**Chapter 15. Welcome to the Jungle: Adidas Cloud Native Transformation Case Study**

*By Daniel Eichten*

*Adidas is the largest sportswear manufacturer in Europe and the second largest in the world. With an ambitious plan to not just retain its significant share of a fast-paced, highly competitive market but continue growing it even further, the company turned to cloud-native technology to help hone its competitive edge.*

*Daniel Eichten, senior director for platform engineering at Adidas, helped lead the company’s successful journey through what he calls “the IT jungle” and onto the cloud.*

**Welcome to the Jungle**

We knew we needed a platform that would help us gain speed. For us, this really means speed of development of our products, but also speed of delivering our solutions for internal IT. So the goal was speed all over the place, wherever it makes sense.

The metaphor I use to explain IT at Adidas is to think of it as a jungle. Nowadays IT departments are not only the people who are building a lot of things but also the guides who help you through the IT jungle. We see ourselves more as partners for the business side to help orient them in the jungle and make sure they don’t get lost. Being in a real jungle can be exciting but also quite dangerous, and it’s the same with an IT jungle. Very exciting when you do all of the crazy stuff with AR [augmented reality], VR [virtual reality], blockchain, containers, and so on—or, for my team, cloud native and serverless—but it can also sometimes be very, very dangerous for a company if you take the wrong assumptions or simply turn the wrong direction.

Of course, you have animals living in the jungle, and one of the animals in the Adidas IT jungle is the elephant, our traditional core IT systems. This is the stuff that we develop ourselves, the enterprise Java applications, so a couple of hundred thousand lines or even sometimes a couple million lines of code. We call them elephants because they are quite heavy and they take a lot of time to develop. So if you want to create a new elephant, usually it’s a period of two years until this elephant comes to life. It’s a long project. Controlling these elephants and keeping them alive also requires a special handler, trained in taking proper care of them.

There are good things about elephants: they can get a lot of work done, and they can move really big things. They are very robust creatures. But they are cumbersome, very slow, and they are hard to train to do new things.

At the other end of the animal kingdom, then, you have the tiniest of creatures, the ants. These are the microservices, which can also do a lot of things and get a great deal of work done because there are so very many of them, all very busy at the same time. These tiny creatures are not very smart one by one, but all together they can accomplish amazing tasks. They can actually move the same load an elephant can, but by working in a completely different manner: they cut it into a million tiny pieces, and each ant takes one piece. The challenge is how to control them and direct them. In nature ants communicate via pheromones; IT ants, microservices, are controlled by systems that give all the little creatures direction and coordinate them.

In between you have carnivores, your crocodiles and jaguars. At Adidas crocodiles are our SAP systems, our enterprise resource planning and data processing software. They look the same as 200 million years ago. And then the newest visitor to the IT jungle is the jaguar, or Salesforce, our software as a service customer-relationship platform. These are both dangerous creatures because you don’t see them much; they mostly stay out of sight—but they are eating all of your budget. Both are quite expensive, for the licenses and usage.

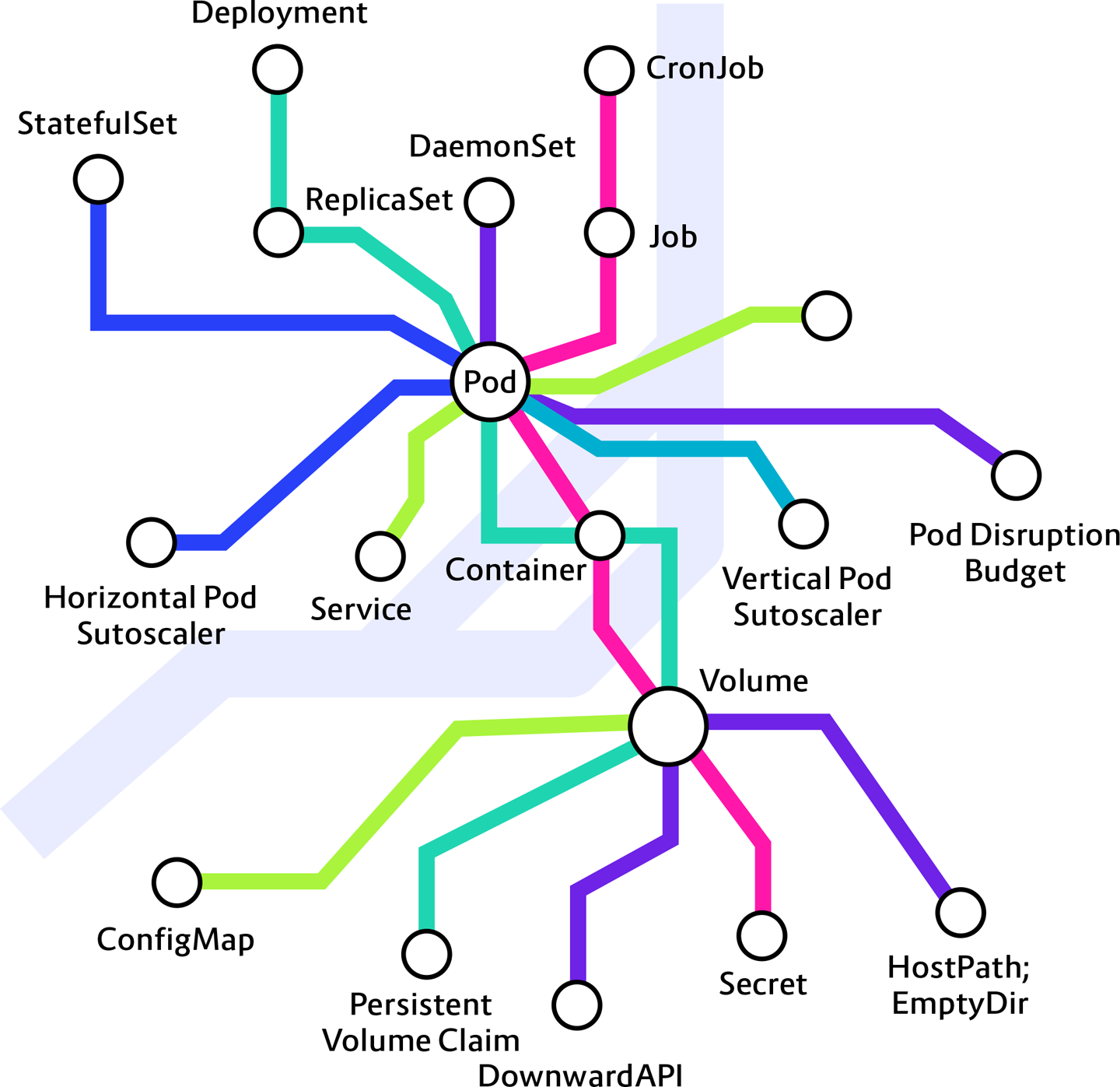
Our goal then was to tame the IT jungle. To build a cloud native platform that could be as reliable as an elephant, as fast and as efficient as a colony of ants, and to also take on the work of the crocodile and jaguar.

**A Playground for Kubernetes**

That’s when our Day Zero came, when we said, “OK, we have to do things differently. What should this new world look like, what do we need it to contain?” We sat together and collected some requirements: it should work on-premise. It has to be open source. We don’t want to have any vendor lock-in. It should be operable, it should be observable, it should be elastic, it should be resilient, it should be agile, and so on. And when we looked at the last five points, this is exactly what the Cloud Native Foundation came up with as their definition for cloud native. Everything embedded into CNCF also checked against these five items. So for us it was pretty clear that everything we were looking for could be found in cloud native.

And we looked a little bit around and then we found something like [Figure 15-1](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch15.html#map_of_the_kubernetes_api), which is not the map of some city transportation system, but the structure of the API for Kubernetes.

What it really looked like, to us, was the map out of the IT jungle: something that made containers and microservices understandable and controllable, that could harness all those ants to move the big things we needed. But what we also saw was complexity. In Kubernetes we found something that made impossible things possible but easy things rather complicated. Knowing that there were a lot of unknowns, we understood that this was nothing we could run on our own. We needed a guide, a partner who could really help us to get that pretty map up and running all of the different clouds, in our data center and so on. We found that partner in Giant Swarm. They were the only company willing to not just give us the product, but take the operations, give us consulting, and do that in all of the environments we needed to cover. No one else was, at least in 2016, really willing to go that far. All of the other companies were trying to sell their products, but they did not have the guide service we needed. They would do their demos, and then I would say, “OK, it’s all nice, I can install it on AWS and on-prem, but whom do I call if something breaks?” And then they really had to admit, “Yeah, sorry, we don’t do that kind of support.”



**Figure 15-1. Map of the Kubernetes API (image courtesy of Adidas)**

What Giant Swarm did for us was to provide a consistent toolset around Kubernetes, essentially a Kubernetes distribution, and help us to implement it successfully. Our Day 1 still didn’t come for a while, even after we found Giant Swarm, because enterprise processes always take a little bit longer than expected. It was not until 2017 that we were able to begin building.

Giant Swarm started us off with a Kubernetes playground cluster that they provided in their environment. This is a very good way to begin: learning by doing, exploring. You install, you de-install, you run stuff and see what happens. After some time we had created multiple namespaces. *[Note: A Kubernetes namespace provides the scope for container pods, services, and deployments in a given cluster.]* So I forcibly deleted all namespaces and our cluster made “boom” and exploded, and nothing was working anymore. Pure panic in the entire team! But it was a good learning experience.

More importantly, it was also a clear indicator that the tools that we had always used before for CI/CD, for building stuff, for compiling stuff, for testing stuff, even for running and monitoring stuff—these were not the tools that would help us in the future.

Because we were shifting really from storing, MSIs and JARs and NPMs, to storing containers, we needed a new toolset around CI/CD, around our code repositories. We needed new monitoring (Microsoft SCOM proposed by the infrastructure department were definitely not working like a charm in a container environment). And we also had to rethink security, obviously. But where to get started? How to know the right tools to pick?

We got in touch with the Cloud Native Computing Foundation, which at that point in time, 2016, was very helpful because you could ask, “Hey, what do you have in the area of XYZ?” and the answer was very simple. The original landscape map had only a few things. For each area you picked that tool from the shelf, you used it, and it was fine. The landscape started with Kubernetes. Then Prometheus for monitoring, which we also use, and they added Envoy as a proxy, which we use implicitly through Istio but not explicitly.

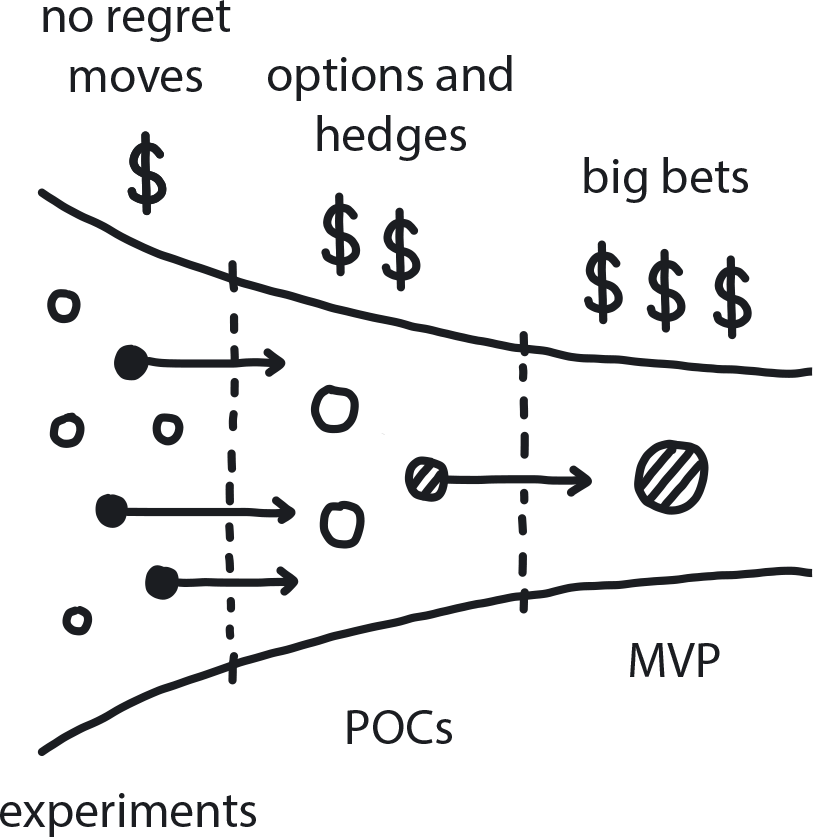
Fast travel to 2019, though, and the CNCF landscape map now has over 1,000 tools and options. So currently the ecosystem is kind of a mess, and a problem for anyone trying to build a proper platform toolset now. In some cases it’s easy: if you just look at container runtimes you have a couple of options, which are official CNCF-incubated or even graduated projects. For other use cases, though, it’s really, really, really hard to pick. So many options now, and how to know which ones will endure?

Fortunately, we did not have so many choices at the time, and the ones available worked out well for us. So we got started running with Kubernetes and Prometheus, producing and operating a full environment until we felt confident we were ready to move on to Day 2 and put something into actual live production on our new cloud-native platform. But what would we try first?

**Day 2**

It had to be something meaningful. No one is giving us applause or any great feedback or additional money for further projects if we play it safe now and, say, move some of our HR department’s request forms from an external SaaS service to an internal cluster. That would be useful but not a big change for the company. A really big change that we did need was the thing that was becoming our biggest retail door: our e-commerce store. That was a big reach, risky to get wrong. But we decided only with great risk do you get great rewards—and more project funding, right?

**PATTERN: GRADUALLY RAISING THE STAKES**



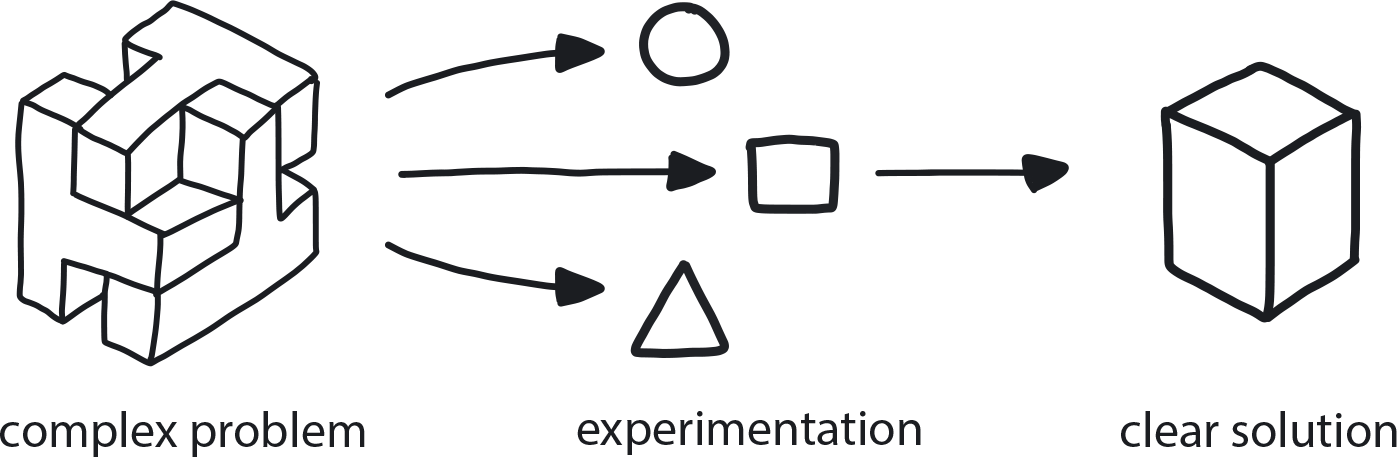
**Figure 15-2. Gradually Raising the Stakes**

In an uncertain environment, slowly increase investment into learning and information-gathering; eventually you uncover enough information to reduce risk and make better-informed decisions.

*See also the related next-step patterns No Regret Moves, Options & Hedges, and Big Bet.*

To keep the risk contained somewhat we started with migrating just our Finland e-commerce platform; there’s not too much volume on that one. When we did the first small tests we started easy, like opening a faucet very slowly. We opened up the valves a little bit and everything was working like a charm, and the website was fast and responding great, and it was pretty cool. So then we increased the throughput a little bit more, expecting that to also go very smoothly, but it turned into the cluster being on fire. Panic all over again! But after breathing into some paper bags to relax, we set out to figure out what went wrong. Again, the cluster was fine, but there was a cluster performance load-testing tool that we were still learning. It turns out there is a setting applicable to single nodes but that we thought was applicable all over the cluster, so when we spun it up we accidentally tested 100 times the amount of throughput we had intended.

**PATTERN: EXPLORATORY EXPERIMENTS**



**Figure 15-3. 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.

We were a bit guilty of trying to solve imaginary scaling issues, which taught us another very important lesson: when you pick a new tool set, you can’t expect to immediately be the master of it. And you have to be open-minded to working differently than you always did before. Because of my experience before cloud native, when the cluster was on fire I thought it had to be a cluster issue, something wrong with the cluster itself. It took some time to realize the cluster was fine—we were actually testing the wrong thing.

There are some very technical details here, beyond the scope of the patterns we’ve presented. They’re included to bring a deeper insight into this use case. The story itself is what’s important here, and even non-technical readers can understand what happened even if the more technical terms are unfamiliar.

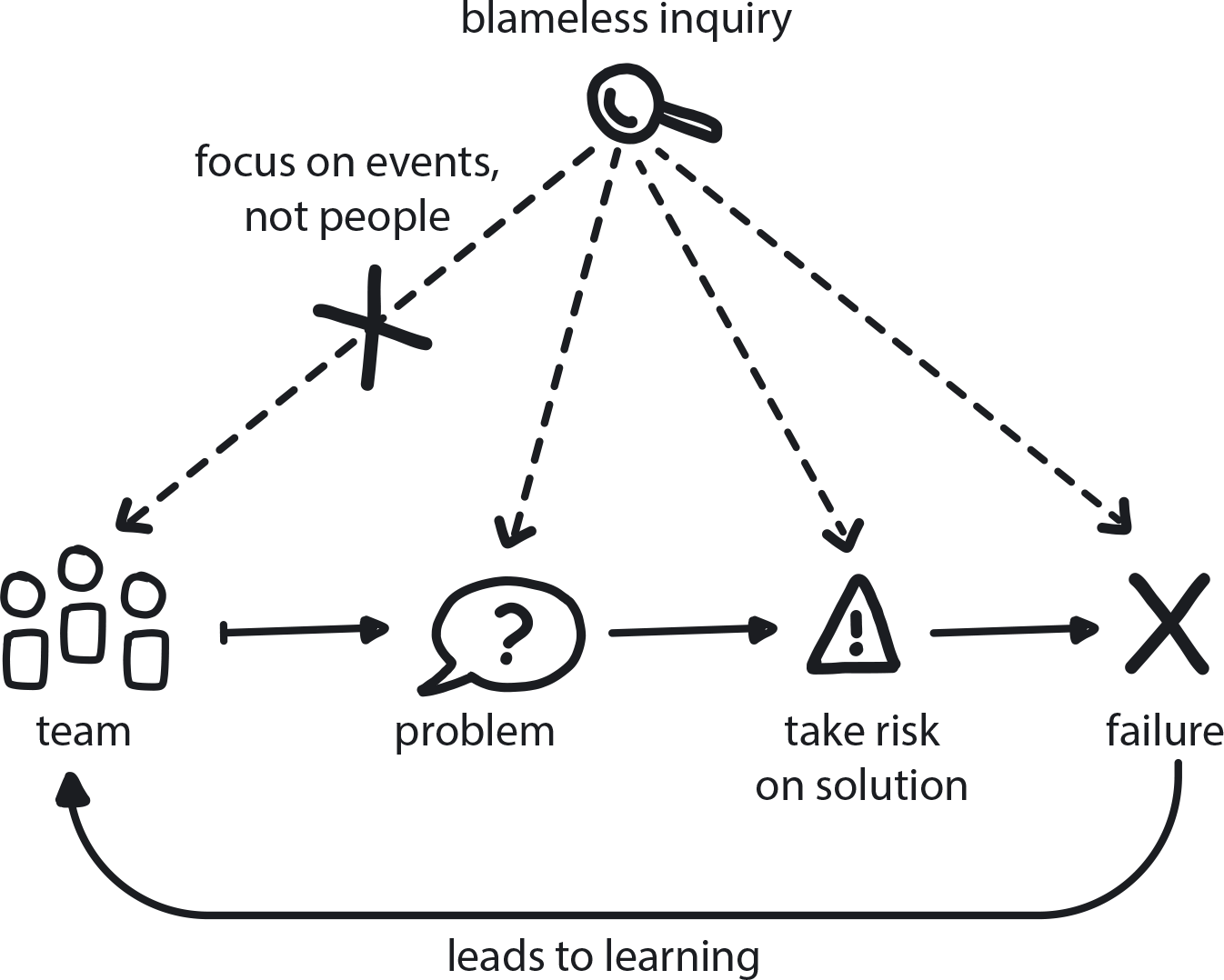
At this point, we were still Day 2, but I would say around lunchtime. It was October 2017, approaching the period where holiday sales start and we have e-commerce traffic spikes for Black Friday, Cyber Monday, and so on. By now we already migrated most of the countries to the cloud data center in preparation, and we had an unexpected opportunity to test on a global scale. Adidas introduced a special edition shoe designed by Pharrell Williams that suddenly went through the roof. People were going crazy, hammering the servers and the systems, trying to grab a pair of these shoes. The cluster was on fire and went down. We investigated and found the cluster running the e-com app was actually OK—instead, the application itself was not responding anymore. It turns out our e-com application was fighting with our ingress controller for resources, and apparently the application won. So whenever the ingress controller was forwarding a request to the application, the application was eating so much CPU that the ingress controller was not even able to respond anymore to handshake, and this is when Kubernetes went in and killed it.

What did we do wrong? Well, we completely forgot that we had to put some reservations there for core components. So, again the iterative approach—failure, and what do you learn from it? We actually were happy it had happened with just this one item, because we got to learn that lesson before Black Friday.

So we learned from it and we prepared for the next big sale event. We scaled everything up, everyone was dialed into a war room, a virtual one, we sat ready and waiting … and then none of the countries showed up. What? No traffic?! This time it was because of some caching feature implemented on our CDN [content delivery network] level, which was not letting any traffic through to the cluster.

This brings us to the practice of outage-driven infrastructure. Because, to be honest, similar situations happened a few more times, and we aren’t the only ones. Other platform engineers I talk to say it is the same for them, to the point where there is a nice GitHub repository from Zalando, another European e-commerce retail site, sharing their and other production screw-up stories and all the different ways they have killed their clusters. In the meantime this turned into a whole website, [*https://k8s.af/*](https://k8s.af/). Our count isn’t quite there yet. But it was good to be able to show that repo to management to show that this learning curve is normal, and it is steep, and we aren’t the only ones who have to go through it.

**PATTERN: BLAMELESS INQUIRY**



**Figure 15-4. Blameless Inquiry**

When a problem occurs, focusing on the event instead of the people involved allows them to learn from mistakes without fear of punishment.

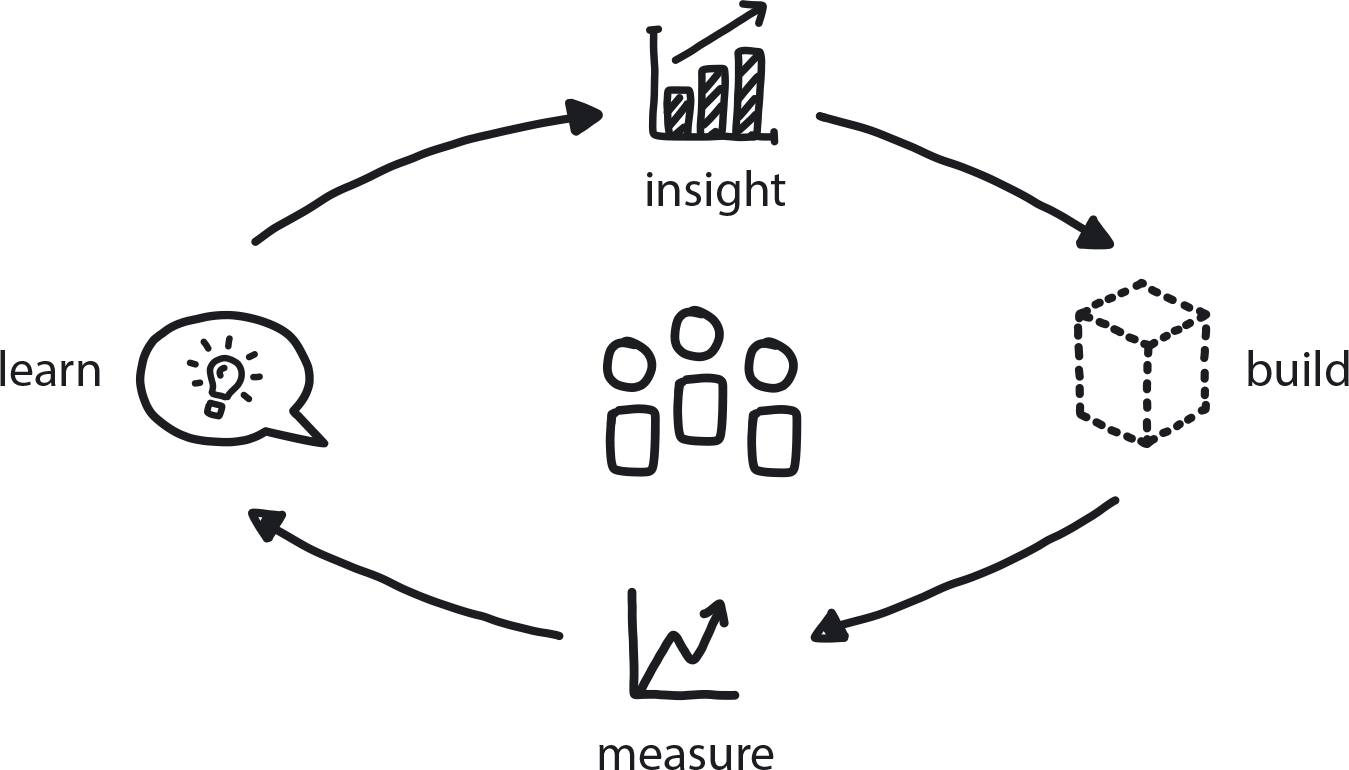
To help guide our learning curve, we use a product pyramid to help guide decision making on how we’re developing our e-commerce application for resilience, speed, and every other metric, including not blowing up. [Figure 15-5](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch15.html#the_product_development_pyramid) shows the one the Adidas cloud platform team uses.



**Figure 15-5. The product development pyramid tool used by the cloud platform team at Adidas**

This pyramid helps us prioritize and think through: how do we do release management? How do we do incident management? After we had these kind of events, though, we realized there were gaps in our pyramid. These gaps included doing proper postmortems, which was only partially missing—we did it for the big ones, and we learned from it, so that’s great. But there were plenty of smaller events where we didn’t do postmortems, and then sometimes running multiple times into the same problem.

**PATTERN: LEARNING ORGANIZATION**

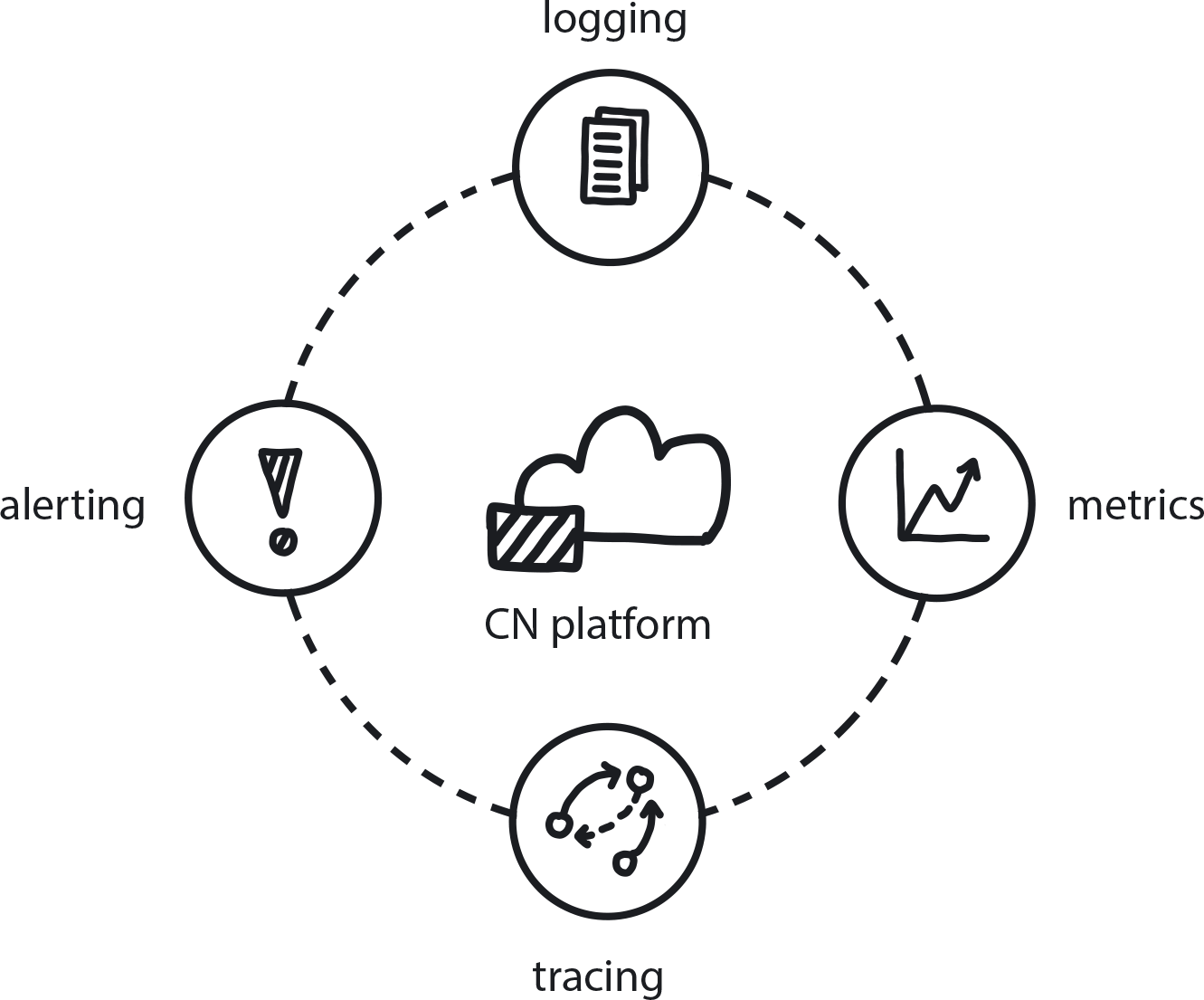


**Figure 15-6. Learning Organization**

An organization skilled at acquiring information, creating insight, and transferring knowledge can tolerate risk with confidence and solve difficult problems through experimentation and innovation.

Another gap we needed to address in our Day 2 platform version was proper capacity planning. And there I don’t mean infrastructure so much as how we can develop to make the best use of the resources we already have. Another very important missing piece—so important that it’s on the bottom holding up everything else—is monitoring, or the better word is observability. This is something that we worked on heavily all over the place and constantly; now, for us, everything begins with observability.

**PATTERN: OBSERVABILITY**

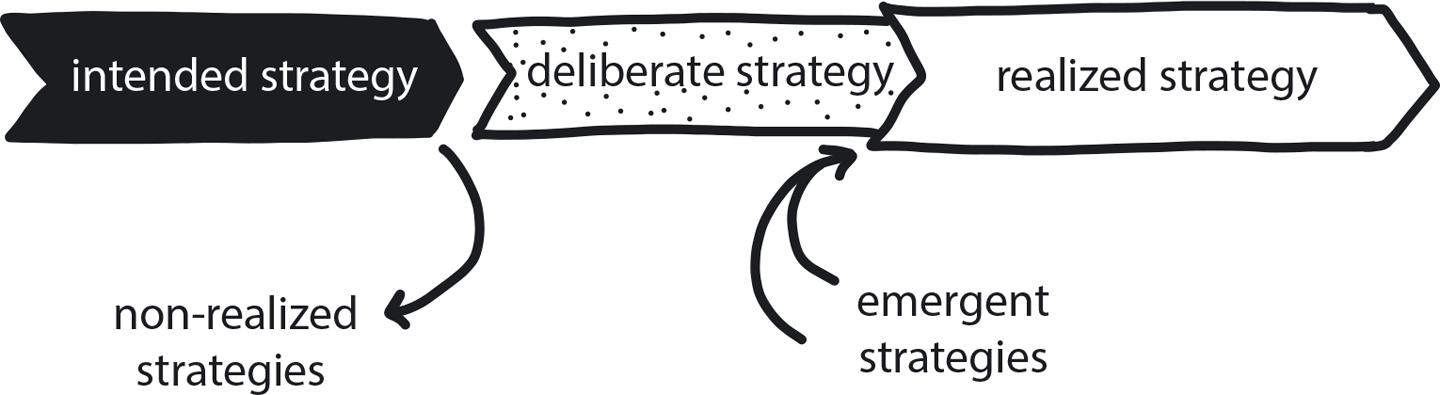


**Figure 15-7. Observability**

Cloud native distributed systems require constant insight into the behavior of all running services in order to understand the system’s behavior and to predict potential problems or incidents.

Speaking of beginning. If there is one thing I wish I had known when we started, it would be that, with cloud native, sometimes decisions don’t stay decided for very long. And that this is actually OK. This is now reflected in our cloud strategy, to constantly revise what we do.

**PATTERN: DYNAMIC STRATEGY**



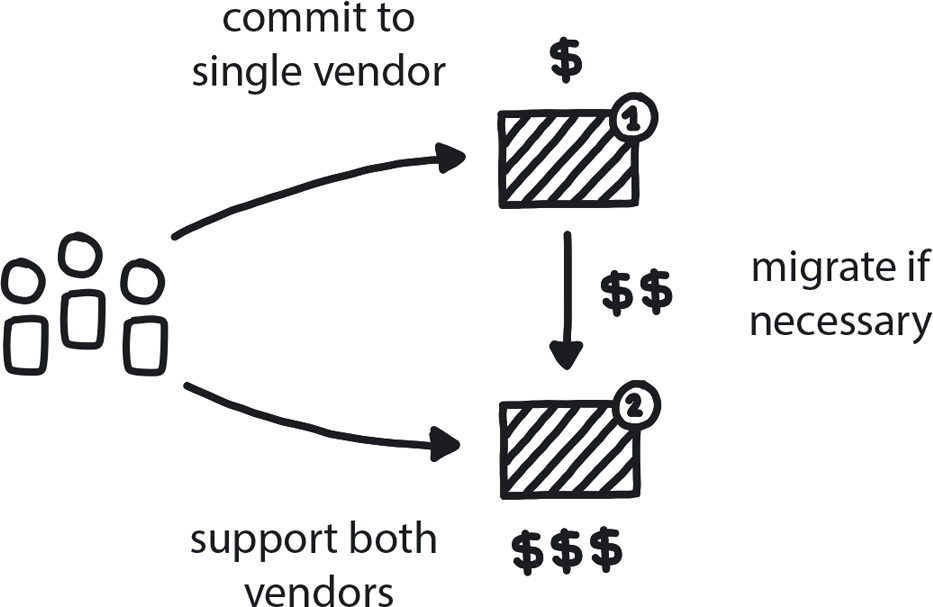
**Figure 15-8. Dynamic Strategy**

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.

There was another decision which we made early on, when we didn’t know enough to be making such big commitments, that slowed us down quite a bit. This was the assumption that, in order to be as agnostic as we can possibly be, everything had to work and feel on-prem exactly as it is in the cloud. This assumption was already wrong as soon as we made it because the engineers working on applications that are running on-prem are so different in their mindset and in the solutions they develop. They don’t have to think about the crazy scaling patterns like you have sometimes in the cloud.

So now we have to go back and revise the results of this early decision. I think we could have saved a lot of time and a lot of manual work if we had not reinvented the wheel to make sure our cloud also worked on-prem. If we just would have said from Minute One, “OK, it’s one cloud vendor, and we use the highest-level services we can find on them. And only if there’s a good reason do we go down and build it ourselves.” I think that’s the one thing I would say which would have sped up the process quite heavily.

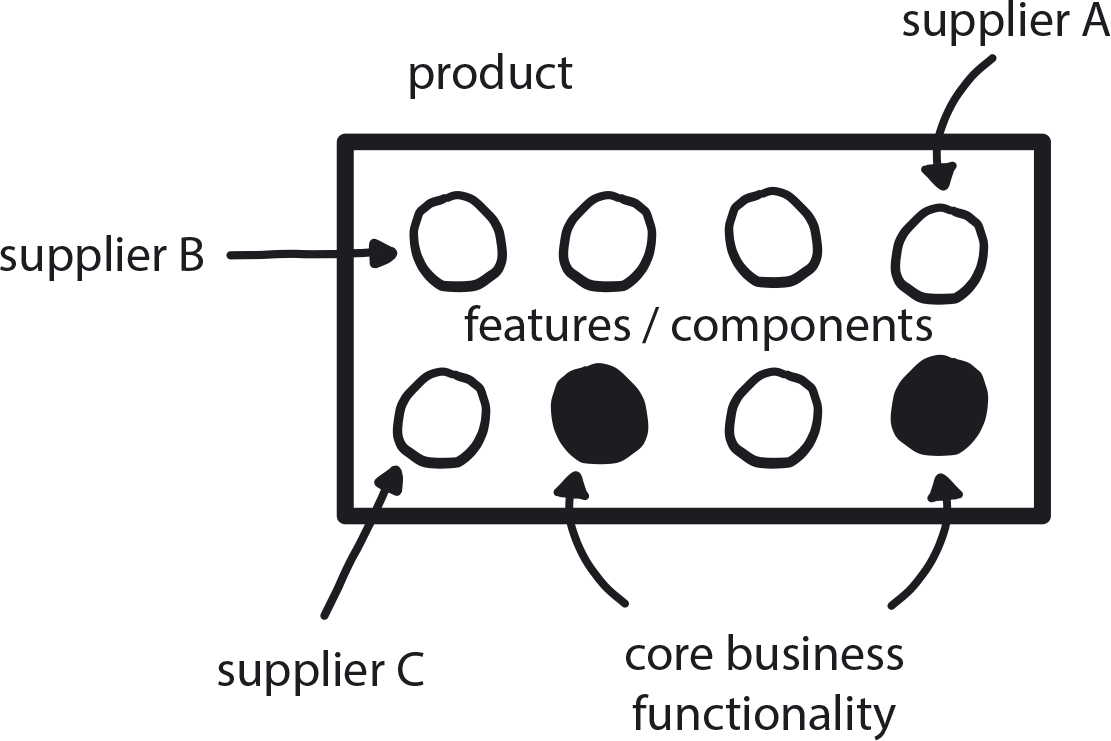
**PATTERN: EXIT STRATEGY OVER VENDOR LOCK-IN**



**Figure 15-9. Exit Strategy Over Vendor Lock-In**

Public cloud vendors can handle all aspects of building and operating a cloud native platform, and their tools are often excellent—but when committing to a vendor/technology/platform, it’s important to identify an alternate solution and any costs associated with switching over.

**PATTERN: AVOID REINVENTING THE WHEEL**



**Figure 15-10. 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.*

Ultimately, building so much custom to make sure everything worked on-premises as well as in the cloud is nothing I regret, even though it slowed us down. It’s always good to know how things work and the insights gained are valuable. Like now I can buy a managed Kubernetes and I still would know how things would work underneath. That’s always a good thing, right?

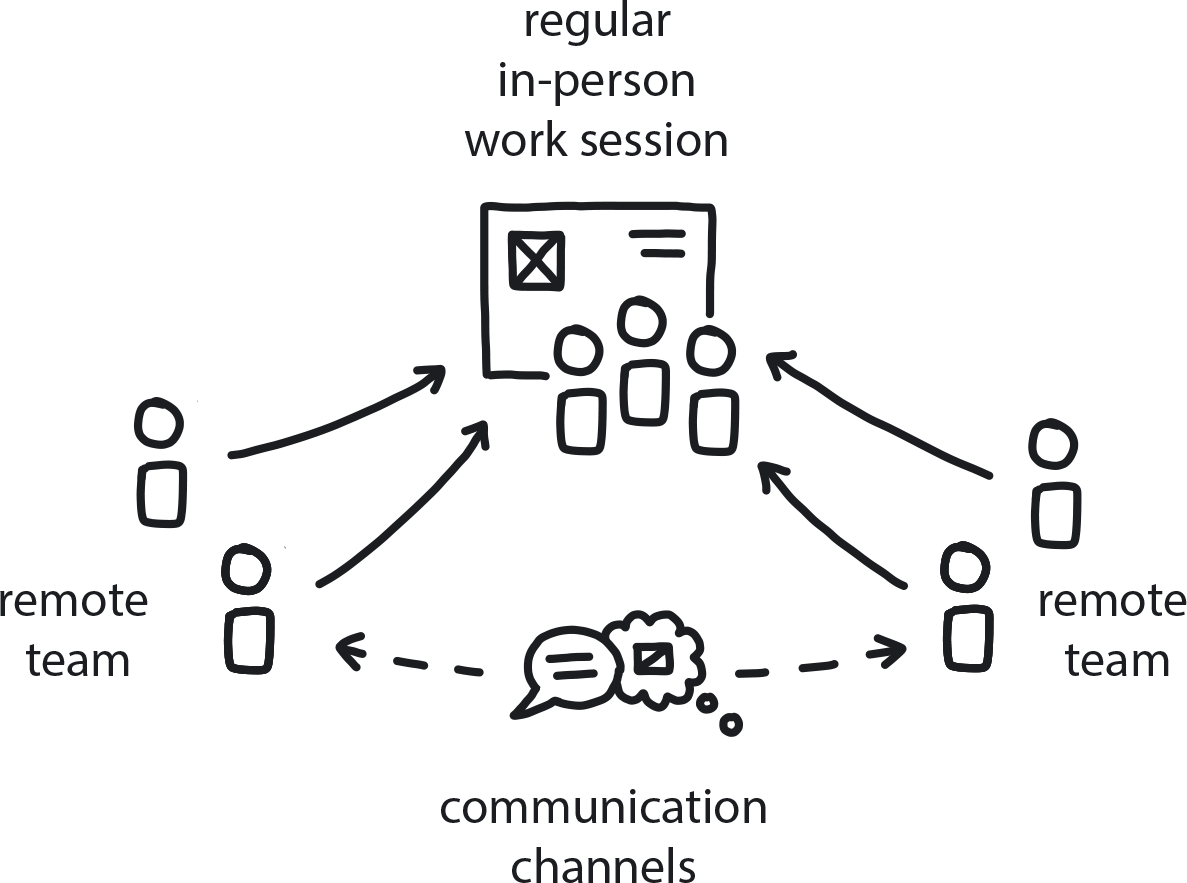
**Day 2 Afternoon in Cloud City**

And that leads us up to now, which we can say is afternoon on Day 2 in the city that we have grown.

From one cluster and a single e-commerce application, we have now grown to five global locations with, in some cases, more than one production cluster each. We are in China, we are on-premise, we are in Singapore, we have clusters in Ireland, we have clusters in Oregon. And when I say clusters, that’s always Kubernetes. In each location we have Kafka clusters next to it [*Author’s note: Apache Kafka is a high-throughput distributed messaging system used to facilitate scalable data collection.*] and all the infrastructure necessary to also get all the monitoring and observability tools, plus also our CI/CD processes. So, each of these production clusters goes from core to content.

This story tells our engineering technology journey. What I did not realize when we started was how it would also become an engineering team journey. Four years ago the engineering team was a handful of people, heads down, trying to work the best we can, unfortunately not talking much to each other because we were really spread across the organization—there was no one uniformed engineering department. Turn the clock forward to now, when our most recent companywide engineering day was 600 people, not only engineers but also service managers, ops people, architects, and even some business product owners.

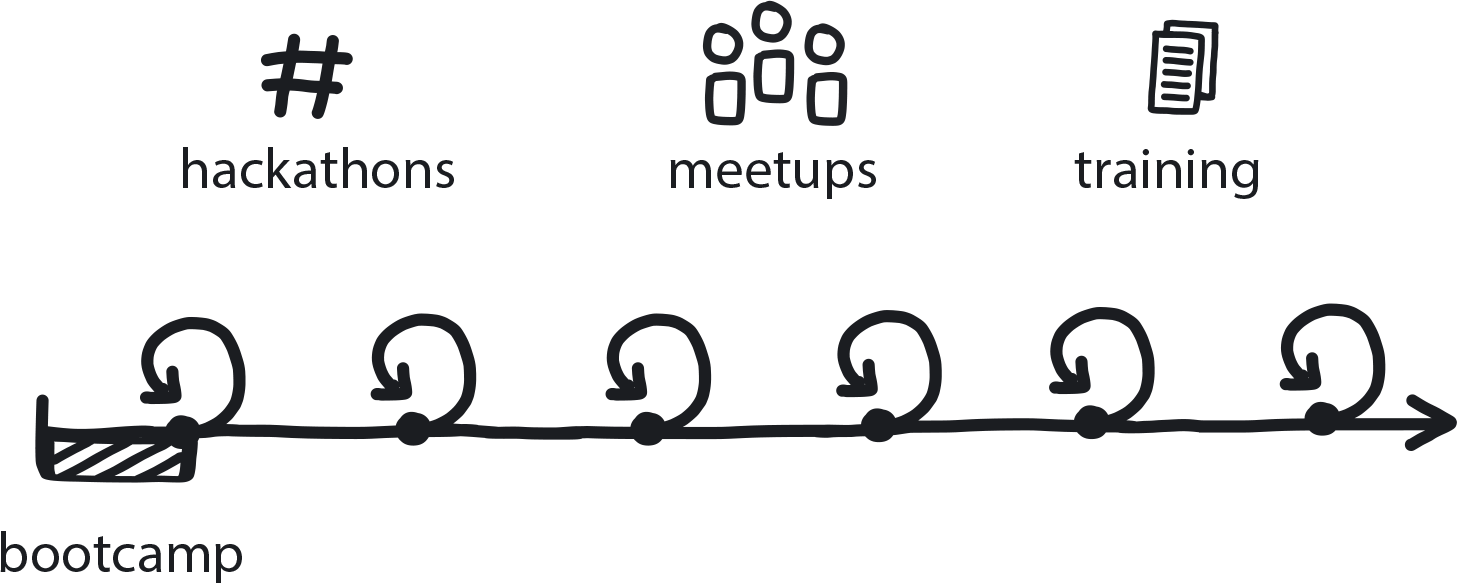
**PATTERN: REMOTE TEAMS**



**Figure 15-11. Remote Teams**

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

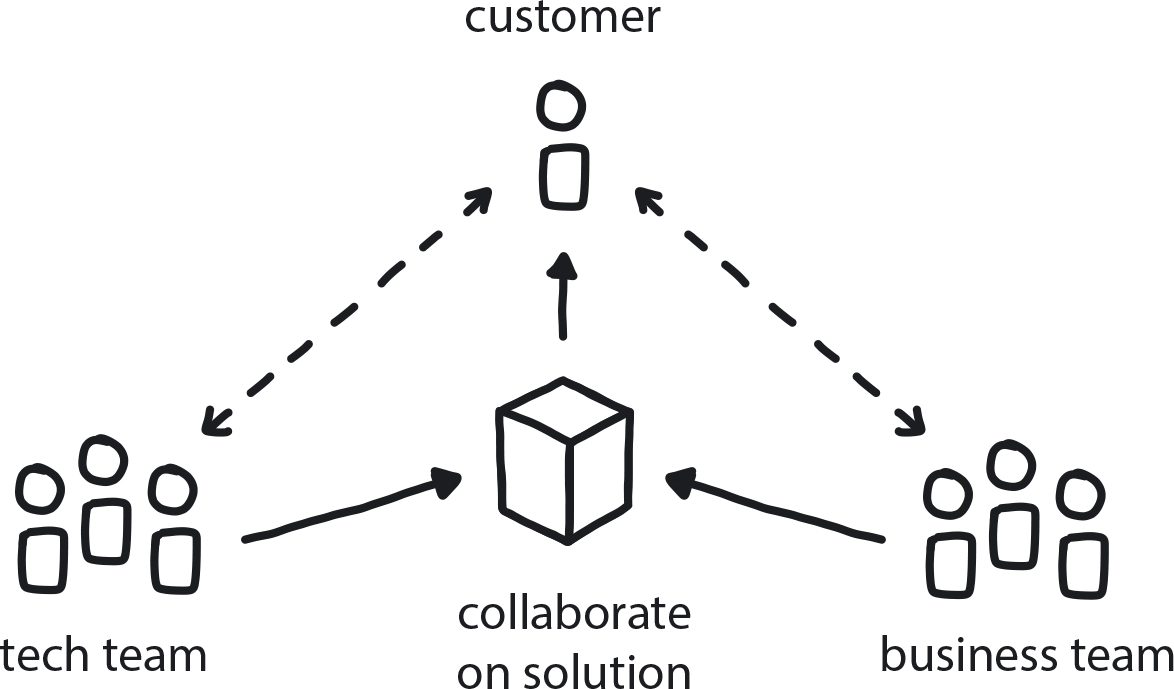
**PATTERN: ONGOING EDUCATION**



**Figure 15-12. Ongoing Education**

Prioritize ways to help teams continually develop their cloud native knowledge and skills.

**PATTERN: INVOLVE THE BUSINESS**



**Figure 15-13. Involve the Business**

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

Numbers wise, now we have 300-plus internal engineers. We’re still working heavily with extra partners, too, so that gives us up to 2000+ Bitbucket accounts. (One of these partners is Container Solutions, who we originally brought in to create ongoing cloud native education programs, trainings, hackathons, etc.)

Our CI/CD system does roughly 100,000 builds per month. Every week we gather 25 terabytes of metrics on our central logging infrastructure. The data lake is over 750 terabytes as of today. Beyond 10 million Kafka messages and it’s constantly growing because we now include more and more systems. We have nearly 4,000 Kubernetes pods across all of our clusters and the code base in Bitbucket recently crossed the border of 100 million lines of code. We also make heavy use of all the tools available on the AWS platform, 28 of those at last count, including Amazon E2, ECS, CloudWatch, Elasticsearch, and Amazon API Gateway.

These numbers are always changing, too, and what that says is this is not simply a new technology that you get installed and then it’s done. In the same way you can’t say, “We are training now everyone on Agile, and therefore we are an Agile organization!” It simply doesn’t work that way; there’s no “set it and forget it.” All of these things have to be in place and working in tandem. The processes have to go hand in hand with the technology, and both must be hand in hand with the culture: from the top down the management style and the organization itself has to change.

This is truly the most important lesson of all, the number one thing to know about succeeding with cloud native: If you don’t have all three of these things evolving together, your tech, your processes, and your people, well. I don’t want to say your transformation is headed for certain failure, but you will have to hit some hard boundaries and some hard rocks before you really can get through.