never see any of the logs. In the pre-cloud-native approach, where you just send it off to the QA department, you have very little incentive to take it seriously to the degree you’re going to deliver quality, supportable code. Whereas in the cloud-native way, you’re the one watching the metrics, you’re watching the logs, you’re going to make very sure it’s supportable—that you’re not going to be woken up at 4 a.m. by something crazy going on. So the level of accountability we require from our engineers is, I think, a key part of how we deliver so well at speed.

**Continuous Delivery**

I think CD is probably the most important of the principles that we built Starling Bank around, but it might be the least important to talk about, because I think it’s comparatively well-understood. But it’s worth pointing out that we release pretty fast. Not as fast as Amazon, but nonetheless, we release and deploy our entire backend between one and five times every day. Every day we’ll also make at least that many infrastructure changes as well, so AWS CloudFormation-type changes. The mobile apps we release every one to two weeks.

**PATTERN: CONTINUOUS DELIVERY**



**Figure 14-9. Continuous Delivery**

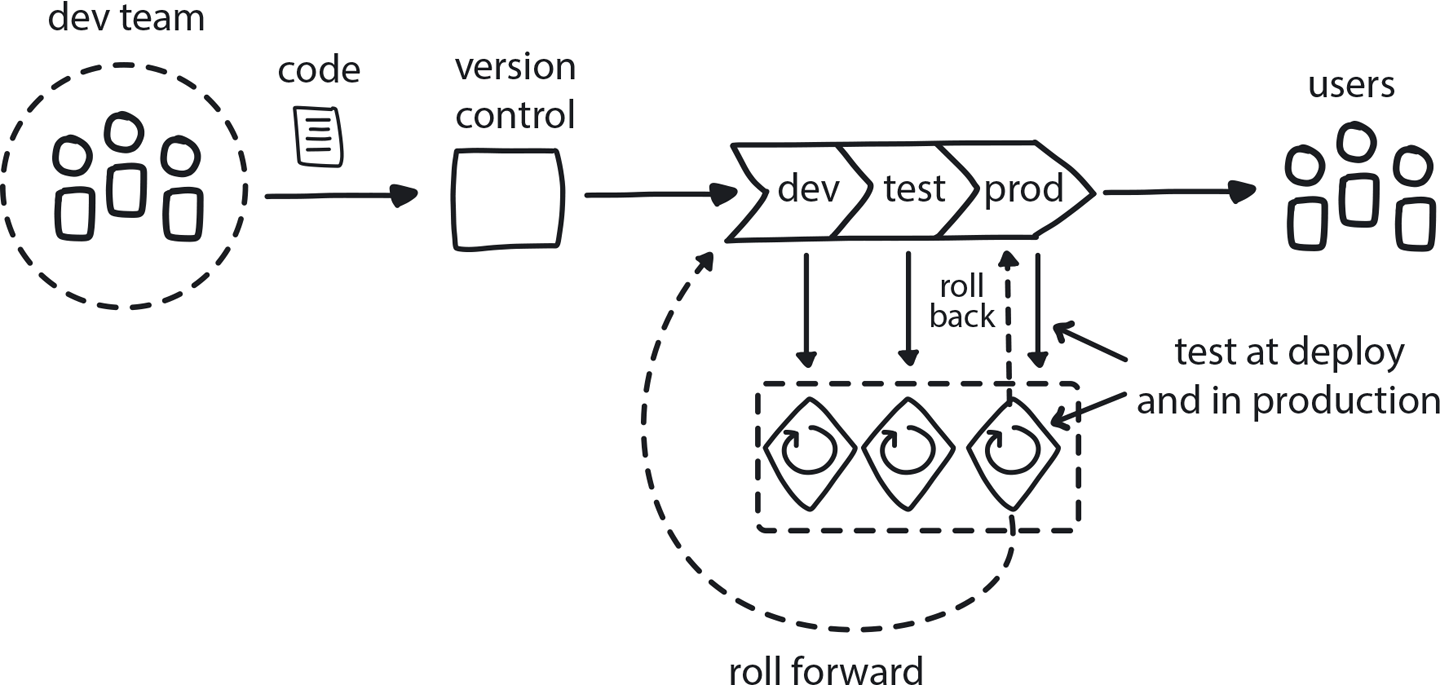
Keeping a small build cycle means code is always ready for production and features can be immediately released to customers—and their feedback quickly returned to developers.

More important than the actual rates, though, is this point: if a single day passes without a release, we’re already worried. This is a big red flag for us, for a lot of reasons. A day without a release is a day that we haven’t practiced one of the most critical functions in our business. Therefore it’s a day that we’re less sure that we can do it. It’s an extra day accumulating risk for the next release. All of our software processes are built around minimizing risk of release by keeping those releases small, minimizing risk of changes themselves by keeping changes small and incremental. If we can’t do a release, we’re building up more and more changes that are going to result in a riskier release, and we hate that.

A day without a release is also a lost opportunity. We frequently use releases to get extra diagnostics or things into production to gain insight into what’s happening in areas that we don’t understand. It’s one day potentially more vulnerable to out-of-date dependencies, because every time we do a build we use the latest versions of our Docker image dependencies and all that sort of stuff.

Finally, it’s a clear red flag that something somewhere is broken, because otherwise why aren’t we releasing? It suggests that we’ve got a paralyzed process somewhere and that we have a degraded incident-response capability. One of the great things about all this automation is that we can get to 4 a.m. and we see a bug and we don’t have to do a rollback. We can actually wake up a couple more developers, do a fix, and roll it out, and go through all our automated QA and be confident by 5 a.m. that what we’re doing is right.

**PATTERN: RISK-REDUCING DEPLOYMENT STRATEGIES**

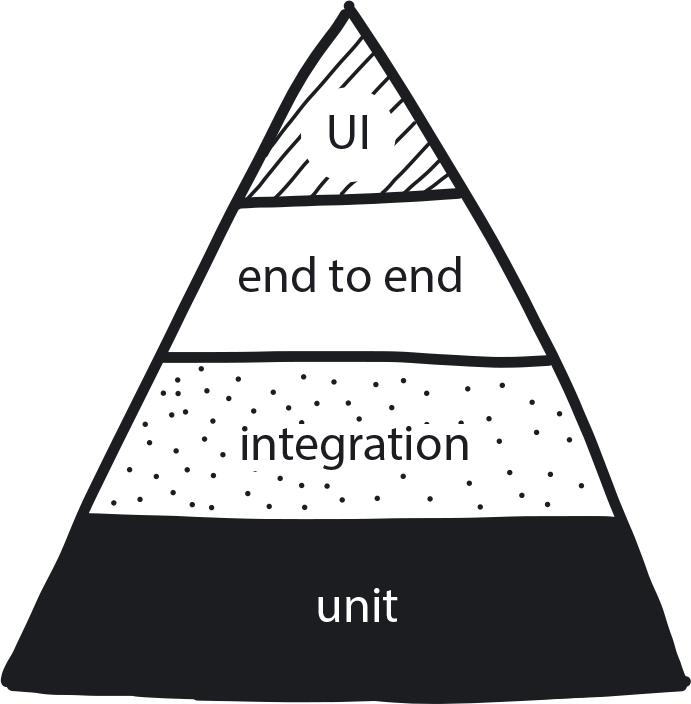


**Figure 14-10. Risk-reducing deployment strategies**

Employ release tactics to decrease the chance of problems happening when changes are introduced into the production system.

One interesting thing we do at Starling is the take-ownership ceremony we do in the engineering Slack channel when changes are going out. According to continuous delivery, this should be theoretically pointless, right? If all your tests have gone green and you’ve got signoffs, then you should be good to go, right? There shouldn’t be anything else. But we make this ceremony out of it anyway, because this is really about ownership. A way of publicly saying, “OK, we’ve gone through all this automation now, guys, and everything sounds good, but who’s really responsible for this? It’s me and it’s you, and it’s you. Are we all good?” And everyone involved saying, “Yes, we’re good.” And then the button gets pressed and, just for fun, the actual release is announced by animated gif on Slack. It can be anything on the theme of “rolling” and some are quite creative—parrots on roller skates, famous cartoon characters. You can sometimes tell just by a quick look at the channel who’s doing a release because people have their favorite images dancing around.

**PATTERN: AUTOMATED TESTING**

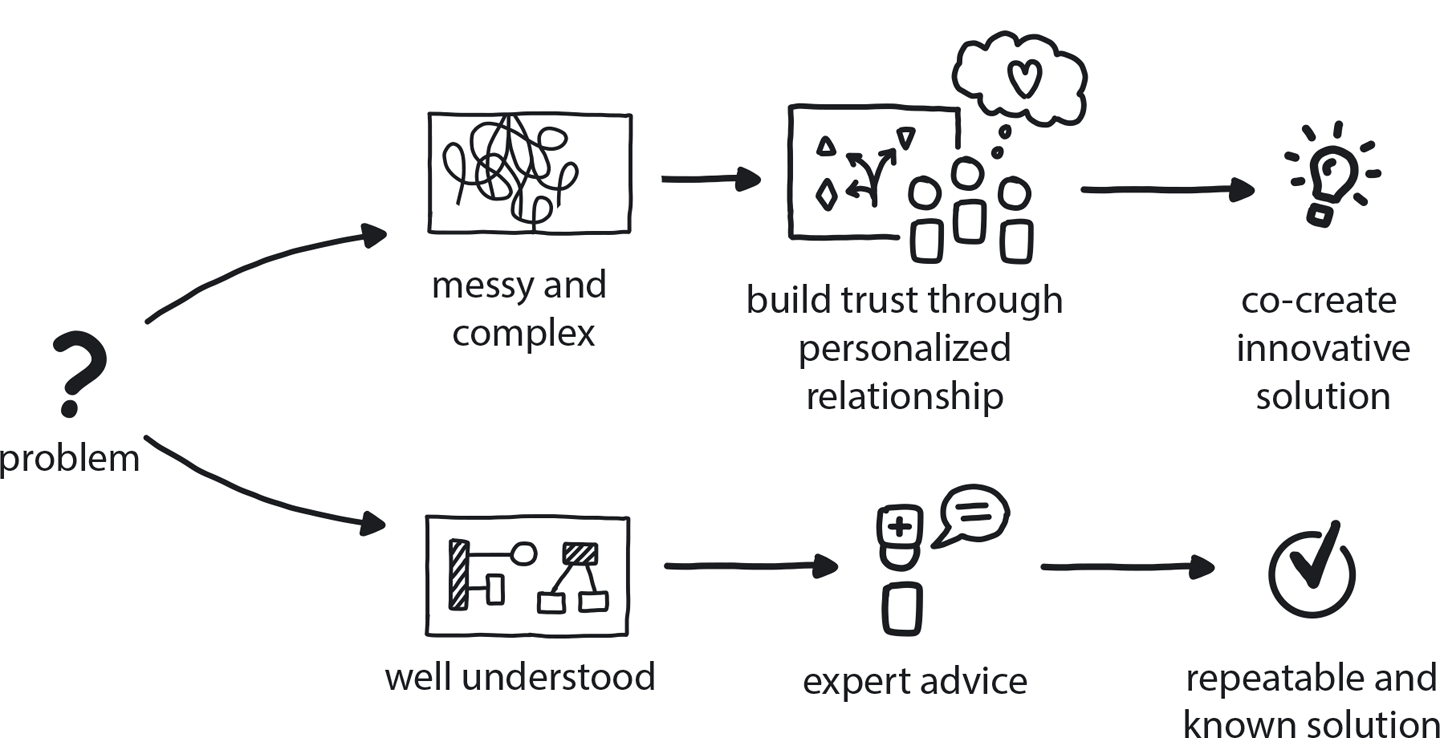


**Figure 14-11. Automated Testing**

Shift responsibility for testing from humans (manual) to an automated testing framework so developers can deliver faster and have more time to focus on improving features to meet customer needs.

Whimsy aside, what is really important about this ceremony is commitment. In an incumbent bank you would have no idea when your code is finally going out the door, so what sense of ownership can you possibly have? None. With this, your code goes into production and on the way home from work you’re going through a turnstile station in the Tube, seeing someone use their Starling card, and knowing that your code has just gone into that—it’s very motivational.

**PATTERN: PERSONALIZED RELATIONSHIPS FOR CO-CREATION**



**Figure 14-12. Personalized Relationships for Co-Creation**

Solutions to complex problems are best created collaboratively by teams with high levels of interpersonal connection.

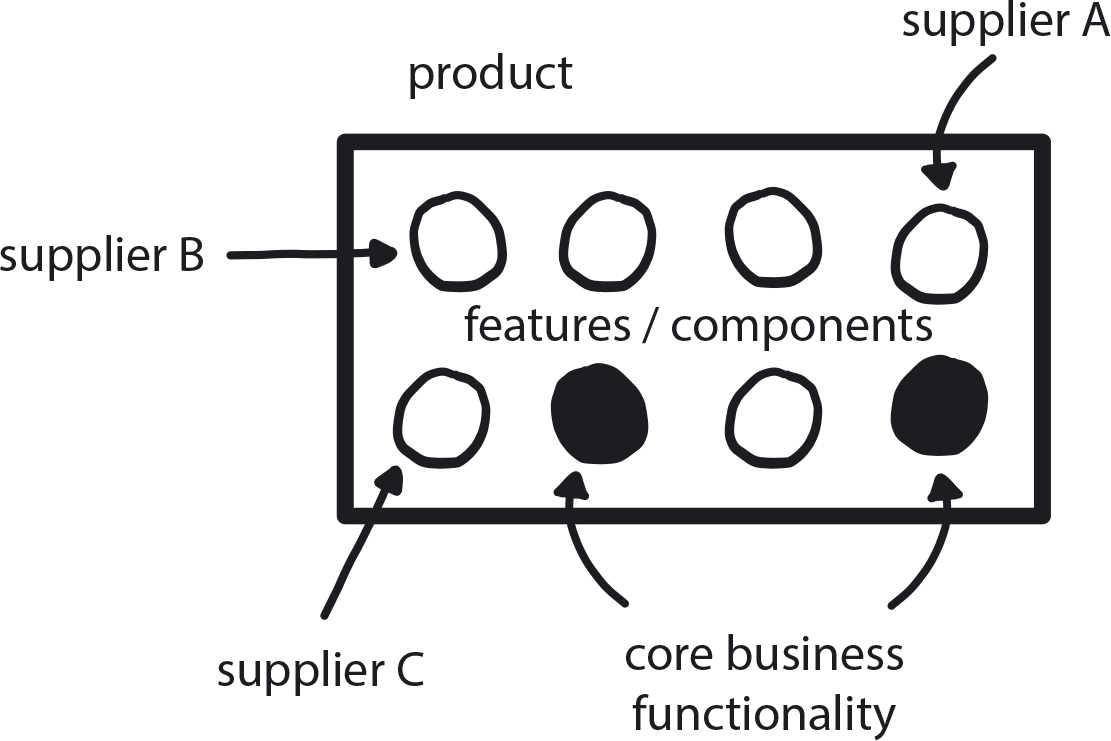
**Cloud Everything**

When I say Starling is a cloud bank, I mean it. All our processing is in the cloud, all our balances, accounts, services, APIs, everything. Initially this was purely AWS, but we are becoming increasingly cloud-neutral as we mature to address some of the regulatory and commercial challenges we face. By building portably and making use of open source technologies, it’s possible to remain largely vendor neutral, and even in advance of running redundantly across more than one cloud provider, we maintain AWS exit plans.

Only when we are absolutely forced to host a physical piece of hardware (like a Hardware Security Module for instance) do we use space in traditional data centers.

As well as our core services, a lot of our tooling consists of cloud services. We have over 100 SaaS [Software-as-a-Service] services in use, outsourcing as much non-core business functionality as we can: many of our customer support capabilities and items like SMS delivery rely on cloud integrations to some degree. This is great for all sorts of reasons. Not having to run them is really important. (Side note: this does complicate some of our disaster-recovery planning, because if we’ve lost AWS, it is likely we’ve lost half of the tooling that we use to actually deliver our software. Imagine a world where *every* service dependent on AWS has instantaneously and irrevocably disappeared, and suppose that’s a world in which you’re trying to rebuild a bank. Yep—that’s a disaster. We have to consider it.)

**PATTERN: AVOID REINVENTING THE WHEEL**

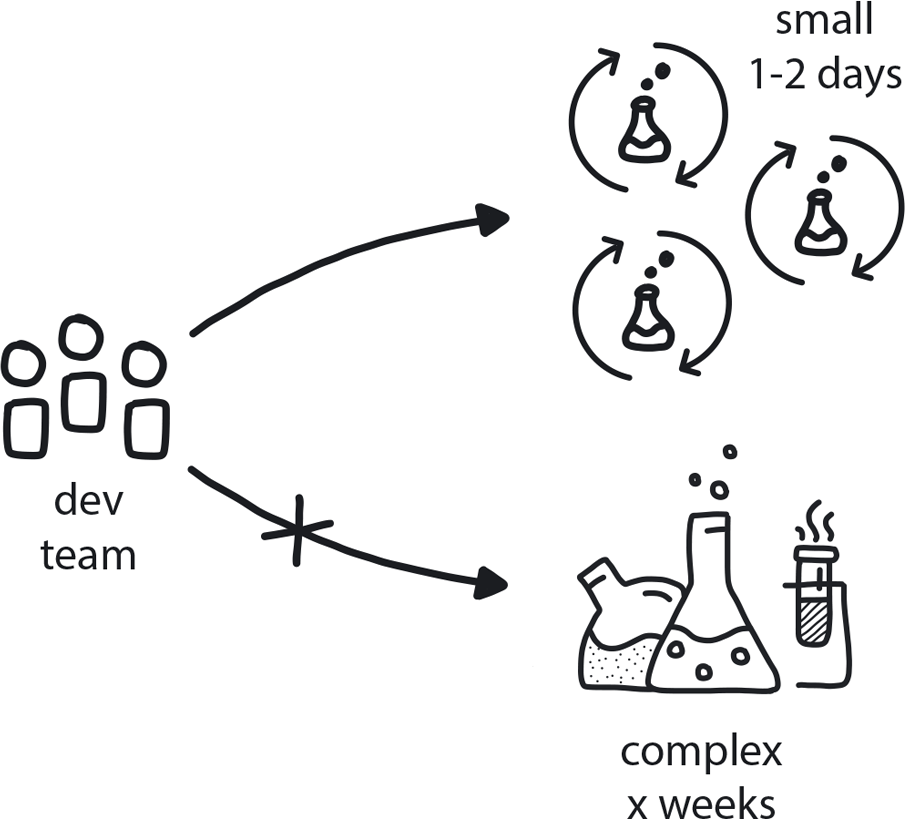


**Figure 14-13. Avoid Reinventing the Wheel**

When possible, purchase solutions for any need that is not your actual core business instead of trying to custom-build perfect tools.

Most of these cloud benefits are pretty well-known, but one benefit that I don’t think gets enough appreciation from companies considering a cloud migration is the ability to experiment. If you’re in AWS and you want to try something out with a hundred servers, then you try it out for an hour, and if it’s rubbish, you stop. Who cares? No cost. You can’t do that sort of thing with on-prem infrastructure!

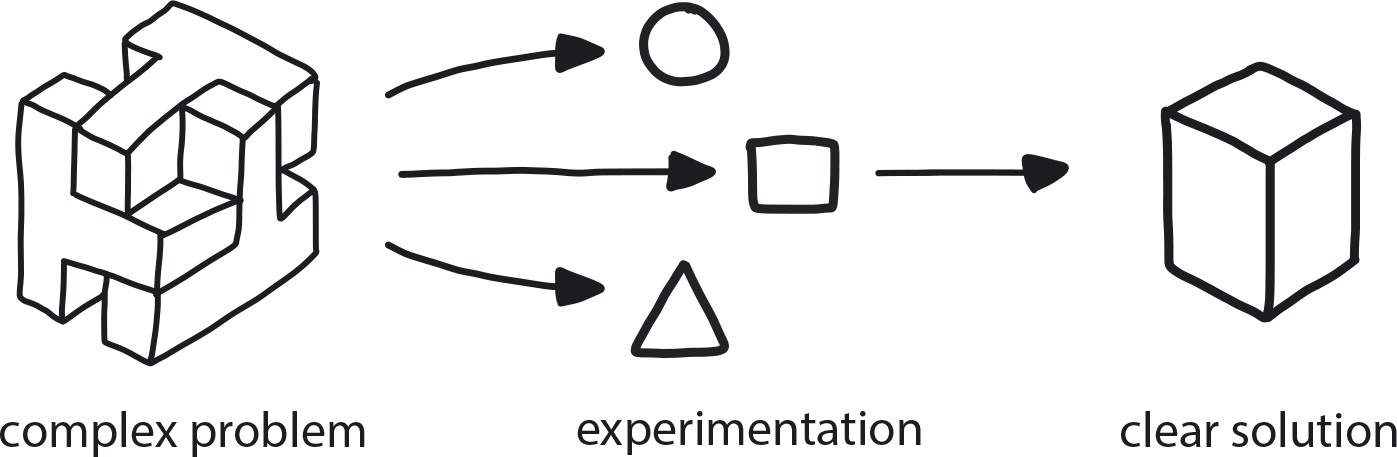
**PATTERN: REDUCE THE COST OF EXPERIMENTATION**



**Figure 14-14. Reduce the Cost of Experimentation**

When someone has an idea to validate, the costs (both financial and organizational) of doing experiments around it needs to be as low as possible.

**PATTERN: EXPLORATORY EXPERIMENTS**



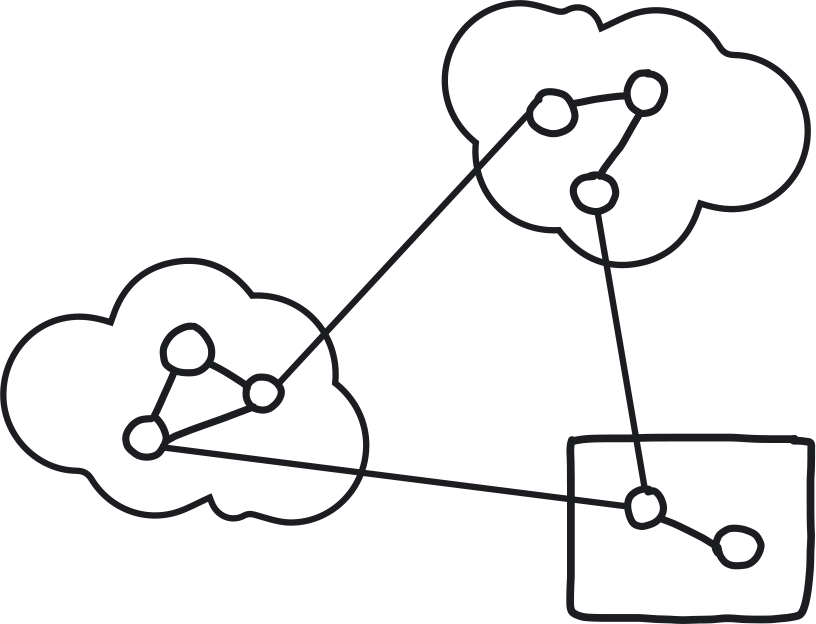
**Figure 14-15. Exploratory Experiments**

When dealing with a complex problem with no obvious available solution, run a series of small experiments to evaluate the possible alternatives and learn by doing.

**Resilient Architecture**

This last principle does get a bit techie but summarizes what was important to us as we built for resilience: self-contained systems [SCS]. This is a distributed system architecture approach, or manifesto, I guess, which is broadly the same as microservices. In fact SCS was a bigger influence on our architecture than the microservices movement. The SCS approach shares a lot of concepts with microservices: isolated, independently deployable units, the lack of centralized infrastructure—there is no getting away from distributed systems in cloud native!—and the ability to make technology choices. But there is less of an emphasis on size (the “micro” bit) leaving more of the emphasis on the independence.

**PATTERN: DISTRIBUTED SYSTEM**



**Figure 14-16. Distributed Systems**

When software is built as a series of fully independent, loosely coupled services, the resulting system is, by design, very fast, resilient, and highly scalable.

SCS inspired our approach to our services, which, really, I should call miniservices rather than microservices, because they’re not that small. We’ve got about 30. There are organizations out there, even small ones, running thousands of microservices That’s a very different architecture. I often think that, as well as our motivations, they’ve started out from a point of view that regards many of the in-language facilities for segregation and abstraction as somehow bankrupt or passé. Well, we don’t accept that. We don’t write that way. Our SCS services are larger and fewer in number than microservices. But they are each strongly autonomous, independently deployable, they each have their own database, they don’t share any infrastructure. They don’t even have a Kubernetes cluster or a data layer in common.

What SCS means is splitting an application into its required functionalities and making each one of these a standalone system. The overall app becomes essentially a collaboration of many smaller self-contained software systems, all delivered independently via the cloud. Each SCS is an autonomous web application that can fulfill its primary use cases on its own, without having to rely on other systems being available. The SCS approach fits with our “Conwayization” trajectory quite well. The fewer, larger components make things simpler, and in many cases we can avail ourselves of the facilities offered by the language or the database where they might not be available with a more fine-grained microservices architecture, while hopefully avoiding the problem of a large monolith that eventually becomes unsustainable. But this is some fairly subtle cloud-native architecture philosophy that could be a book on its own, so let’s talk about immutable infrastructure and chaos instead!

Immutable infrastructure to us means crash-safe architecture where we can kill any of our servers at any point and it doesn’t matter. There is no state or configuration on any of our servers that cannot be automatically reproduced from scratch in a minute or two. We can blow away servers at will and other identical ones rise in their place. We have our own chaos demon that sits there and kills four or five servers a day in production just to make sure that we can. For us, killing a production server is considered a safe operation. Occasionally we will do more targeted tests as well to prove database failover and suchlike, and there are times you have to be creative to do this. We also use synthetic load in production when it makes sense. There was a time where we didn’t have enough card traffic to really be confident that our services would scale up, so we were running a hundred thousand fake card authorizations in our infrastructure every day just to make sure that it was all there. If we had a sudden surge, we could dial back the fake load to allow space for real load.

We do zero-downtime releases and because we’ve got immutable infrastructure, we can do this mainly just by killing servers. So we update some settings somewhere, then we kill servers one by one, and then they come back at the newer versions and it’s all very nice.

We designed our architecture to be resilient but at the end of the day we know it is resilient, because we are continually beating it up ourselves.

We have diverged from pure SCS architecture in some ways to suit our own business needs, and eventually we coined the name DITTO[**1**](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch14.html#ch14fn1) for our homegrown approach to architecture. DITTO: do idempotent things to others. Basically, this is taking self-contained systems, which gives us our sort of microservices—our independent autonomously running, independently deployable services—and governing how they interact. It covers how we keep things operationally simple (no buses or external caches, just a “bag of services”) while still ensuring loose coupling (async over HTTP via “202 Accepted,” DNS for service discovery) and resilience (idempotence and pervasive retry processing).

Again this is just a principled selection of trade-offs: we accept some complexity of development, for instance, building some capability directly into our services rather than offloading it to specialized external components, in return for a system that has some nice operational characteristics, chief amongst them resilience and simplicity and portability of deployment.

**And, Really, That’s It**

So no radical new secrets revealed here, just a lot of careful thinking about what Starling needed and how to architect this entirely on the cloud for optimal pace and resilience. Our original motivation for going cloud native was because we believed that it would help us move faster. We believed that by using Infrastructure-as-a-Service, DevOps, and continuous delivery, we would organically grow an innovation culture in our tech teams.

Where we ended up: CD and DevOps plus DITTO architecture gave us a super-resilient system, even in the face of bugs and human failure, both of which are inevitable.

All of which enables us to move fast enough to deliver plenty of UK banking firsts: first to deliver in-app provisioning of both Google Pay and Apple Pay. First current account available to the UK public entirely on the cloud. And so on.

Architecture brought us here. Chaos keeps us honest.

[**1**](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch14.html#ch14fn1-marker) In fact, credit for this must go to Adrian Cockcroft who was none too impressed with my previous acronym LOASCTDITTEO (Lots of Autonomous Services Continually Trying To Do Idempotent Things To Each Other). I’m still rather fond of the old version.