**Chapter 12. Applying the Patterns: A Cloud Native Transformation Design, Part 2**

*Welcome to the second half of our detailed design pattern for a cloud native transformation. Here we will lay out patterns from start to finish, explaining the order and reason for the choices as we go along. It’s a lengthy and involved process, so we have divided it into two chapters. Part 1 covered the period from pre-initiative prep through the point where we have researched, experimented, and prototyped our way to uncovering the likely best transformation path. In Part 2 we begin verifying the path by building a production-ready cloud native platform, move through onboarding everyone onto the new system with a new way of working, and then finish by shutting down the old one.*

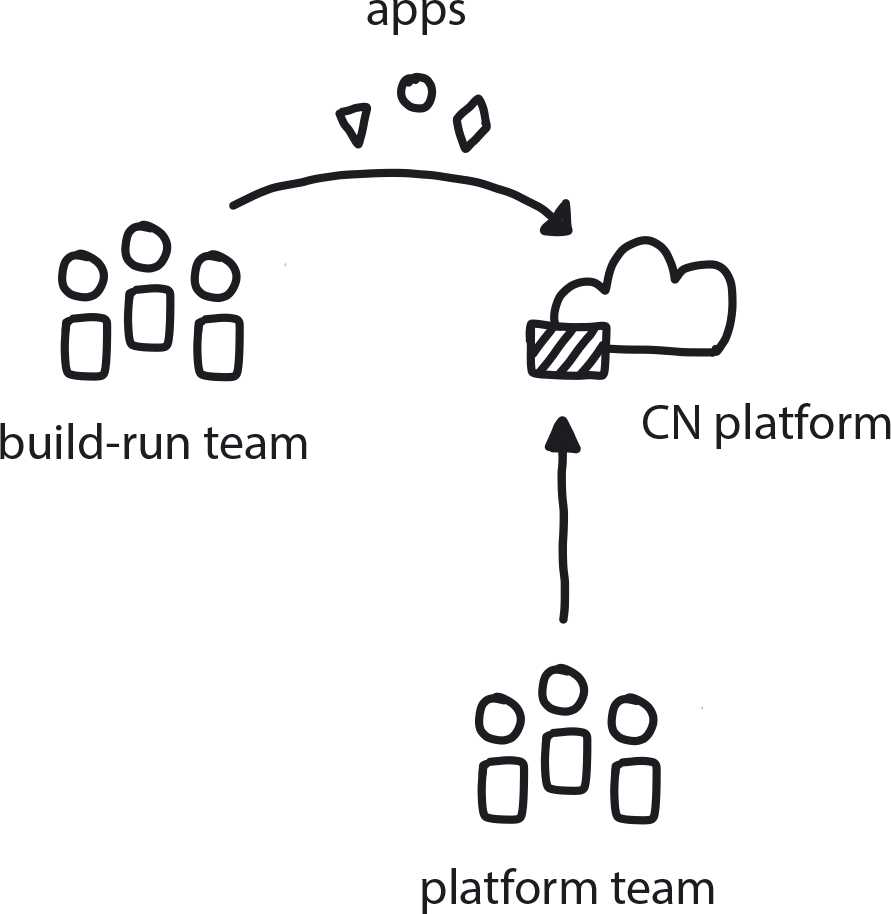
**PHASE 3: BUILD**

Phase 2, the Design stage, concludes with a major turning point in the transformation: the goal is clear, the tech is available, substantial knowledge has been gained, and all major questions are answered, thanks to a series of PoCs. Now is the time to build a real platform. Development teams are on hold and waiting for the new platform to start building new applications and refactoring old ones.

The MVP Platform pattern is the bridge between designing the new cloud native system and then actually building a functional, production-ready version. It’s the final test of whichever winning candidate emerged from your PoCs.

This is also a pivotal moment for the Core Team, because this is the point where the team often divides and multiplies. Frequently, several members split off to form the Platform Team, while the remaining Core Team members focus on getting the first test application ready and preparing materials for eventually onboarding all the rest of the teams in the company. (Remember them, off to one side, keeping the old system running? We told you their turn would come, and it’s almost here!)

**PATTERN: PLATFORM TEAM**



**Figure 12-1. Platform Team**

Create a team to oversee architecting, building, and running a single, consistent, and stable cloud native platform for use by the entire organization, so that developers can focus on building applications instead of configuring infrastructure.

The Platform Team is an essential component of cloud native systems’ architecture. It is also separate, by design, from all the other engineering teams in an organization.

Cloud native developers know how to build for distributed systems. They know how to build in network and security and all other necessities so all parts of the distributed system become part of the application. However, these applications still need to deploy somewhere. A common mistake companies make is expecting their developers to not just build applications but also take charge of the platform they run on, all in the name of DevOps. This is detrimental in many ways, but there is one problem to rule them all: if there is not one standard, unified platform in place, each application team will build their own version.

This is how you end up with seven different systems deployed in different ways with different tools—even running on different clouds. Everything is random, and it’s impossible to refactor these to a standardized platform for production. This is exactly what we saw happen during WealthGrid’s second transformation attempt.

To prevent this all too common problem from ever arising in the first place, we have the Platform Team pattern. A platform team sits off to one side from the rest of the engineering teams and is fully invested in designing and maintaining the platform to make sure it is stable and provides all needed functionality. Devs don’t need to worry about provisioning or any other infrastructure details, they just deploy to the platform. Platform people handle Kubernetes and below, while the devs work on top of Kubernetes with APIs. This line of separation may move as the emerging serverless and service-mesh paradigm becomes increasingly adopted into the cloud native stack and change the balance.

**EMERGING PATTERN: SERVICE MESH**

A service mesh is a dedicated infrastructure layer built right into an application to govern service-to-service communication, and one of cloud native’s most recent forward iterations. It is perhaps a bit too early to write a full Service Mesh pattern, but it’s definitely an emerging innovation to watch closely.

While Docker and Kubernetes reduce the operational burden on deployments by standardizing packaging and deployment, they do not address runtime. But what happens once the application is running? Even fully independent and autonomous microservices need to align with some kind of organizational standards to work together efficiently. The service mesh paradigm has risen to address this gap and standardize the runtime operations of microservices applications in the same way that Docker and Kubernetes standardized deployments.

(Service Mesh is also of service in deployment, too, where it serves as more intelligent and adaptive networking. Without something like a service mesh, you may need to add what is effectively networking or deployment knowledge into your microservice code.)

The service mesh is emerging as a potent new component of the cloud native stack. High-traffic companies like Netflix, Google, and Ticketmaster have added a service mesh to their production applications (Google in fact created Istio, an open source service mesh tool). Gartner and other industry research firms advise that in less than two years, a service mesh will simply be a requirement for any organization running microservices in production.

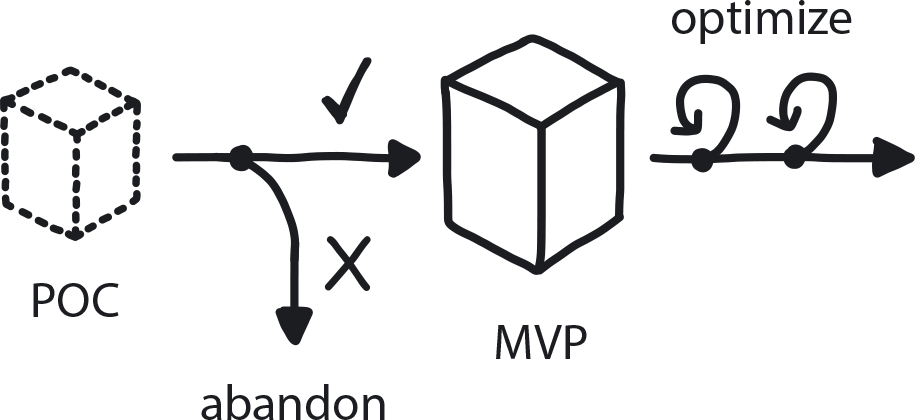
The technology is not quite ready for us to include in our current transformation design. It’s not mature enough for a company like WealthGrid to embrace at this time. But we predict that very soon people will be talking about Istio and Linkerd with the same excitement and reverence currently used for discussing Kubernetes. This is a prime example of how fast things are evolving and it is why we emphasize over and over that the core of cloud native is *how* we create and deliver software, not where.

When the Platform Team does its job well, standardization and developer freedom are both in place. Strong standardization is a good thing in that it limits support and maintenance costs and increases development velocity across the organization by providing a consistent set of tools and shared expectations. Too much standardization, however, shuts down any ability for devs to experiment with new tools and approaches.

The trick is finding the balance between freedom and separation: you want to standardize for economy of scale, but a fundamental principle of microservice architecture is choosing the best tool for the job. Thus the Platform Team’s prime directive is to optimize standardization while building a platform that allows developers to choose the best and most appropriate tools for their jobs.

The first job of the Platform Team is to create a minimum viable product version of the platform. Given that many if not all the team members came through the PoC process as part of the Core Team, they should be pretty well acquainted with the ins and outs of whatever candidates have been identified to go into the cloud native platform stack. This is where the MVP Platform pattern enters our transformation design.

**PATTERN: MVP PLATFORM**



**Figure 12-2. MVP Platform**

Once Exploratory Experiments and PoCs have uncovered a probable path to success, build a simple version of a basic but fully functional and production-ready platform with one to three small application running on it.

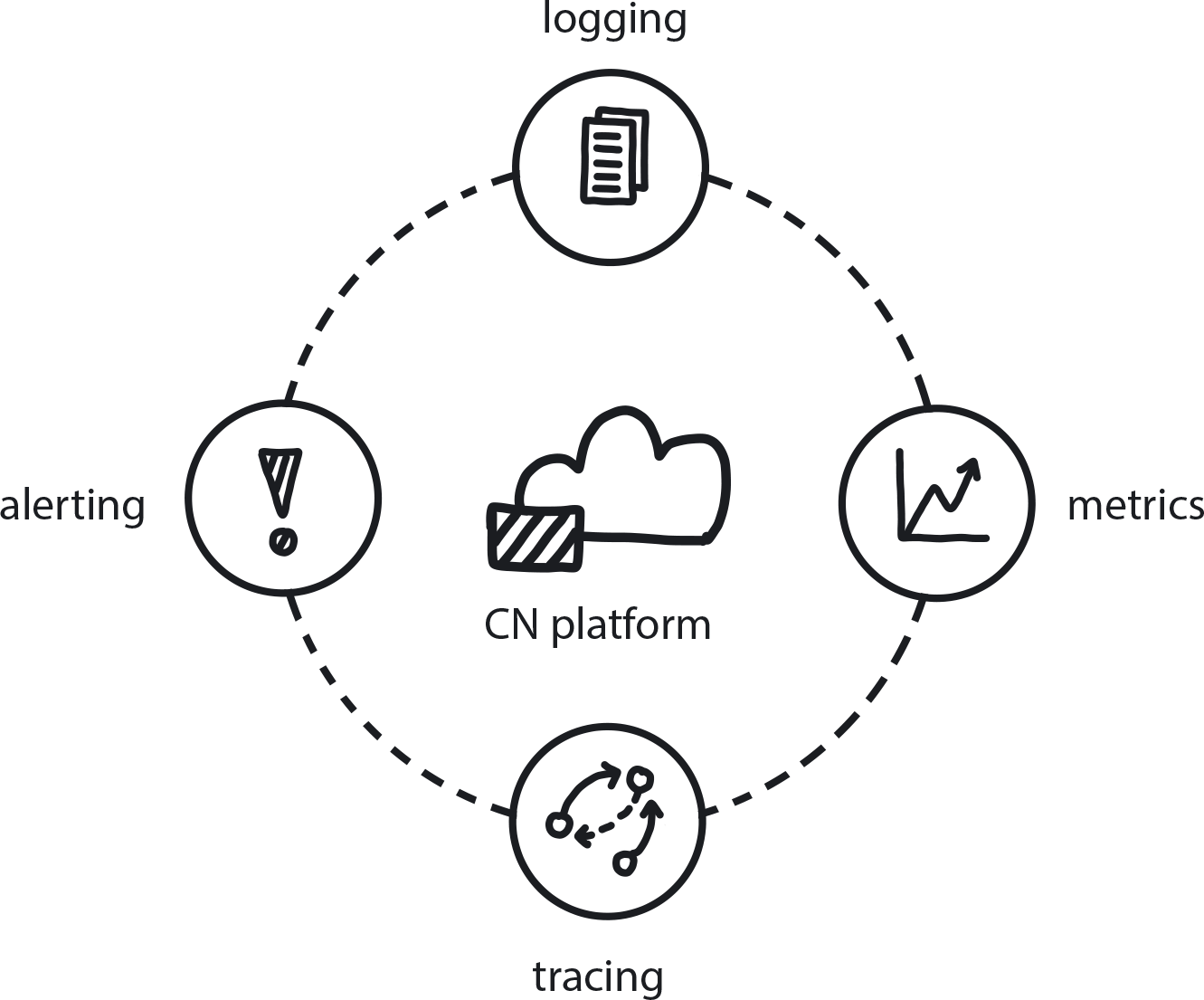
The MVP is a bare-bones system, built to have minimal yet useful functionality and in a very short period of time. Quick turnaround is key, because it’s important to get the platform in front of users as fast as possible to get feedback and make improvements. “Minimal usefulness” can vary according to your company’s needs, but it should serve at least some actual business functions. Extendability is critical. Even if this version does not ultimately go into production, you still need to understand how to continue expanding platform capabilities and capacity.

Don’t confuse the MVP with your final-version platform—we aren’t there yet! The MVP is meant only as a very basic thing that can be done in a few months by a few people, meant to serve as the basis for future development. It does need to be production ready. The point is to test the architecture in real-world conditions—but only on a small scale, with a small number of users. Yes, it runs, but running is not the same thing as ready to go into full production (though it is definitely a major step in the right direction). The MVP is far from feature complete; though it has security, monitoring, and observability in place, these are only rudimentary versions.

Doing all this should take two to six months to produce the first basic, bare-bones version, with Continuous Integration and Continuous Delivery in place from the very start. After that, definitely plan for an additional two to three phases to bring the platform to full functionality, refining as you go based on user response. Two important aspects of this full functionality to include in the MVP are Observability and Secure System from the Start.

As organizations move towards containerized workloads and dynamic microservice architectures, monitoring after the fact no longer suffices. Observability must be built in to give better insight into an application’s performance.

**PATTERN: OBSERVABILITY**



**Figure 12-3. 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.

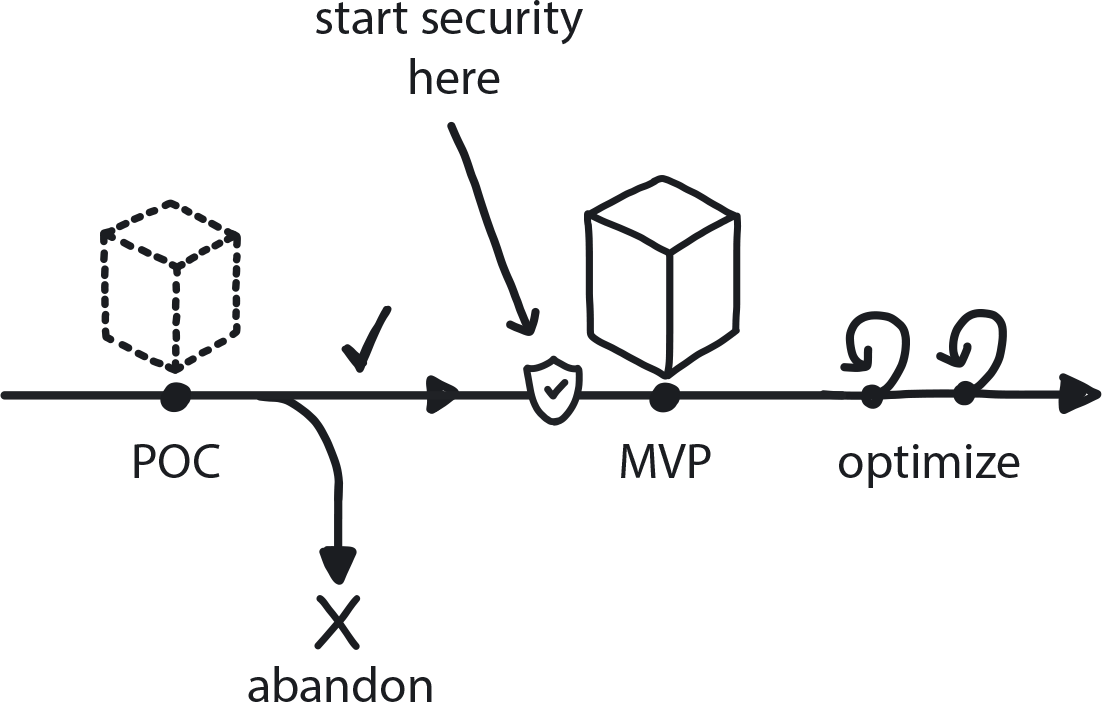
In cloud native you must observe the system to see what is going on. Monitoring is how we see this information, but Observability is the property we architect into a system so that we are able to discern internal states through monitoring external outputs. Furthermore, you have built resilience into the system because you are working from an assumption that any single component of MVP system may go down at some point. So you optimize by understanding things on a system level, rather than trying to track each separate component.

In a 100-container system, if one container fails, it’s nothing. A non-event. Kubernetes will restart it. But you do want to record that it happened in order to analyze trends in the system’s behavior.

Observability is one of the key elements of microservices architecture. Well-engineered services expose metrics that both allow efficient system monitoring and trace each interaction the customer has with it. Observability gives engineers the information they need to adapt systems and application architectures to be more stable and resilient. This, in turn, provides a feedback loop to developers that allows for fast iteration and adaptation to changing market conditions and customer needs.

Like Observability, security also needs to be baked directly into the platform from the very start. Perimeter security measures simply do not work in distributed systems. Unfortunately, transition teams tend to delay setting up security until the relatively late stages of a project and then attempt to bolt something on at the end.

**PATTERN: SECURE SYSTEM FROM THE START**



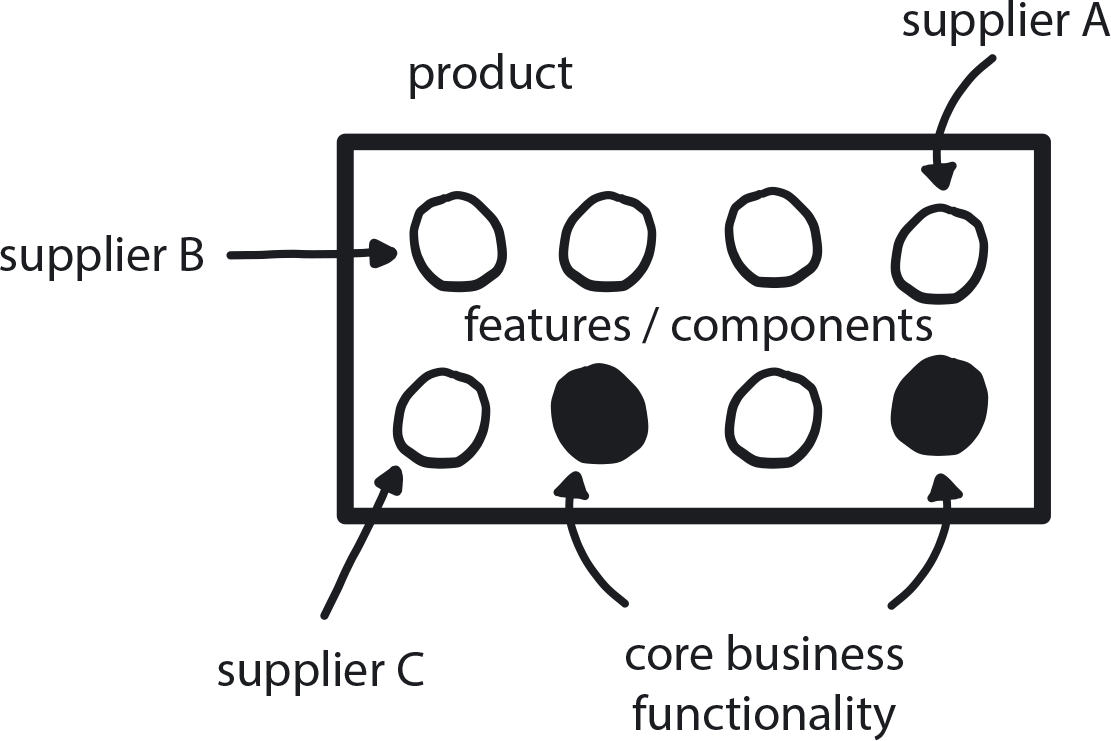
**Figure 12-4. Secure System from the Start**

Build security into the platform beginning with the earliest versions to ensure your distributed system is unbreachable by design.

The only way to handle a cloud native system’s complex security requirements is to build the MVP as a highly secure system from day one. This means providing in-depth security training for teams; choosing tools carefully and reviewing how they work together to check for gaps; using best practices to secure containers, clusters, APIs, access rights, etc. and ensuring that automated security testing is part of your workflow.

While considering the needs that must be fulfilled in the final version of the system, we have some good news: you certainly don’t have to custom-build every piece.

**PATTERN: AVOID REINVENTING THE WHEEL**



**Figure 12-5. 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.

Not every single piece of your platform needs to be custom built—cloud native is new, but it’s not *that* new. There are plenty of existing tools and services available for purchase to run as, for example, Software-as-a-Service. When assembling your platform, use existing tools when possible, especially for functionalities that aren’t part of your core business. Choose these even if the fit isn’t exactly perfect: Whether commercial or open source, third-party products are typically better quality, better maintained, and more frequently extended than anything you can build yourself.

Whew! Fully building out even a basic MVP, and even with some purchased pieces, is a pretty big achievement. Significant investments of budget, time, and work have now been invested in building and improving it. Once the MVP platform is ready, it’s time to pause and review the current transformation strategy to make sure it’s still valid before we move even further with extending MVP functionality. It could be painful now if, for some reason, it still turns out to be the wrong direction for the company; it will be painful indeed to abandon it and begin again. Better to find out now, though, than many months down the road.

**Prepare for Onboarding**

Achieving the MVP is big news, and a major transformation milestone: You now have a few apps in production on a real cloud native platform! This is a good time for celebration and even more evangelism.

The Platform Team will continue working to prepare the MVP for full production. Meanwhile, we’ve still got all those proficient teams working on the legacy products and delivering value to the customer in existing ways. Mostly they still need to keep doing what they’re doing; however, once we get close to achieving an MVP platform, it’s time to start preparing teams that will be gradually onboarding. Let’s go now to check in on the Core Team, which has been preparing the way for onboarding the rest of the organization onto the new system.

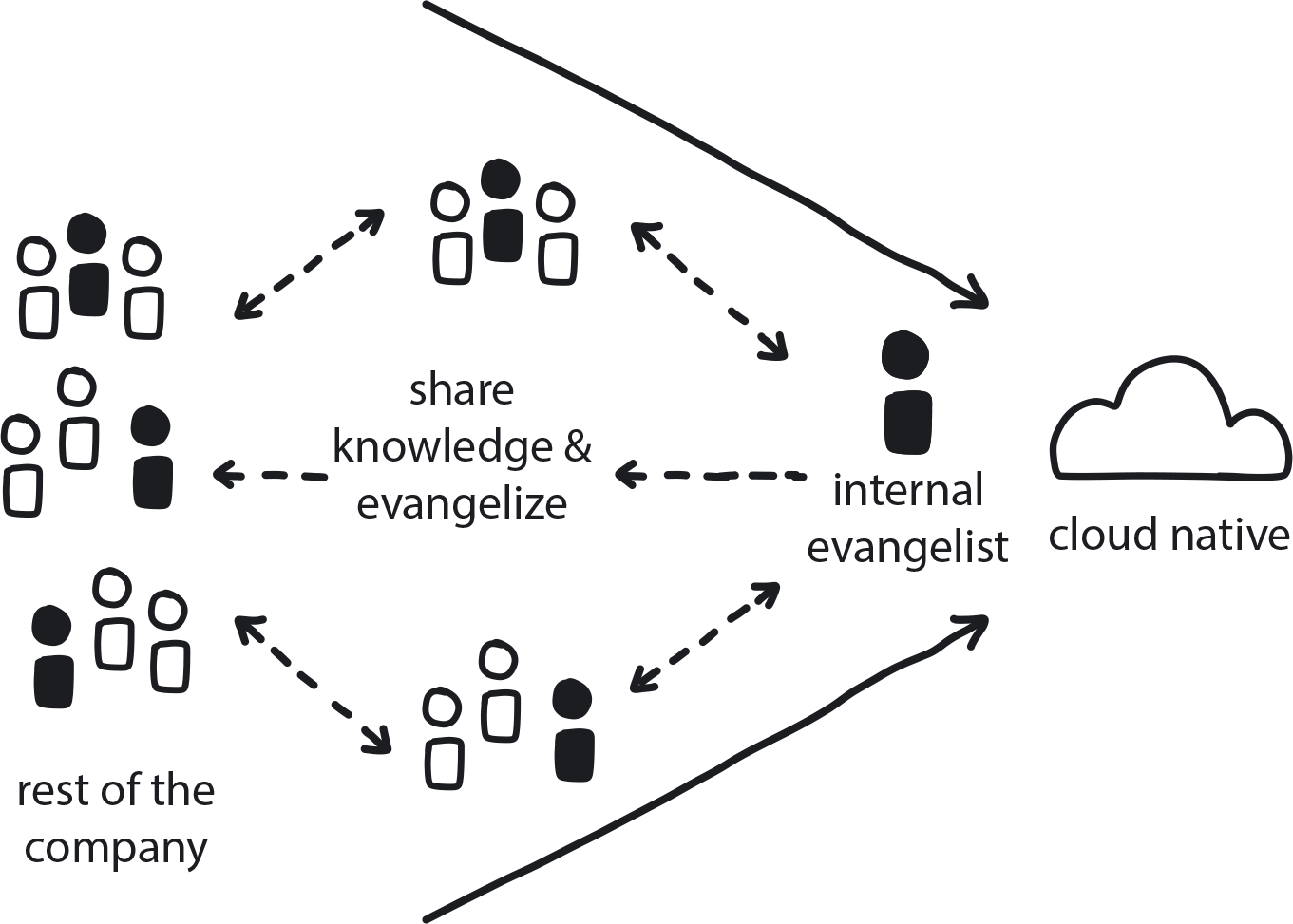
Prepare for onboarding means more than educating teams and moving them onto the new system, though. Now is the time where organizational culture also begins to shift in a broad and permanent way to reflect the company’s new cloud native approach. Pay attention now, because this is actually the most important part of the whole transformation. It’s where we begin the long, slow process of strangling both the technical and organizational monoliths.

At this point, word has almost certainly spread through the company that there is a big change coming. People are likely feeling anxiety that their work is going to change. They need to know when and how. When there is little available information, negative attribution bias leads people to assume the change is not going to be good for them. Actually, the majority of cognitive biases all rise from a lack of information one way or another, so preparation during the pre-onboarding period is all about reducing anxiety by sharing an abundance of information.

It’s important to give people opportunities to learn but equally important to make sure they understand that this is a gradual and long-term process. If you start sharing information about the transformation initiative from the Executive Commitment stage onward, by the time teams are actually scheduled to onboard, it will be a trivial procedure with little resistance. Indeed, if the groundwork is laid properly, they will be excited and ready for the change.

Internal Evangelism is the key to managing the pre-onboarding process, which should begin months before the actual onboarding itself. Starting up Internal Evangelism at the same time the MVP phase begins is a good move.

**PATTERN: INTERNAL EVANGELISM**



**Figure 12-6. Internal Evangelism**

Provide plenty of information about the transformation across the entire company right from the start to create understanding, acceptance of, and support for the initiative.

For the first six to 12 months of the initiative, only the Core Team has much active involvement in the transformation. With proper evangelism, however, people are excited and want to help, to be involved. Use this excitement to engage people: raise the level of general cloud native knowledge across the organization.

The Transformation Champion is often the person in charge of Internal Evangelism, organizing events like lunch and learns—“Come eat pizza and learn about microservices!”—to start building general knowledge across the organization. At this time, the emphasis is on basic education over dedicated training: how containers work, what Kubernetes does. Regular open sessions that anyone can join should be part of Internal Evangelism. Slowly the knowledge builds up so when the actual onboarding comes there is a feeling of familiarity and comfort with the concepts.

Meanwhile, the Transformation Champion is also directing internal marketing efforts to promote awareness of and engagement with the cloud native initiative across the entire organization, for example, sponsoring internal newsletters, hackathons, and regular demo sessions to show the platform development’s current status keep everyone informed and engaged. (Incidentally, these same tactics can, and really should, also be extended externally. This is a great opportunity to evangelize outside of the organization to start building a better technological reputation for the company. This will make the company more attractive for good engineers looking for interesting and challenging jobs.)

The lack of internal evangelism certainly damaged WealthGrid’s second transformation attempt. Even when they were throwing most of the engineering teams at the transformation effort, the rest of the organization had no insight into what was happening. The sales and marketing teams could only see that the company’s offerings were not being improved while their competitors were introducing new customer features, and there was a lot of frustration because no one thought to Involve the Business (see pattern). Finally, since there was no internal evangelism happening around the transformation, biases were likely at work—status quo bias, hostile attribution bias—creating internal resistance to the transformation and further causing it to lag or fail.

Internal events are opportunities to prevent all of that from happening in the first place. Talking about the plan, and the schedule for when the time does come for onboarding everyone onto the new system, makes sure everyone is informed, engaged, and has appropriate expectations. Don’t worry about predicting exact timeframes: People don’t mind if the transition gets delayed when they know that it is certain to happen.

**Onboarding the Right Way, at the Right Time**

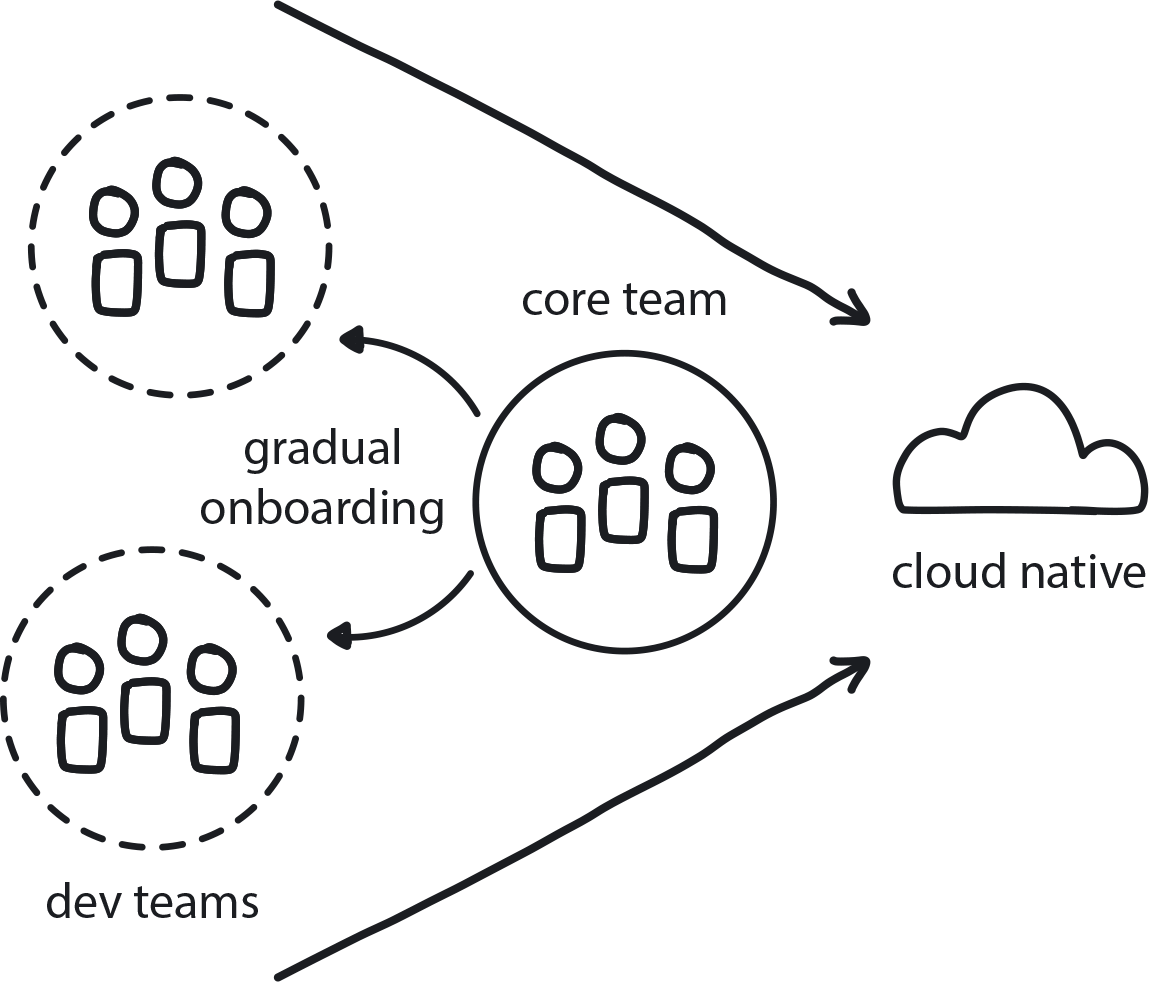
When the MVP platform is nearing full production readiness, we have reached another pivotal moment in the transformation process: it’s time to begin actively moving the proficient teams onto the new system and thoroughly educating them in using it.

It is important to approach onboarding in not just the right way, but also at the right time. Formal onboarding is the time where teams are slowly restructured into cloud native Build-Run teams, and taught to work in the new way, and it should only begin when the actual migration is very close to starting. This is a very important period for the teams—literally, their own moment of cloud native transformation—and requires an appropriate level of time and attention invested in moving them through it properly.

Only the teams specifically scheduled for onboarding (and only a few teams at a time) are actively engaged in the process; all others remain exactly as they are until their turn arrives. This is important because if you teach people new skills, they are excited to use them. If they have to wait six months or a year before they can apply their new knowledge, they are really frustrated—and, without use, these skills quickly erode anyway.

Again we have a super pattern, Gradual Onboarding, to get this evolution underway through application of its constituent sub-patterns.

**PATTERN: GRADUAL ONBOARDING**



**Figure 12-7. Gradual Onboarding**

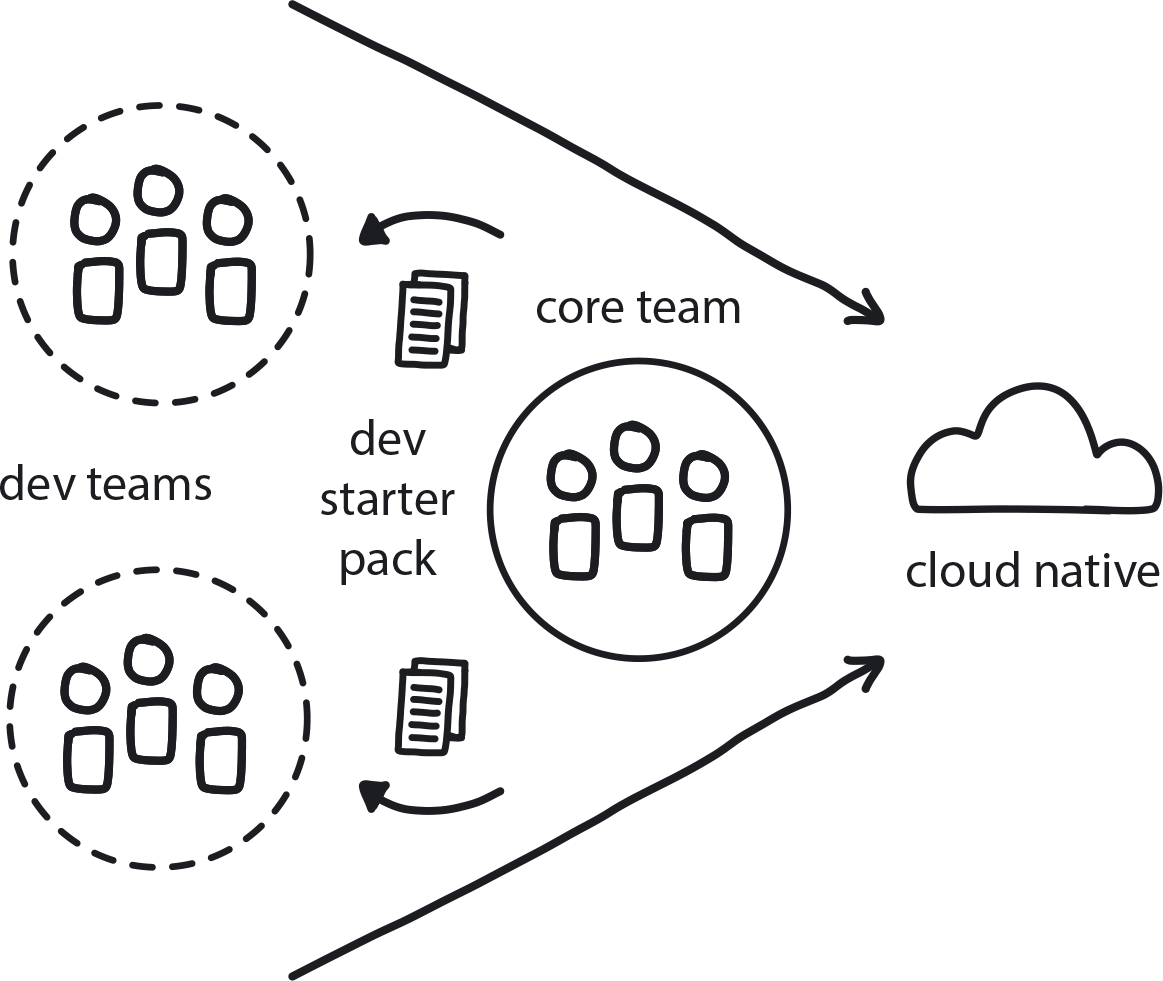
One to three months before the new platform goes live, begin training a couple of teams at a time, with a pause between each cohort to incorporate feedback and improve the process/materials.

The main goal of the Gradual Onboarding pattern is to start small organizational changes early in the cloud native transformation and keep them growing slowly and steadily. The teams working on the legacy software are slowly introduced to the new platform, given access to education, training materials, and experiences—described in the Dev Starter Pack pattern—and begin to build skills by working on practice projects as laid out in the Demo Applications pattern.

This whole process happens very gradually, as in only two to three teams at a time. This gives time for the Core Team to work with the new groups in depth and fully support them in completing their new cloud native knowledge and skills without rushing.

The creative team also takes breaks in between onboarding cohorts in order to use their feedback to improve the platform and onboarding materials. Remember Reference Architecture, from our earlier pattern? That is going to come in handy here, too.

**PATTERN: DEVELOPER STARTER PACK**



**Figure 12-8. Developer Starter Pack**

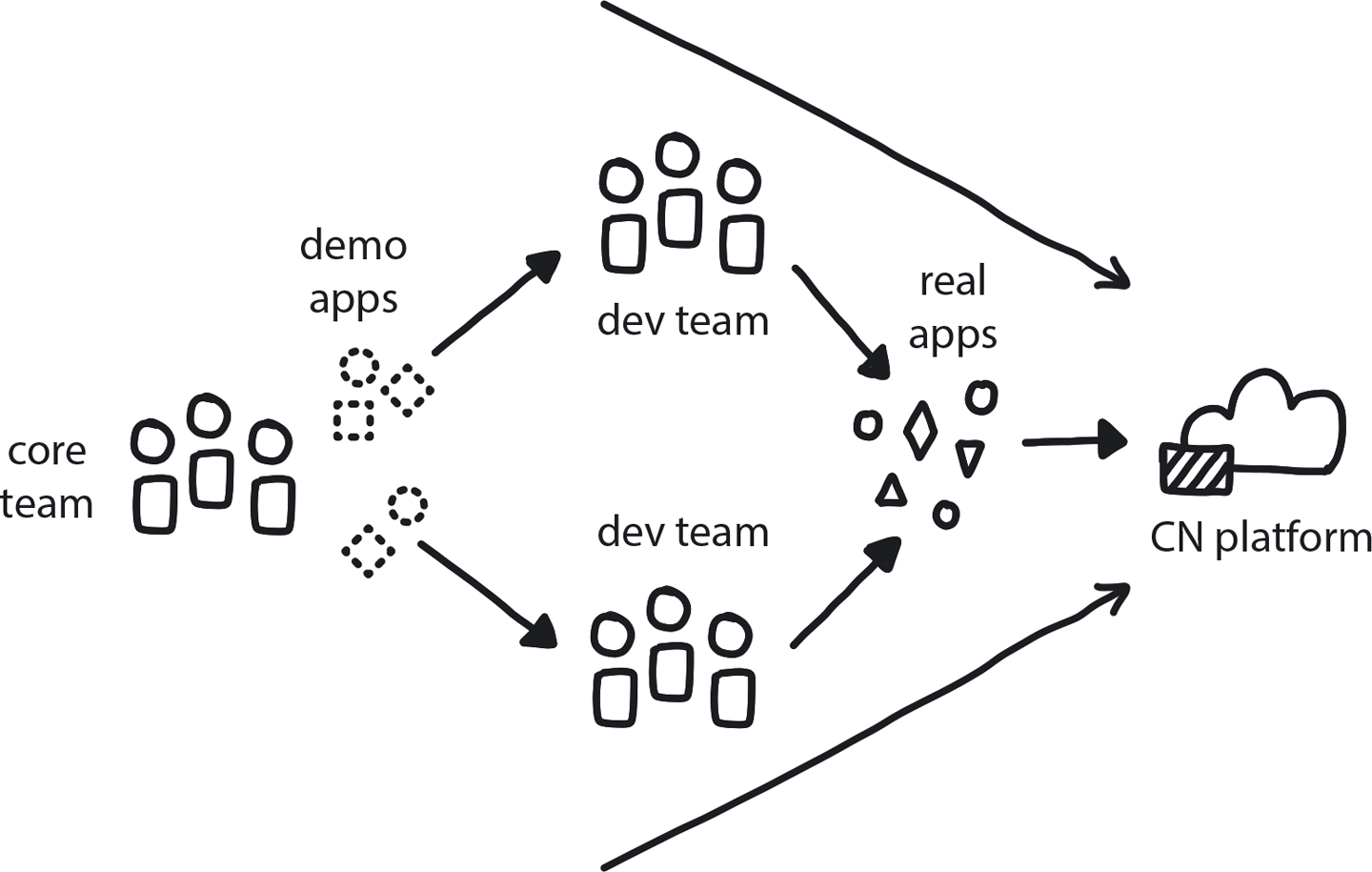
Provide a “starter kit” of materials, guides, and other resources to help new teams onboard to the new cloud native system quickly and with confidence.

Teams onboarding to cloud native don’t know the tools or technologies and are typically provided with very poor onboarding materials. Time is wasted while they try to figure things out on their own. Worse, if the onboarding resources are truly lacking, they might be forced to create their own cloud native practices. Inevitably these will be uninformed and unlikely to fit properly with the platform. Since each team will have to figure these out on their own, there will be variation in practices between teams.

Provide developers onboarding to cloud native everything they need to start working immediately.

This cloud native “starter kit” of materials should include tool configurations, version-control repository, CI/CD pipelines, example applications for practice, target platform description, trainings, and more. This Developer Starter Kit should include a Demo Application to practice on.

**PATTERN: DEMO APPLICATIONS**



**Figure 12-9. Demo Applications**

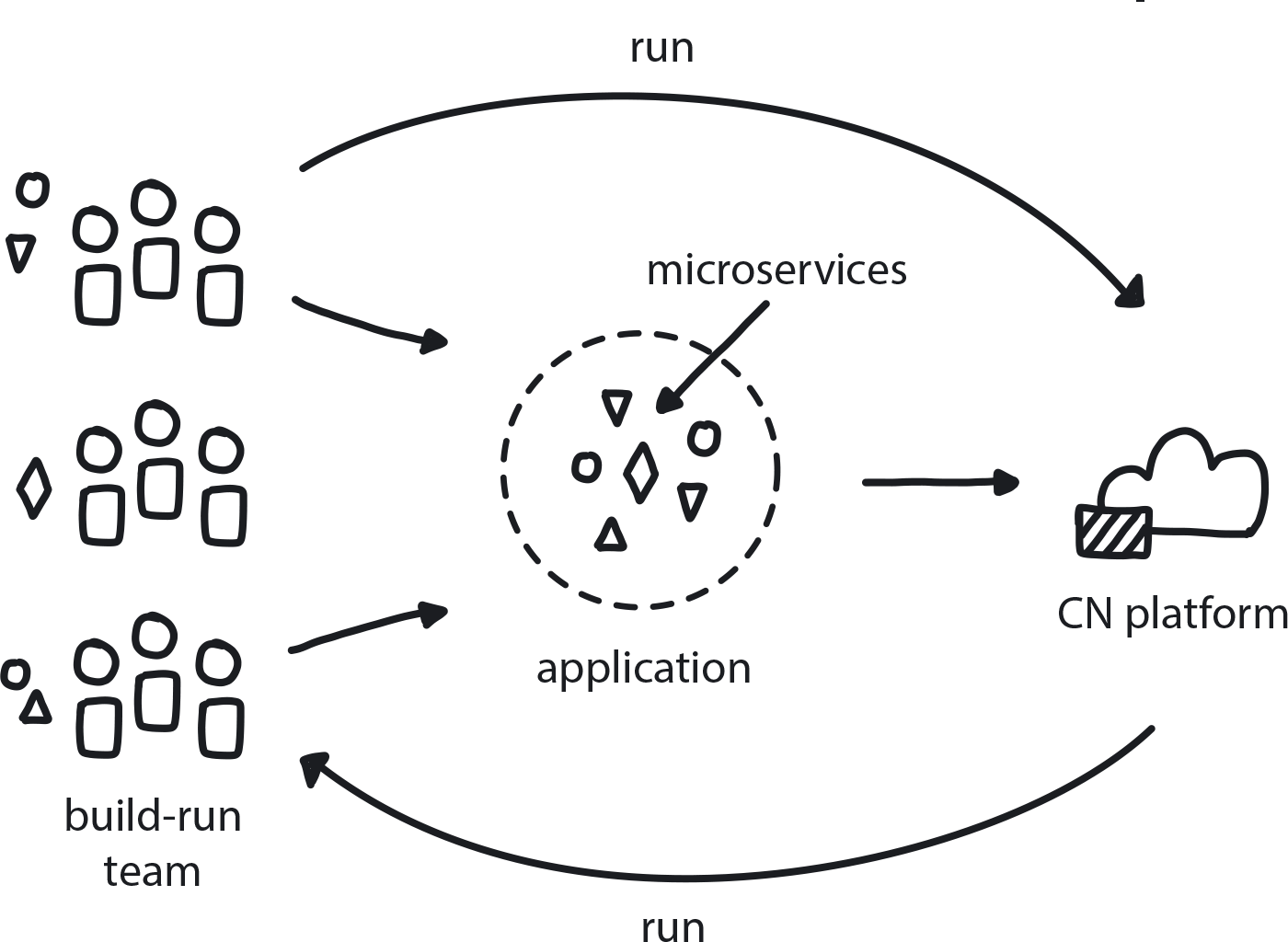
Teams onboarded to the new cloud native system receive demo applications as an educational starting point for building their own cloud native applications.

These are very helpful tools for making onboarding an easy and effective process. As mentioned earlier, though, a crucial part of this phase is also shifting the organizational culture from traditional to cloud native. This means that, even as you are training the teams in the new cloud native tools and techniques, you are also teaching them a new way of working. You’re refactoring them structurally and culturally to fit with these very different processes and methodologies, and even introducing new ways of thinking that may at first seem totally opposite to the world they worked in before.

“Gradual” is the important word here. Iterative education is the foundation of the Prepare for Onboarding pattern and progresses over time, from very general background exposure to cloud native concepts to advanced training. It is important that the education provided is appropriate to the stage of onboarding; if a team is not scheduled to be on the new platform for six more months, basic introduction to concepts like microservices, Docker containers, and Kubernetes is good. But they don’t need advanced training yet—save the hands-on learning for when they are actively moving onto the new system.

This gradual education process shapes and drives the essential cultural shift that developers must undergo as part of the move to cloud native: the transformation into Build-Run Teams.

**PATTERN: BUILD-RUN TEAMS**



**Figure 12-10. Build-Run Teams**

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

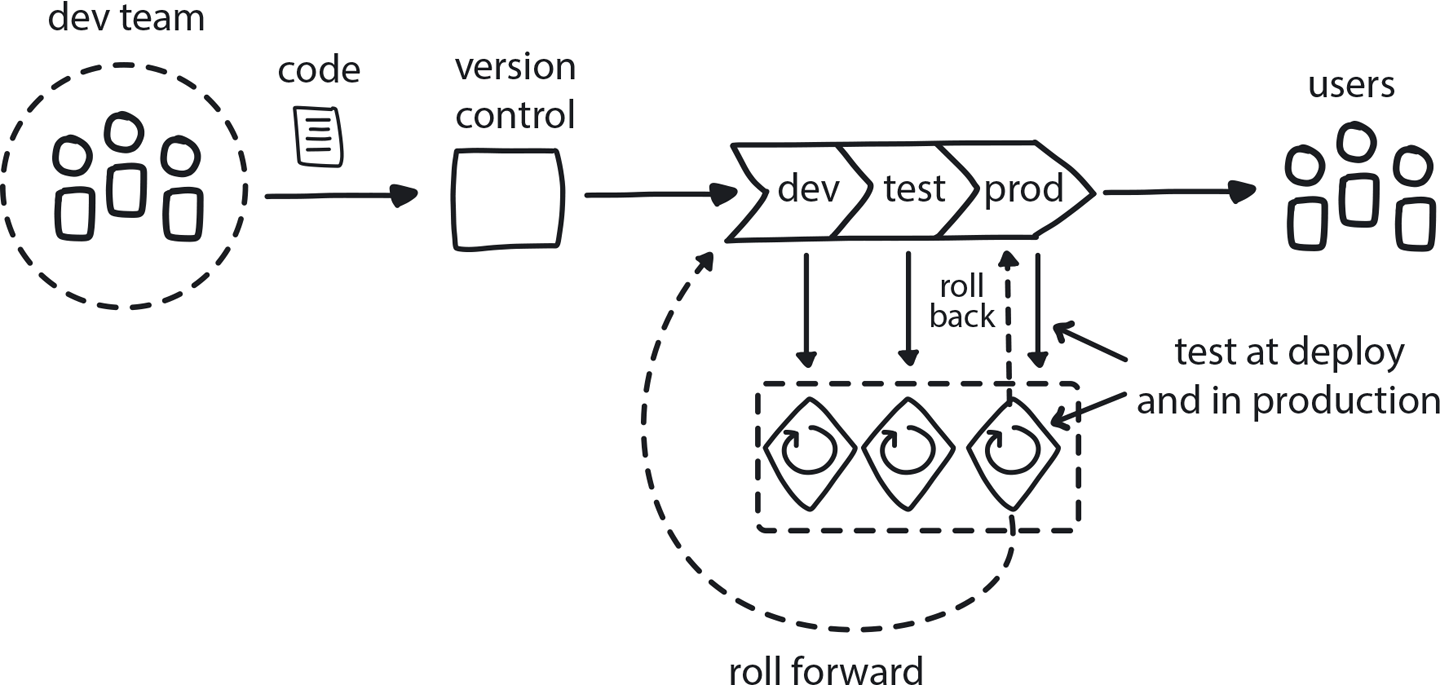
In traditional software development, developer teams rely on the Ops team to deploy artifacts to production. This handover kills production speed and agility. However, trying to fix that by putting an Ops person on each developer team is how you end up with 10 different irreconcilable platforms. Conway’s law shows us that software architecture will come to resemble the organization’s structure. So if we want the platform to be independent, then the teams developing an application need to be separate from the team running the production platform.

Most teams onboarding to a new cloud native system will be coming from a Waterfall approach, where specialist teams hand off artifacts by essentially tossing them over the wall to the next team. Agile invented cross-functional teams, removing the wall between developers and testers, but there is still a handoff to Ops for deployment. DevOps further tore down the walls dividing this process, but the model as it’s most commonly understood and implemented needs to evolve to align properly with cloud native development practices.

The Build-Run Team pattern is a specific formulation for a DevOps team that is optimized for cloud native. This team should be able to build (and test) everything needed to deploy their service to production and then support it afterward. Build-Run teams are the definition of “you build it, you run it” when it comes to developing cloud native applications. (But not the platform or infrastructure that those applications run upon.) Build-Run teams have full responsibility and control over the requirements for the microservice they are building. They don’t just design and deploy, but also participate in the operational side of whatever service or component they build.

In traditional organizations, provisioning hardware typically requires filling out a form, sending it to ops, and waiting for them to do it on your behalf. Each handover creates a delay of hours, days, even weeks, and this reduces velocity for the entire system. Needless to say, this approach is eradicated in cloud native Build-Run teams, so the Self-Service pattern is an essential partner to the Build-Run teams pattern. Self-Service emphasizes the full automation of any manual or semi-manual practices, so that developers can provision infrastructure and deploy their own applications without handing off to anyone else.

**SELF-SERVICE**

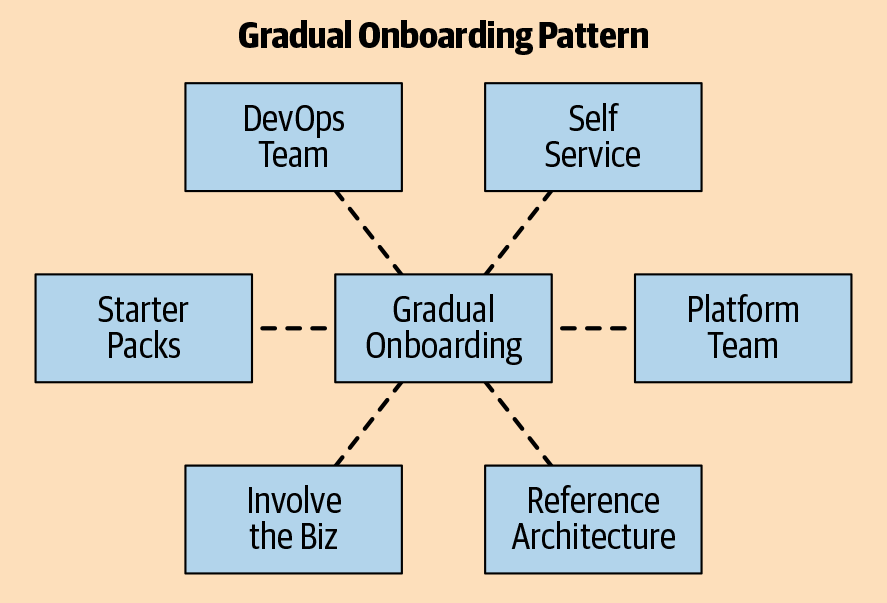


**Figure 12-11. Self-Service**

In cloud native everyone can do their own provisioning and deployment, with no handoffs between teams.

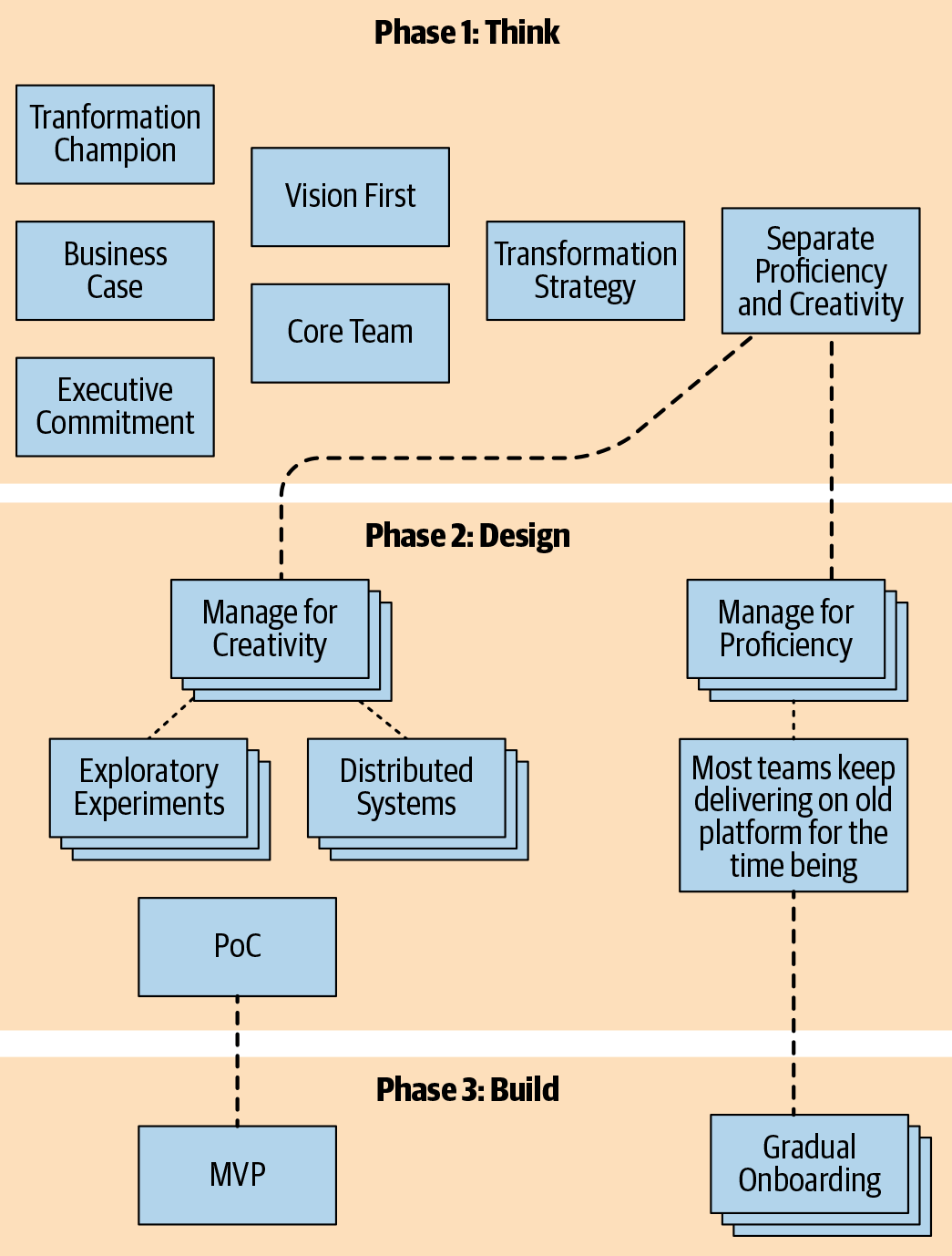
The ability to self-service provisioning and deployment also encourages frequent releases from developers to production. Without it there is no Continuous Delivery, and no Learning Loop from delivering to customers and then incorporating user feedback to create improvements.

Self-Service is the final piece in the metamorphosis of teams from traditional software development processes into cloud native ninjas. [Figure 12-12](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch12.html#gradual_onboarding_super_pattern) shows the patterns that go into this crucial Gradual Onboarding stage of the transformation.



**Figure 12-12. Gradual Onboarding super pattern and the patterns that apply as part of it**

And this brings us to the end of Phase 3: Build. We have our functional MVP being extended into a fully production-ready platform by the Platform Team, while the Core Team takes care of the Prepare for Onboarding activities and then, when the time is right, begins the Gradual Onboarding process. [Figure 12-13](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch12.html#the_transformation_journey_thus) shows our progress thus far, and opens the door to Production Readiness and entering Phase 4: Run.



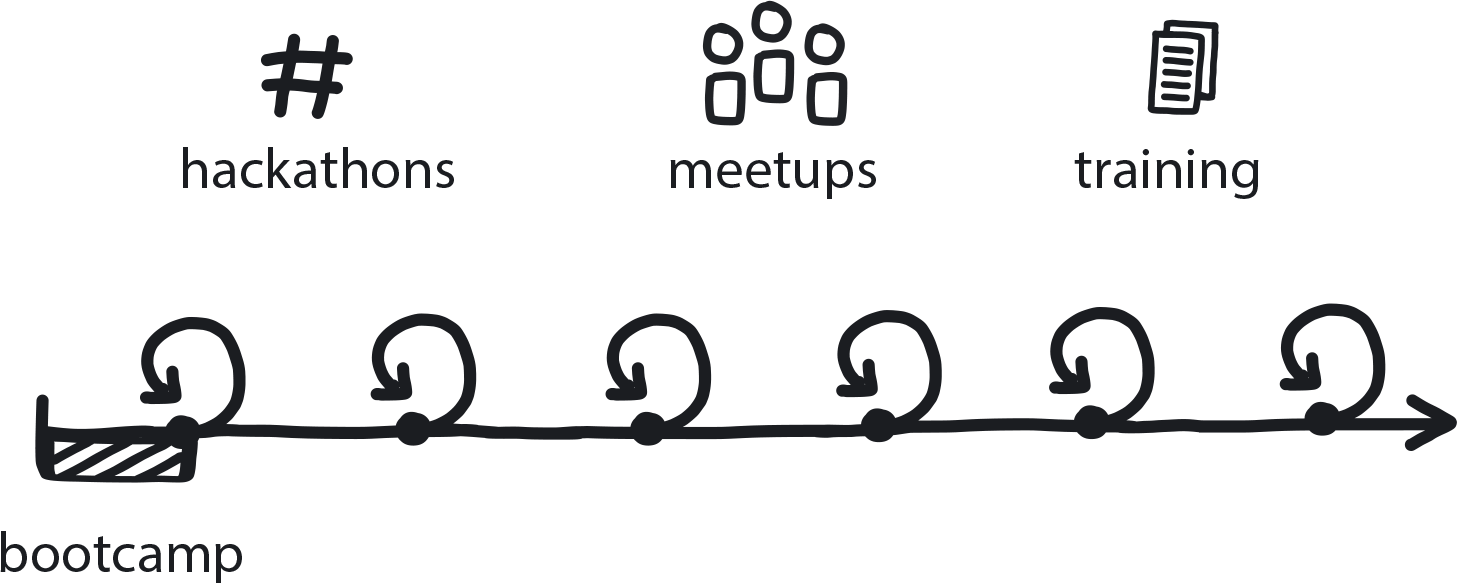
**Figure 12-13. The transformation journey thus far—phases 1 through 3, shown in patterns**

**PHASE 4: RUN**

The platform is completed and ready for production. Onboarding is in full swing. What happens now?

Well, here is one pattern that we have already applied, back during pre-boarding, but that is a prime directive going forward as well: Ongoing Education.

**PATTERN: ONGOING EDUCATION**



**Figure 12-14. Ongoing Education**

Continuously introduce new ways and improve existing ones to help teams continually develop their cloud native knowledge and skills.

Earlier in the transformation teams were given general education in cloud native topics, then more advanced training as they learned how to work hands-on in the new system. Once successfully onboarded, though, education doesn’t stop!

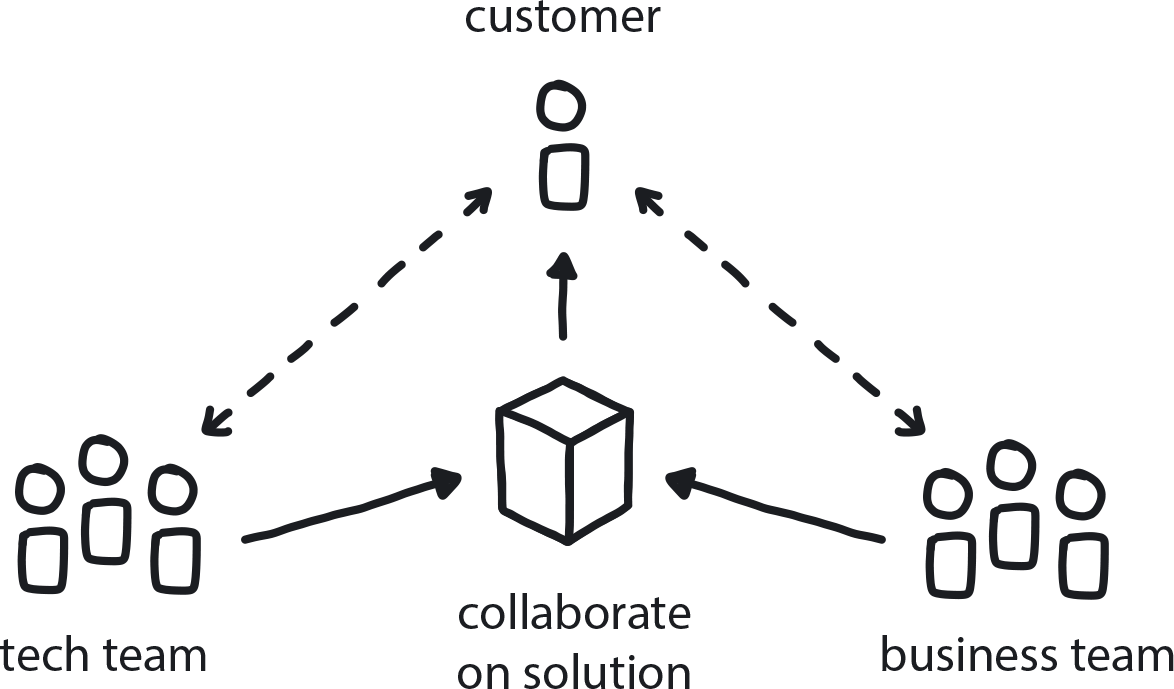
Unfortunately, continual learning often falls to the side—along with creativity and innovation—as teams get comfortable in their new way of working and the company steers its major focus back to proficient delivery. Companies are used to the way things worked in the past, when they bought a big new tool: everyone took two days of massive training and a couple weeks on a learning curve to adapt to it. After that, everyone knew the tool and could forget about education.

Cloud native never rests, however. Ongoing Education assures your teams are always learning, building upon the core of what is to constantly improve, update, and iterate forward.

As the teams delve into their Developer Starter Kits and practice with Demo Applications, they are building confidence in their work on the new platform. One way to keep a positive upward spiral going is to Involve the Business and ensure that Feedback Loops are in place.

Both of these patterns address the challenge of incorporating business needs into the company’s new way of working. When developers are running quick iterations without involving their customer-facing colleagues, the features they come up with could be limited to tech solutions only. Businesspeople, however, can’t run full tech experiments to investigate ideas they might have for adding customer value to the company’s core products/services. To close the learning loop, experiments and changes need to include everyone, from developer to customer and back. The results can then be used to define and drive the next change.

**PATTERN: INVOLVE THE BUSINESS**

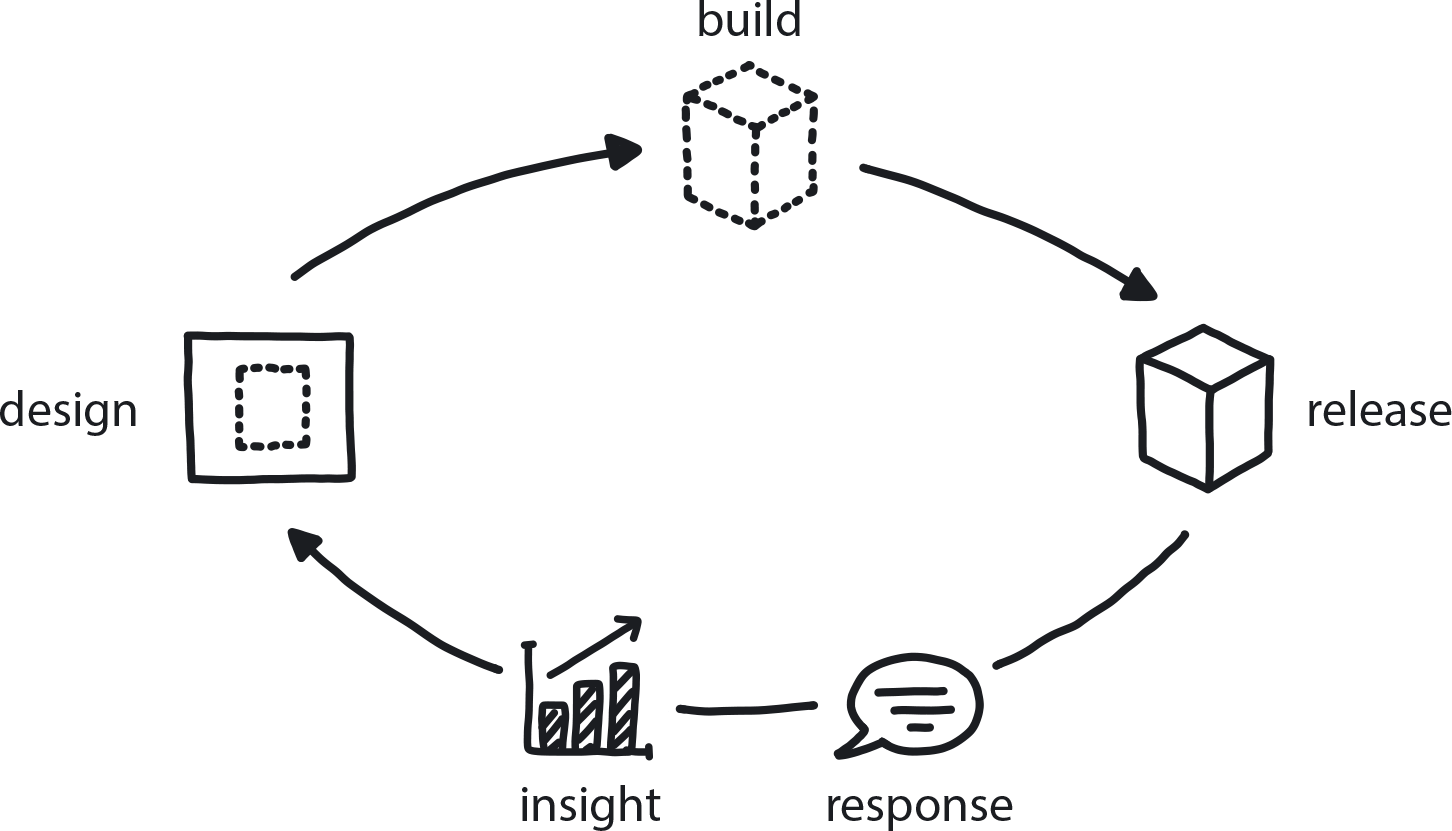


**Figure 12-15. Involve the Business**

Both the business teams and the tech teams within the company need to collaborate to create an effective customer feedback loop that drives product improvement.

Doing this requires close collaboration between dev teams and people from the business side to define experiments for testing new customer value and quickly executing them. The Involve the Business pattern describes practical ways to make sure this happens. Dev teams need to embed measurement of customer feedback into products so the business teams gets insight, and the business teams in turn need to work with devs to define what is most valuable to deliver. The practical way to incorporate this insight into future features or improvements to existing ones is the Learning Loop pattern.

**PATTERN: LEARNING LOOP**



**Figure 12-16. Learning Loop**

Building feedback collection into the delivery process closes the loop between engineers and the people who use their products, putting the customer at the center of the product development cycle.

**Strangle All the Old Things**

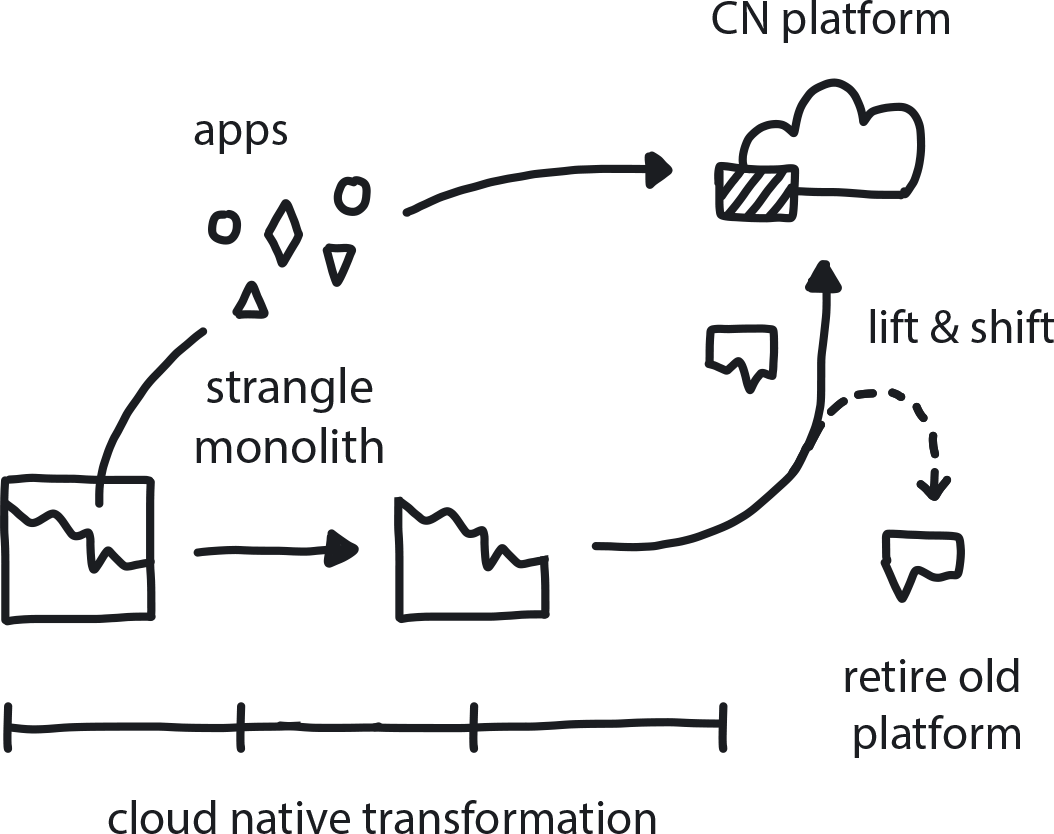
Things are going pretty well! The new cloud native platform is up and running, and most teams are trained, or well into the process of training, to deliver on it. It’s time to migrate the legacy business applications. (Won’t the proficient teams left behind to babysit the old system be happy to hear this!)

Re-architecting a large monolith that’s been around, and slowly but inexorably growing for years and years, is a massive project. Some companies try to do it all at once, by simply trying to re-create the functionalities in a brand new codebase on the brand new platform. This makes a certain amount of intuitive sense, because a clean slate is very appealing, but rewriting a large monolithic application from scratch also carries a large amount of risk.

You cannot start using the new system until it is developed and functioning as expected, but of course our company has little cloud native experience or knowledge to actually pull this off. As we saw in the story of WealthGrid, building a new system from scratch will take a year or longer, and during this time there will be minimal enhancements or new features delivered on the current platform, so the business risks losing market share.

WealthGrid, and everyone else, should never attempt to re-create their monolith on the cloud, for all kinds of reasons. The solution is to instead slowly dismantle it. This avoids the significant risks embedded in the greenfield approach of rewriting the entire application all at once, since you re-create functionalities step by step while appropriately re-architecting them (and your culture) for the cloud. This way, the business value of new functionality is achieved much more quickly, and a cloud native architecture of loosely coupled services means future refactoring work will be simple.

**PATTERN: STRANGLE MONOLITHIC APPLICATION**

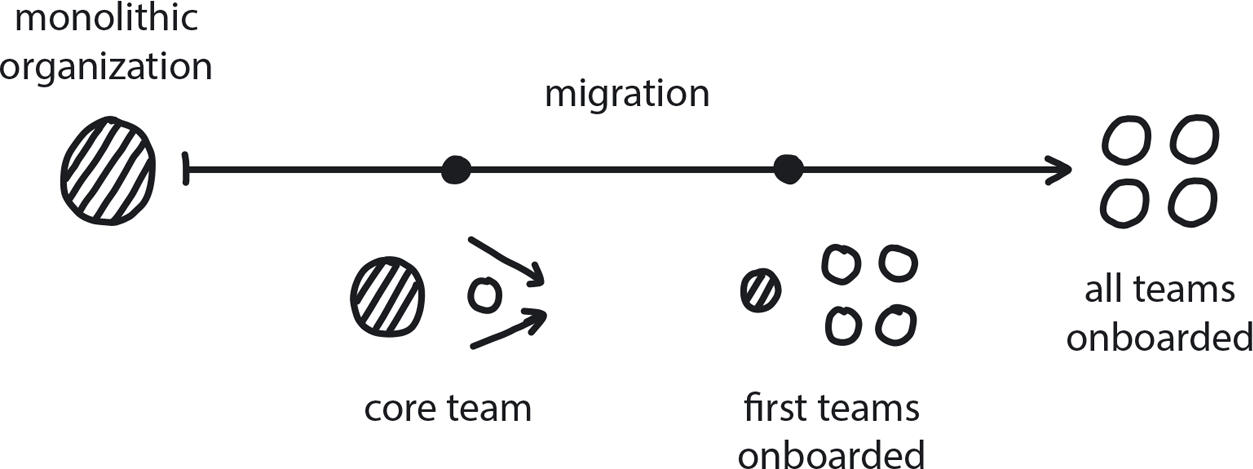


**Figure 12-17. Strangle Monolithic Application**

Gradually split pieces of the old monolithic application one by one, re-architect them into services, and move them over time to the new cloud native platform.

Naturally, the Strangle Monolithic Organization pattern complements Strangle Monolithic Applications.

**PATTERN: STRANGLE MONOLITHIC ORGANIZATION**



**Figure 12-18. Strangle Monolithic Organization**

Just as the new tools, technologies, and infrastructure gradually roll out over the course of a transformation initiative, the organization and its teams must also evolve to work with them properly.

Migrating an existing company to the cloud can take years if done the right way, which is slowly and gradually. Instead, though, many companies commonly make the unfortunate choice to move everyone over at once. It’s hard to do good training in that scenario; without a solid background in the new tech, teams will be slow to become productive on the new system. There will be huge pressure on the Core Team to help solve all their problems. Or, instead of seeking help from the overloaded support team, some people will attempt to hack their own best-guess solutions or fall back on old habits that don’t work well in cloud native.

This is what happens when the cultural and organizational changes necessary in a cloud native transformation are not supported to evolve equally along with the technology. We can simply avoid creating the problem in the first place by following the patterns from the pre-onboarding and Gradual Onboarding phases.

**DON’T CREATE “SECOND CLASS” ENGINEERS**

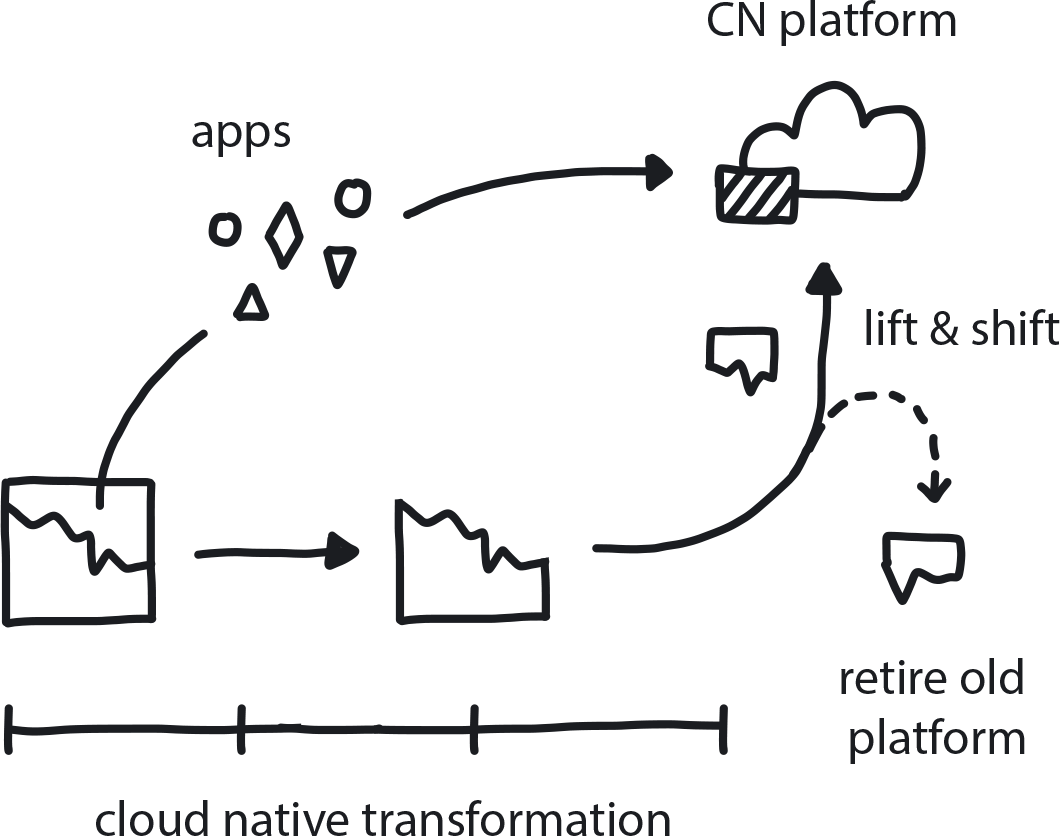
There is a second bad move that is less common, but we still see it often enough to make note of the phenomenon. This happens if a company has a significant legacy codebase that is going to remain relatively intact. It will be ported over to and deployed from the cloud but otherwise remain unchanged. This legacy core will require engineers to maintain and look after it.

In this case, companies create a dual culture by simply leaving one part of the organization in legacy while everyone else moves to cloud native. This divides the company into a second-class citizen scenario, where the teams who will never get moved to cloud native are simply stalled professionally. They’ll never get to play with the cool new tech and build modern developer skills if they are left behind to babysit the legacy system. This, naturally, leads to frustration, resentment, and reduced motivation, not to mention difficulty retaining engineers.

These earlier patterns lay the groundwork for new team structures and ways of working before the teams are actively trying to deliver on the new platform. With these in place, the natural next step is then to slowly strangle the process and cultural relics like specialized teams, along with the monolith itself.

The two systems, old and new, can coexist very well during the transformation process. Once the migration is complete, any remaining pieces of the legacy system can be ported over to the new cloud native platform using the Lift & Shift at the End pattern, and gradually refactored into microservices until the old monolith is no more. Then the legacy system maintainers—the very last of the proficiency teams—are the final cohort to be onboarded to the new system.

**PATTERN: LIFT & SHIFT AT THE END**



**Figure 12-19. Lift and Shift at the End**

It’s important not to approach a cloud native transformation by simply attempting a full “lift and shift” of your existing system onto the cloud. But it can be smart to move some intact pieces of it at the very end.

Lift and shift is a bad move if done at the beginning of a transformation, but at the very end it can actually be a very good move. Sometimes, after everything else has been moved over, we find that there is still a functional piece of the original codebase, or even more than one, that would be prohibitively difficult to refactor to fit the cloud native platform. If this is a strong system that is functioning, stable, and requires few changes, just keep it. Package it so that it sits on one server and works forever, and create the new pieces needed to talk to the old piece. This could be a one or more microservices that essentially serve as an interface to the old system by talking directly to the old database or, even better, to the old APIs.

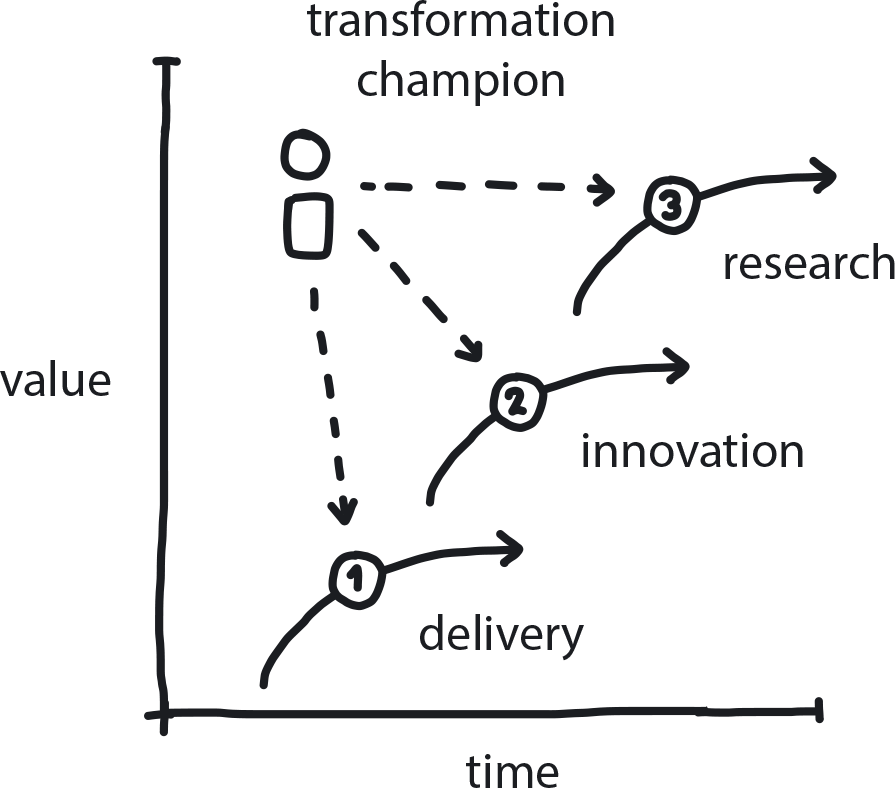
**Keeping Creativity Alive**

The new system is fully up and running, and all the teams are working well using their new-found skills to deliver new features and improvements quickly and iteratively. This is actually a dangerous time due to the temptation to fall back into full proficiency mode. By which we mean focusing almost exclusively on delivering your product or service, while slowly but inevitably forgetting about innovation and creativity.

At this stage the Core Team has successfully delivered a new system and helped everyone onboard onto it. They are the ideal candidates to stay at the forefront of innovation efforts, to be in charge of keeping creativity alive in the company. How does this work in the real world?

In [Chapter 6](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch06.html#tools_for_understanding_and_using), Tools for Understanding and Applying Cloud Native Patterns, we introduced the Three Horizons model for balancing proficient delivery with ongoing innovation. It’s so important to cloud native that it becomes a pattern in its own right.

**PATTERN:THREE HORIZONS**

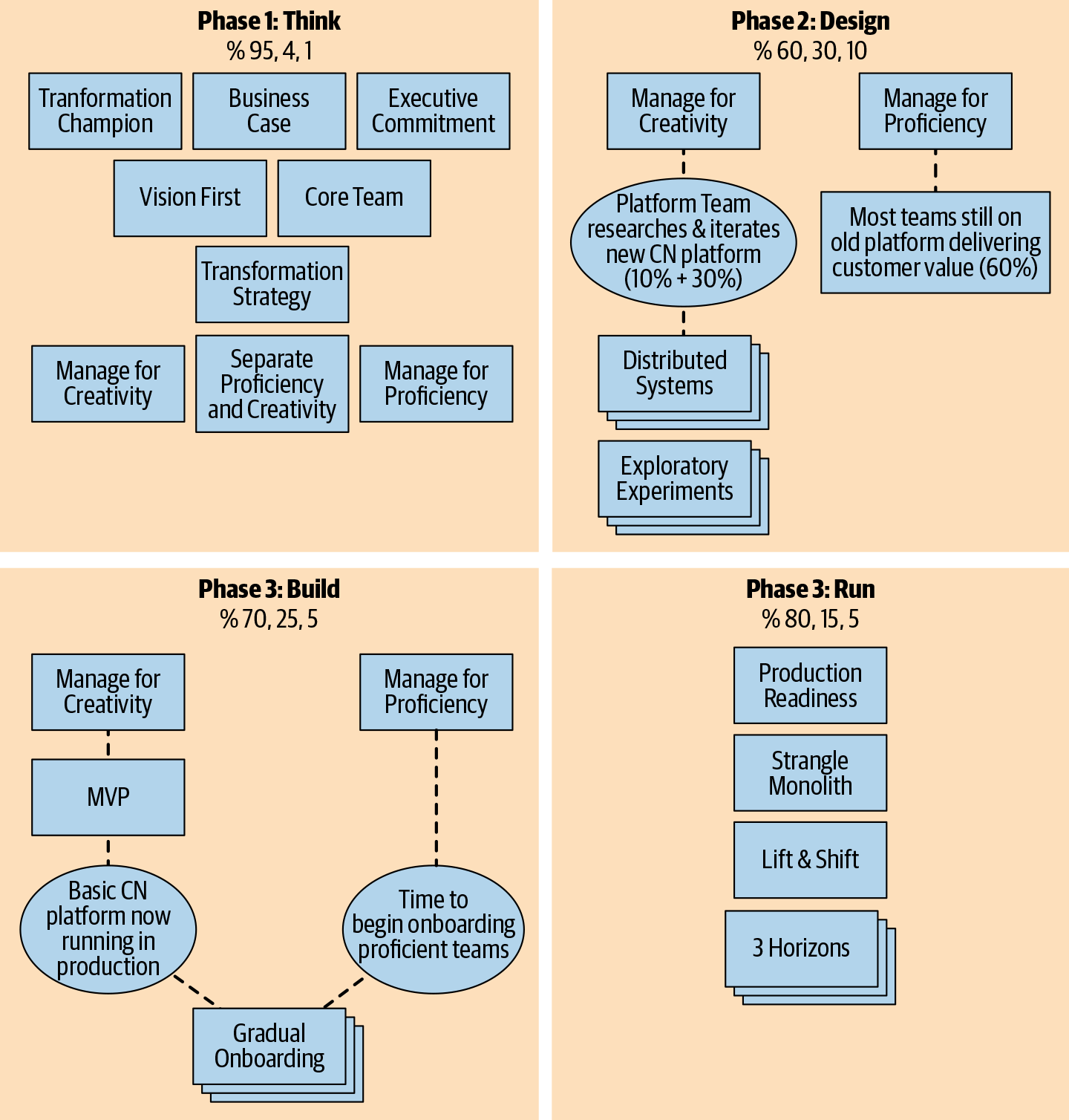


**Figure 12-20. Three Horizons**

Dynamically distributing resources between delivery, innovation, and research makes an organization responsive to change while reliably delivering core business value.

The Core Team now invests their well-practiced creative abilities to H2 and H3, innovation and pure research. They work on identifying and trying out next-generation technologies and methodologies that might interest the company a few years down the road. Everyone else, the majority of teams working in the new system, do indeed stay focused on delivery and so remain in H1—most of the time.

The beauty of the Three Horizons model is its ability to rebalance responsively when circumstances change. The distribution of investment between delivery, innovation, and research can simply be shifted as needed. [Figure 12-21](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch12.html#the_transformation_design) shows how this dynamic distribution changes through the course of a cloud native transformation initiative to reflect varying levels of investment in creative innovation and research, versus proficient delivery.



**Figure 12-21. The transformation design, as illustrated in patterns, shows how each phase recalibrates the distribution of investment between delivery, innovation, and research.**

This varying relationship is demonstrated throughout our transformation design. Before beginning, we had a company that has been putting probably close to 100 percent of its focus into delivery and only a token amount, if anything, into innovation or research for quite some time. It had functionally forgotten how to be creative. WealthGrid is our fictional example, but this accurately describes a great many real-world companies, too. So, right now the numbers invested in delivery, innovation, and research would be essentially 100, zero, and zero, respectively.

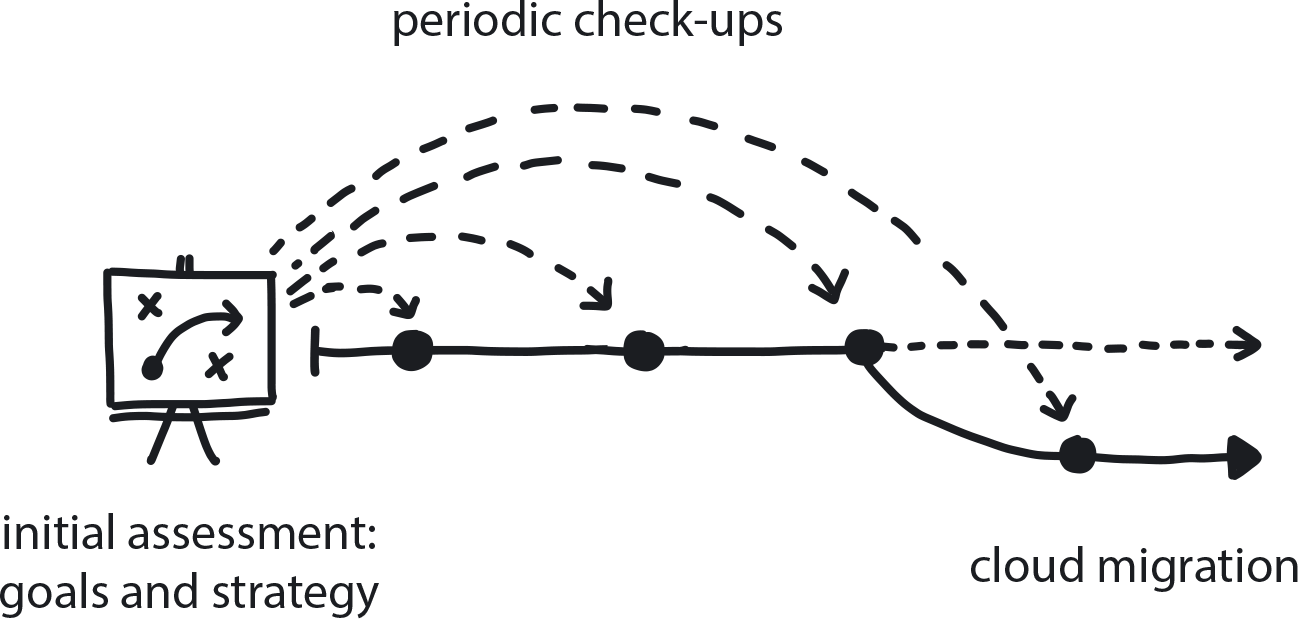
Early in the transformation we begin applying Phase 1 “Think” patterns to investigate the Business Case and gain Executive Commitment for a cloud native initiative, along with other establishing patterns. The numbers shift only a small amount to 95/4/1[**1**](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch12.html#ch12fn1) during this stage because most of the organization is focused on delivering value on the old system while the company leaders and then a small Core Team begins planning and designing the transformation.

As the transformation moves forward, however, more investment flows into the initiative, and people from all across the organization start to become involved. At the height of the transformation, Phase 2,“Design” and Phase 3, “Build,” a larger proportion of the company is involved in H2, innovation mode, because people are being trained how to not just work differently but also think differently as they move to the cloud native system. Phase 2 also has a relatively high H3 number—60/30/10—because so much research and experimentation are being invested in identifying the best migration path. This is an innovative time across the board because the company is dedicating itself to undergoing intentional change, meaning the focus has shifted, very temporarily, from pure delivery.

Once the MVP is production ready, things begin to rebalance once again in the direction of a higher percentage in H1/proficiency. Most of the teams return to steady delivery mode, but a respectable amount of attention is being paid to innovation and research. The H2 efforts are directed toward realistic improvements that could become money-making (or -saving) projects in the next year or so. They are practical explorations of all kinds of things to continuously improve the future of the products, such as maybe adopting serverless, or building A/B testing. H3 is mostly pure research, maintaining awareness of things that may be several years out in terms of technological readiness but that might be valuable eventually.

Right around this 80/15/5 distribution is the sweet spot for cloud native companies to strive for: most of the focus (80%) is on delivering the products or services that drive the company’s profits, but a respectable amount of investment still goes into innovation (15%) and pure research (5%). Keeping an eye on the balance is key, and for this we apply the Periodic Checkups pattern to maintain situational awareness and use Dynamic Strategy to adjust the ratio between proficient delivery versus investment in innovation and research as needed.

**PATTERN: PERIODIC CHECKUPS**

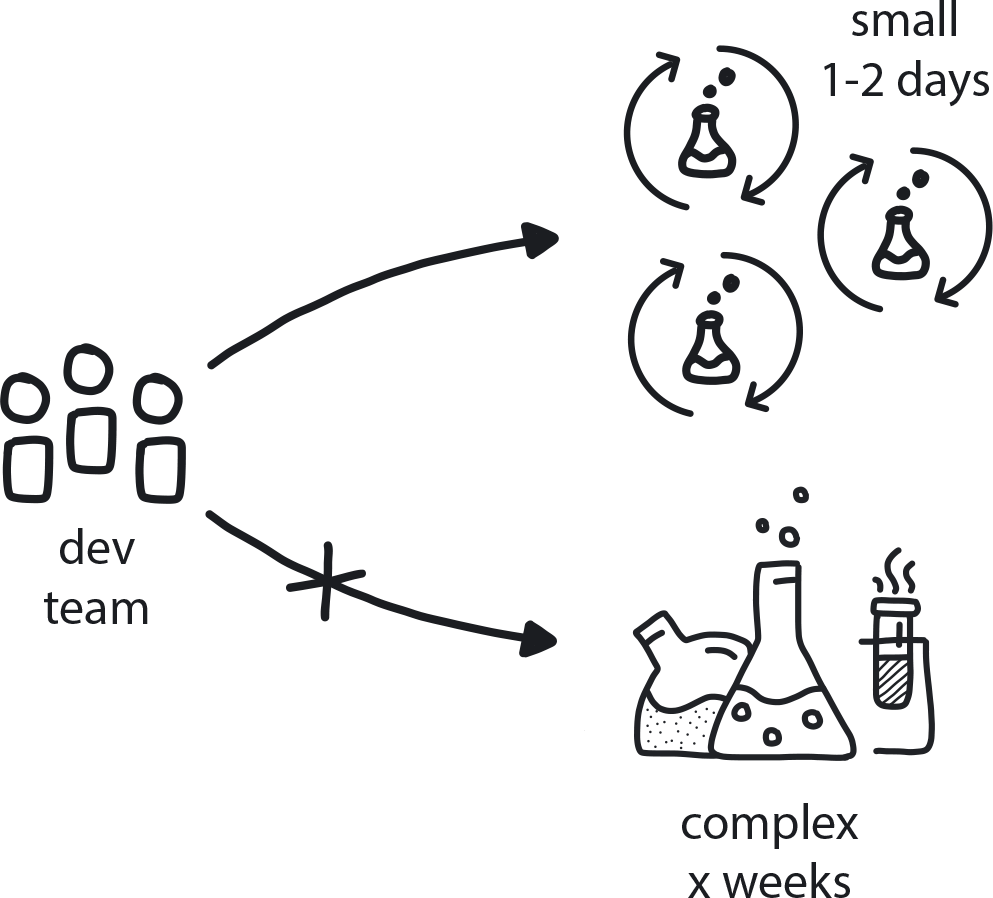


**Figure 12-22. Periodic Checkups**

Frequently reassess vision and objectives to ensure these remain the correct direction to proceed as the business environment shifts.

There are other patterns for fostering creativity to apply at this point, if they have not already emerged organically during the transformation process. Either way, these carry forward in the life of the company, even after the transformation is complete. They are additional supports for creativity and keeping innovation alive across an entire organization, even as the new system swings into mainly proficient operation mode. Two of these patterns are Reduce Cost of Experimentation and Personalized Relationships for Co-Creation.

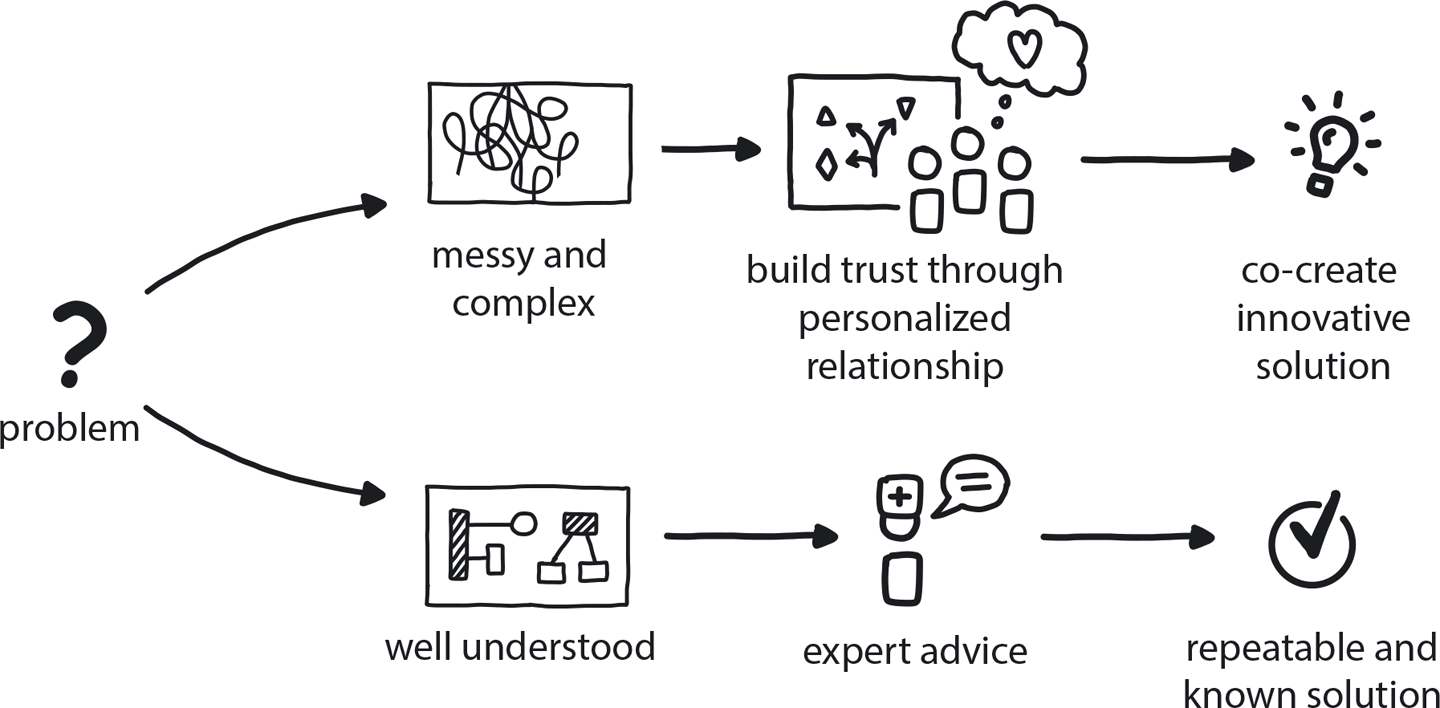
**PATTERN: REDUCE COST OF EXPERIMENTATION**



**Figure 12-23. Reduce Cost of Experimentation**

When someone has an idea that requires validation, the costs of doing experiments around it need to be as low as possible.

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



**Figure 12-24. Personalized Relationships for Co-Creation**

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

**The End?**

All the teams are onboarded onto the new system and working in a cloud native way—Build-Run Teams doing CI/CD and other cloud native methods to rapidly develop new features and deliver quick, incremental improvements. There is a focus on delivery, but also meaningful investment in ongoing innovation, with teams dedicated to more pure creativity. Everything is working well, and everyone is feeling confident. It is time to shut down the old data centers. Congratulations, you have reached the end of your transformation journey!

Definitely pause for a well-earned celebration and to appreciate how far you have come. But also please be aware that the end is only the beginning. You have finished the transformation, but this is merely a milestone in a longer journey. The cloud native road goes on.

And here is where we let you in on the secret at last. Cloud native transformation is never just about microservices and Core Teams and Kubernetes, or even about evolving from Waterfall/Agile. The true metamorphosis was from being an organization that rarely faced change, and that probably even feared it, to an adaptive organization that can change direction as needed, quickly and confidently.

You’ve got the tools now, and we don’t mean Kubernetes. We mean you can now use this same framework of patterns to do anything you like. Whether you just want to create a new product or embrace whatever new tech comes next, you’ll still just apply your new process: Start with a Business Case, define a Core Team, experiment your way to an MVP, and so on to completion.

You are ready for the world of cloud native but—even better—you are ready for whatever comes after that!

[**1**](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch12.html#ch12fn1-marker) These numbers are hypothetical for this fictional example, but based on our observation of how client companies and other organizations the authors have encountered have allocated their resources. Ratios will of course vary from org to org.