**Chapter 10. Patterns for Infrastructure and Cloud**

Once upon a time (and not very long ago), infrastructure meant one thing: provisioning your own on-premise hardware. Cloud computing has radically expanded this definition. Today, enterprise market infrastructure can mean a public or private cloud, on- or off premises, utilizing virtualization, containerization, or serverless (functions as a service) computing—or any combination.

By comparison, legacy infrastructure has become akin to a physical anchor slowing you down and stopping your enterprise from changing. Buying servers, housing them, and configuring, provisioning, and maintaining them takes a great deal of time, attention, and money. For some companies there are still a few advantages to owning your own servers, but most are better off embracing the cloud. All the somewhat abstract changes you are making to your strategy, organization, culture, and processes to deliver software in a fast, responsive way, all the flexibility you are gaining, must be supported on a physical level by appropriately nimble infrastructure.

The entire point of going cloud native is gaining the ability to adjust course quickly. If you want to create stability in an increasingly uncertain environment, the best way to get it is to be able to respond to change quickly. That sounds like an oxymoron, but hear us out. Knowing that it will take you a long time to respond if a disruptor comes along in your market creates constant anxiety. But if you know you can react to meet a threat in a matter of months, it doesn’t matter what the threat might look like—you are confident you can simply handle whatever comes. So the ability to stay steady in a fluctuating environment will create more stability, in the way that gyroscopes spin rapidly around a steady center.

It used to be companies could make a big, painful semi-risky move to new tech and then enjoy nice, stable quiet time for a decade, even two, before having to do it again. Those days are gone. The first iPhone was introduced in 2007; think about the things we take for granted now that were basically inconceivable just over ten years ago. Think about all the human experiences that have been forever altered: communicating, navigating, shopping, taking a taxi. Innovation only keeps accelerating, and we can see that change is happening blindingly fast. We can’t see what the change will be, but we can be ready for it.

Software tools inevitably change. So does infrastructure. The important thing is no longer where you build software, but how. In some ways, infrastructure is the most easily solvable of all the challenges embedded in a cloud native transition, and we have patterns to help you do just that. There are also unexpected pitfalls in the cloud to watch out for. Building custom tools instead of using what’s already available. Major vendor commitments, significant contracts with suppliers: all of these things can be anchors as heavy as that on-premise data center, still blocking you from moving fast, just in a different way. We’ve got patterns to help you choose the right infrastructure, and avoid these pitfalls, too.

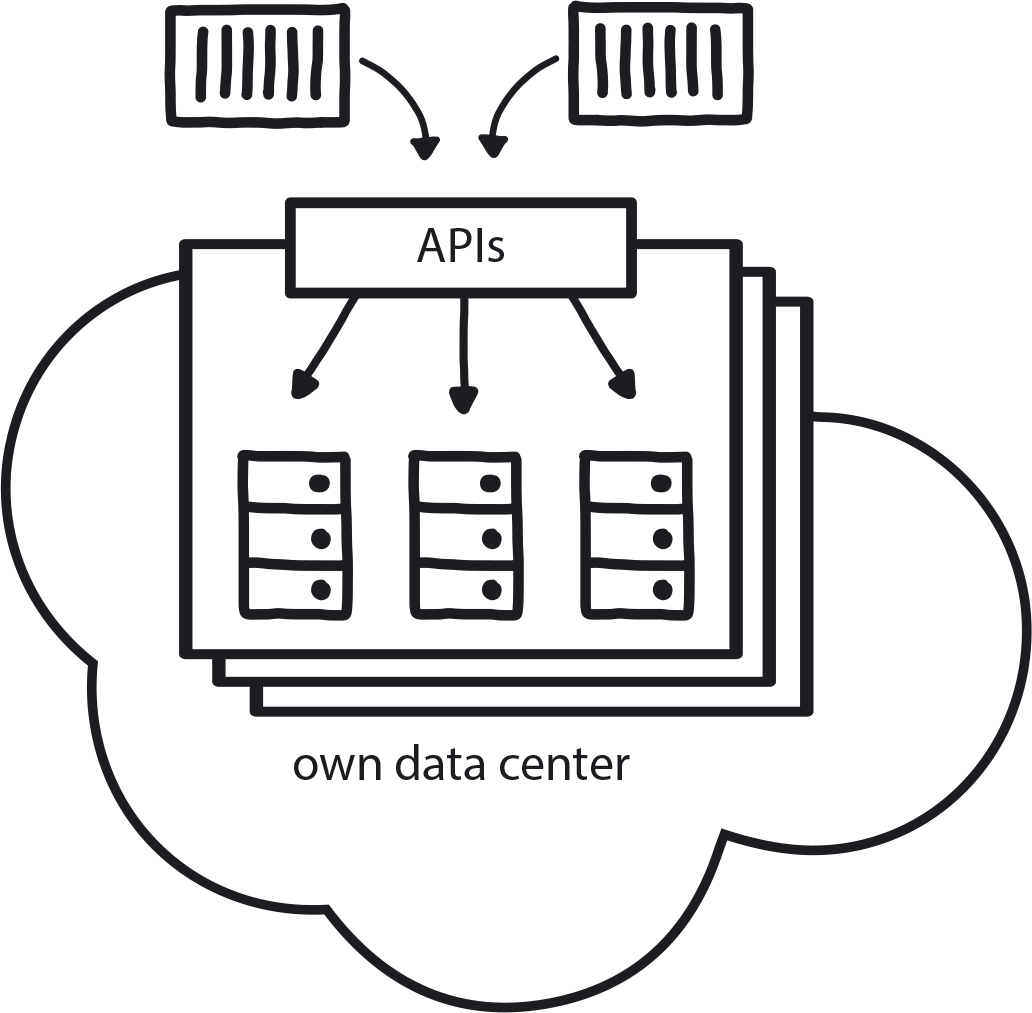
*The following patterns describe and address cloud native development and process. They are presented in an order that we believe will be useful or helpful for the reader, but there is no right (or wrong) order for approaching them: Patterns are building blocks for a design and can be combined in different ways according to context.*

*This chapter is intended as an introduction to the patterns themselves, and there is intentionally little explanation relating them to each other at this point. When considering an individual pattern, the decision is not just where and when to apply it, but whether to apply it at all—not every pattern is going to apply in every transformation or organization. Once the concepts are introduced we will fit them together, in progressive order and in context, in*[*Chapter 11*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch11.html#applying_the_patterns)*and*[*Chapter 12*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch12.html#applying_the_patterns_a_cloud_native)*as a design that demonstrates how patterns are applied in a typical cloud native transformation.*

* [Private Cloud](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#private_cloud)
* [Public Cloud](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#public_cloud)
* [Automated Infrastructure](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#automated_infrastructure)
* [Self-Service](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#self_service)
* [Dynamic Scheduling](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#dynamic_scheduling)
* [Containerized Apps](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#containerized_apps)
* [Observability](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#observability)
* [Continuous Delivery](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#continuous_delivery)
* [Continuous Deployment](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#continuous_deployment)
* [Full Production Readiness](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#full_production_readiness)
* [Risk-Reducing Deployment Strategies](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#risk_reducing_deployment_strategies)
* [Lift and Shift at the End](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#lift_and_shift_at_the_end)

**Pattern: Private Cloud**

*A private cloud approach, operated either over the internet or on company-owned on-premises infrastructure, can offer the benefits of cloud computing services like AWS while restricting access to only select users (*[*Figure 10-1*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#private_cloud01)*).*



**Figure 10-1. Private Cloud**

A company in a highly regulated industry that is, by law, not allowed to use public cloud or a company in any sector that has recently made a major investment in new on-premises infrastructure. In the Private Cloud context, services are provisioned over private—though not necessarily on-premises—infrastructure for the dedicated use of a single organization and typically managed through internal resources.

**In This Context**

**Connecting designated hardware to specific apps reduces their mobility and resilience. Building any kind of manual intervention by the Ops team into an application’s life cycle will create long delays in deployment and reduce quality.**

* There is a lot of fear around data on the public cloud.
* Public cloud vendors are very knowledgeable about, and experienced in, running large infrastructure.
* In much of the world, public cloud is allowed for almost all industries.
* A private cloud is under full control of the company, so you can optimize in different ways and customize different things that are not possible on public clouds.
* Proficient support is required to keep private clouds up and running, so you must have all that knowledge in your (typically limited) team.

**Therefore**

**Decouple the setup of your physical infrastructure from the provisioning required for the application itself.**

Treat all the servers and the rest of the infrastructure as a single large machine managed fully automatically through a set of APIs. Aim to use the same set of tools as would be required on at least one public cloud to ease any future migration.

* Fully automate everything related to running the application.
* Treat your on-premises infrastructure like a cloud; the hardware needs to be completely abstracted away from the users.
* Set up physical servers, network, storage, etc., all separately.
* Deploy and maintain applications through full APIs.
* Even if you need dedicated hardware, treat it as a part of the cloud so it is fully automated.

**Consequently**

**The company enjoys the benefits of cloud native architecture with the additional control and security—but also expense—of hosting your own private cloud setup. Applications are running on the private cloud in exactly the same way as they would run on a public cloud.**

* + Developers can get infrastructure on their own, and delivery is faster.
* + There is full control over the platform and data.
* − Private clouds carry a high cost of ownership and quickly become outdated.

**Common Pitfalls**

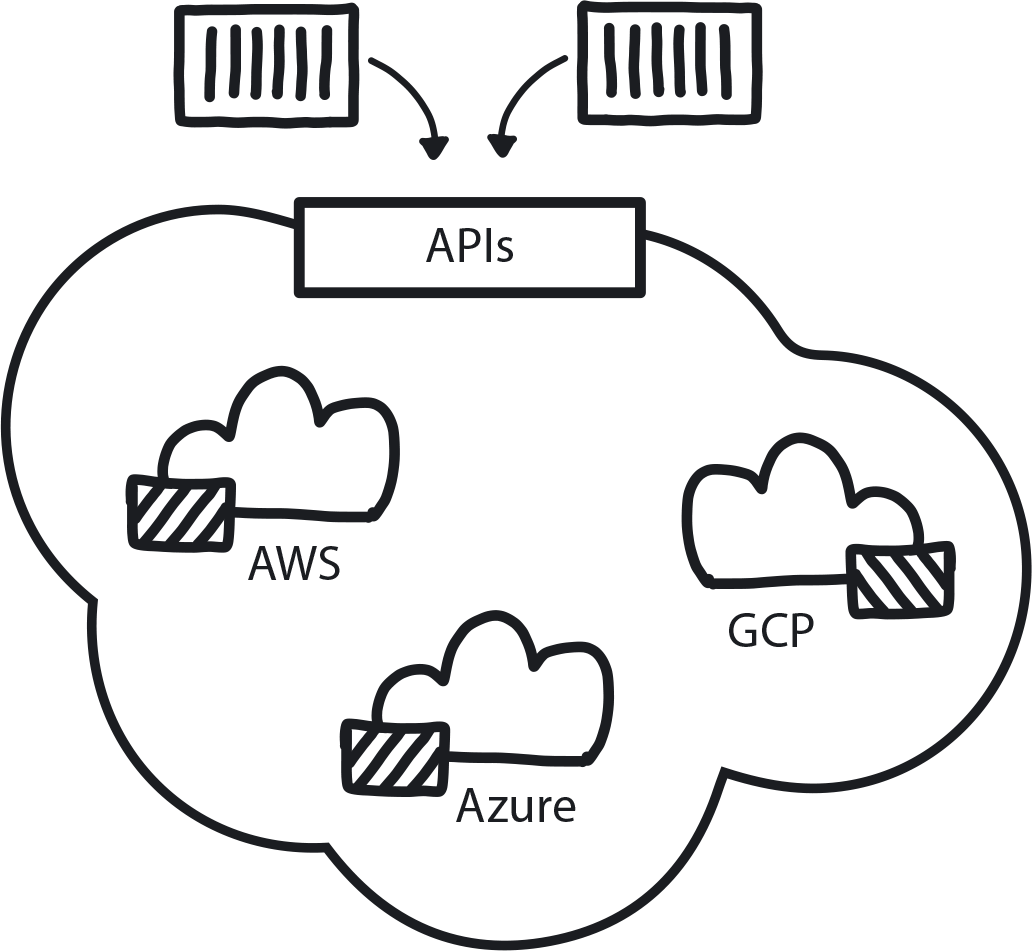
If the private cloud is on premises, just like the old hardware was, it does not get treated as a true cloud platform. People try to build the new system using the techniques they always used on the previous on-premises infrastructure.

**Related Patterns**

* [Public Cloud](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#public_cloud)
* [Automated Infrastructure](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#automated_infrastructure)
* [Communicate Through APIs](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#communicate_through_apis)
* [Distributed Systems](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#distributed_systems)
* [Avoid Reinventing the Wheel](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#avoid_reinventing_the_wheel)

**Pattern: Public Cloud**

*Instead of using your own hardware, rely on the hardware managed by public cloud vendors whenever possible (*[*Figure 10-2*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#public_cloud01)*).*



**Figure 10-2. Public Cloud**

You are moving to microservices, you are continuously growing your codebase, and teams now require automation for Continuous Delivery. The amount of manual work is going down, but the cost of maintaining hardware is going up.

**In This Context**

**Procuring, installing, and maintaining hardware becomes a bottleneck that slows down the entire organization.**

For most businesses, infrastructure and hardware are not their core business, so they do not invest heavily in maintaining and improving them. Public cloud vendors, however, invest a great deal of money and talent to optimize their services.

Furthermore, when you own the hardware, costs are typically higher: you need to overprovision for peak consumption. If Black Friday traffic is 10 times higher, then you must maintain that 10-times capacity the other 364 days of the year—there is no elastic capacity expansion.

* Private clouds quickly become outdated because they are not a high priority for the business.
* Never enough resources to make a private cloud as good as public cloud.
* In some business areas regulations may not allow use of public cloud due to security concerns and explicit data regulation.

**Therefore**

**Hand over the management of hardware and rent capacity from public cloud vendors like Amazon, Microsoft, Google, and similar instead of owning, managing, and creating full automation for infrastructure.**

* Rely on full automation.
* Use APIs to connect with public cloud.

**Consequently**

**You rent fully automated, scalable, resilient infrastructure from a public cloud provider that can be increased and decreased on demand. You pay only for the resources you are actually using.**

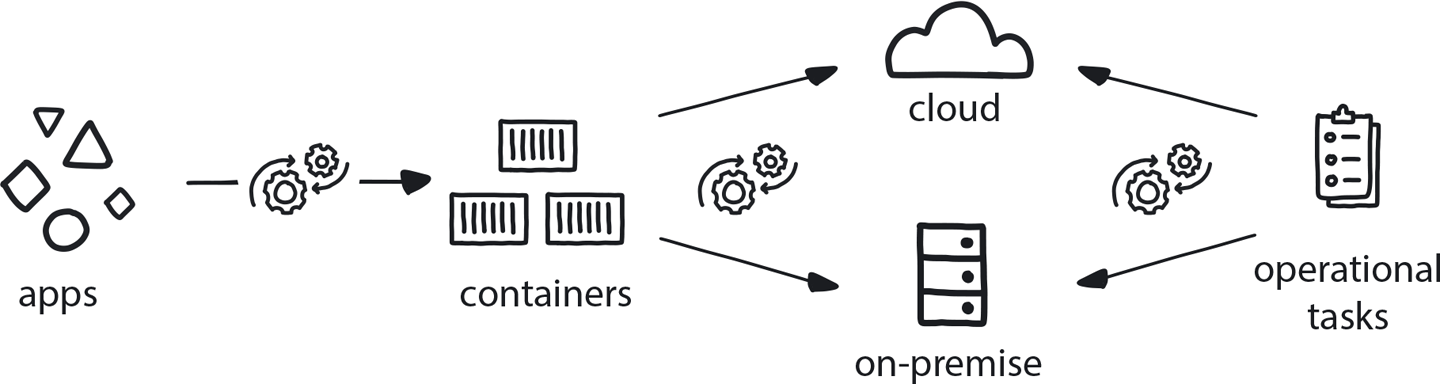
* Take advantage of integrated services.
* Public cloud vendors constantly upgrading to latest software and services.
* Built-in services like maintained databases, machine learning frameworks, and all kinds of SaaS continuously being built by the vendors.

**Related Patterns**

* [Private Cloud](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#private_cloud)
* [Avoid Reinventing the Wheel](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#avoid_reinventing_the_wheel)
* [Exit Strategy Over Vendor Lock-in](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch07.html#exit_strategy_over_vendor_lock_in)
* [Serverless](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#serverless)
* [Automated Infrastructure](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#automated_infrastructure)
* [Dynamic Scheduling](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#dynamic_scheduling)

**Pattern: Automated Infrastructure**

*The absolute majority of operational tasks need to be automated. Automation reduces interteam dependencies, which allows faster experimentation and leads in turn to faster development (*[*Figure 10-3*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#automated_infrastructure01)*).*



**Figure 10-3. Automated Infrastructure**

A company is moving to cloud native and adopting cloud native patterns such as Microservices Architecture, Continuous Delivery, and others. Teams are independent and require fast support services from the Platform Team. Most of the operational tasks are performed on demand by the Ops team.

**In This Context**

**Manual or semi-automatic provisioning of infrastructure leads to dependencies among the teams and to long waiting times for results, hindering experimentation and slowing development.**

* Traditional operational teams don’t have sufficient levels of automation and, due to high workload, no time to learn new technologies.
* Public clouds provide full automation of infrastructure resources.
* Manual requests and handover between development and operations teams is very slow.
* Number of operations engineers in manual systems must scale up proportionally to growth in infrastructure demands.
* Experimentation and research take longer and require more resources due to involvement of an already-busy operations department.

**Therefore**

**Dedicate at least 50% of the Ops team’s time to automating the operational tasks, and eliminate all manual infrastructure provisioning and maintenance tasks.**

Any manual work that is required in between the changes committed by the developer and the delivery to production will significantly reduce the speed of delivery and introduce interteam dependencies.

* Treat infrastructure automation scripts with equal importance as the rest of the company codebase.
* Automate, fully and completely: compute, storage, networking, and other resources; patching and upgrading of operating systems; and deployment and maintenance of systems running on top of the infrastructure.

**Consequently**

**Developers spend less time waiting for infrastructure resources and can conduct quick experiments and scale running systems rapidly and easily.**

* + Ops team spending significantly less time on repetitive support tasks and investing more time and resources in ongoing improvement of the system.
* + Full automation will allow the provisioning of exponentially more resources per member of operational staff.

**Common Pitfalls**

Understanding that everything needs to be automated, but deciding to “just do it later.” It is difficult to improve something not built right in the first place. Your team gets locked into doing manual steps, it’s a waste, but there is no time to improve it because they need to keep doing these manual steps to keep everything running.

**Related Biases**

*Status quo bias*

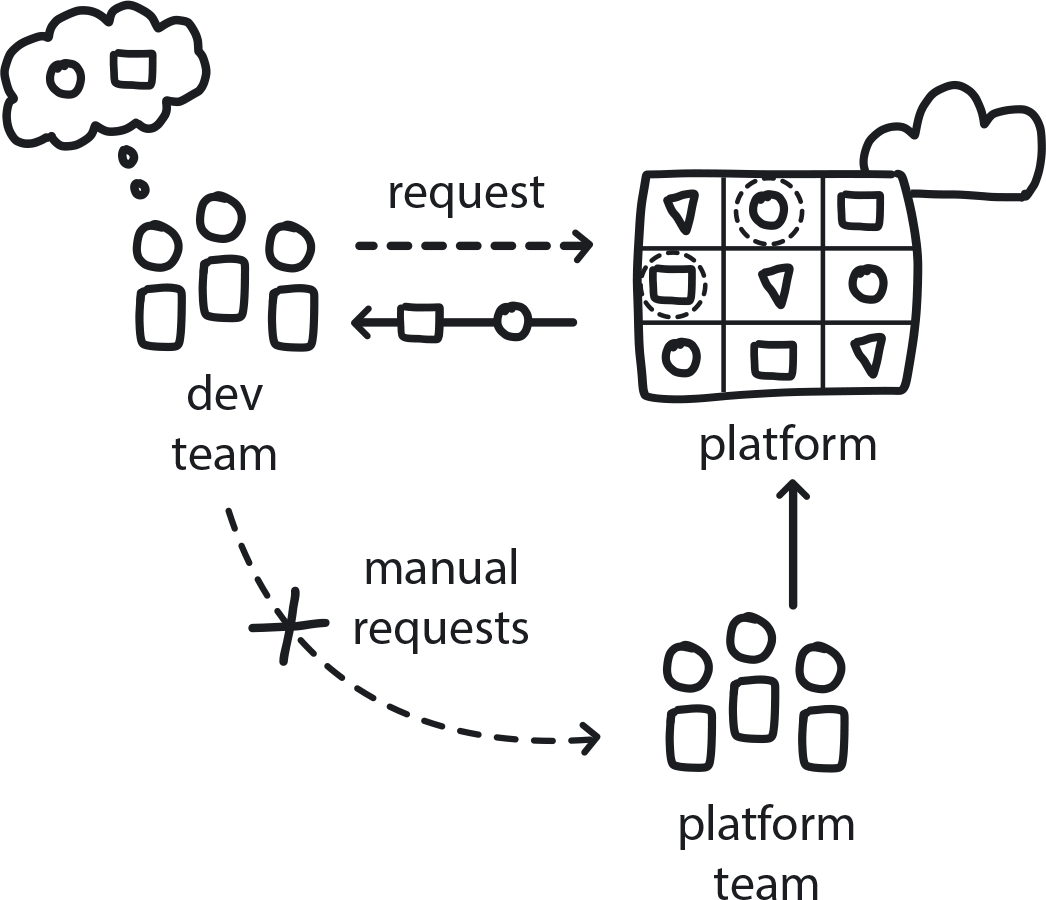
We’ve always done manual provisioning, so let’s just keep doing things that way. It might not be optimal, but we know it works (and changing something that works feels risky).

**Related Patterns**

* [Dynamic Scheduling](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#dynamic_scheduling)
* [Platform Team](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#platform_team)
* [Automated Testing](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#automated_testing)
* [Public Cloud](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#public_cloud)
* [Private Cloud](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#private_cloud)
* [Self-Service](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#self_service)

**Pattern: Self-Service**

*In cloud native everyone can do their own provisioning and deployment with no handoffs between teams (*[*Figure 10-4*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#self_service01)*).*



**Figure 10-4. Self-Service**

The company is moving from Waterfall/Agile and, within a structure of separate operations and development teams, is aiming to set up microservices, CD, and public (or private) cloud. Teams are running many experiments and PoCs and aiming to reduce the cost of experimentation.

**In This Context**

**In traditional organizations, provisioning hardware or doing maintenance work 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 slows down the entire system. This process also discourages frequent releases from developers to production, and without that there is no Continuous Delivery (CD), no learning loop from delivering to customers and receiving feedback.**

* Anything manual is, by definition, too slow for cloud native.
* People will do more of something if they can do it themselves.
* Premature optimization—automating the wrong thing—can be particularly bad in this context.

**Therefore**

**Across the company, everything related to software development should be self-service: everyone should be able to provision infrastructure or deploy their applications on their own without handoff to another team.**

Self-Service is the requirement—Automate Everything is the way you achieve that.

* Create full automation around the platform. All manual or semi-manual processes must be fully automated.
* Create UIs and APIs so people can include this in their automation.
* Grant access to all developers (within reasonable security limits).
* Start during the MVP stage: Prerequisite is basically having the entire cloud native toolset.

**Consequently**

**Whenever anyone needs something done, they can do it on their own without handing off to Ops.**

* + Reduced cost of experimentation because there is less waiting for results.
* + Reduced dependencies.
* − Functional self-service needs a much higher-quality interface, which is a much higher cost.
* − The system is automated to be bulletproof against wrong behavior by non-experts using it, which is valuable but also has cost. If you have experts working on infra, they don’t need foolproof systems—but most people aren’t experts.

**Common Pitfalls**

Automating things too early before it’s fully understood what is going to happen, which leads to the biggest pitfall we see in this context: self-service for things you don’t actually need to do. Aim for automation in the long term but don’t take immediate action too early—you can automate the wrong things, which is a huge waste of time and money.

Automating self service on the legacy system. In many cases this is too difficult and too expensive and takes the focus away from the new infrastructure. It’s often better to keep using manual procedures if they are well-rehearsed and easy to execute.

**Related Biases**

*Pro-innovation bias*

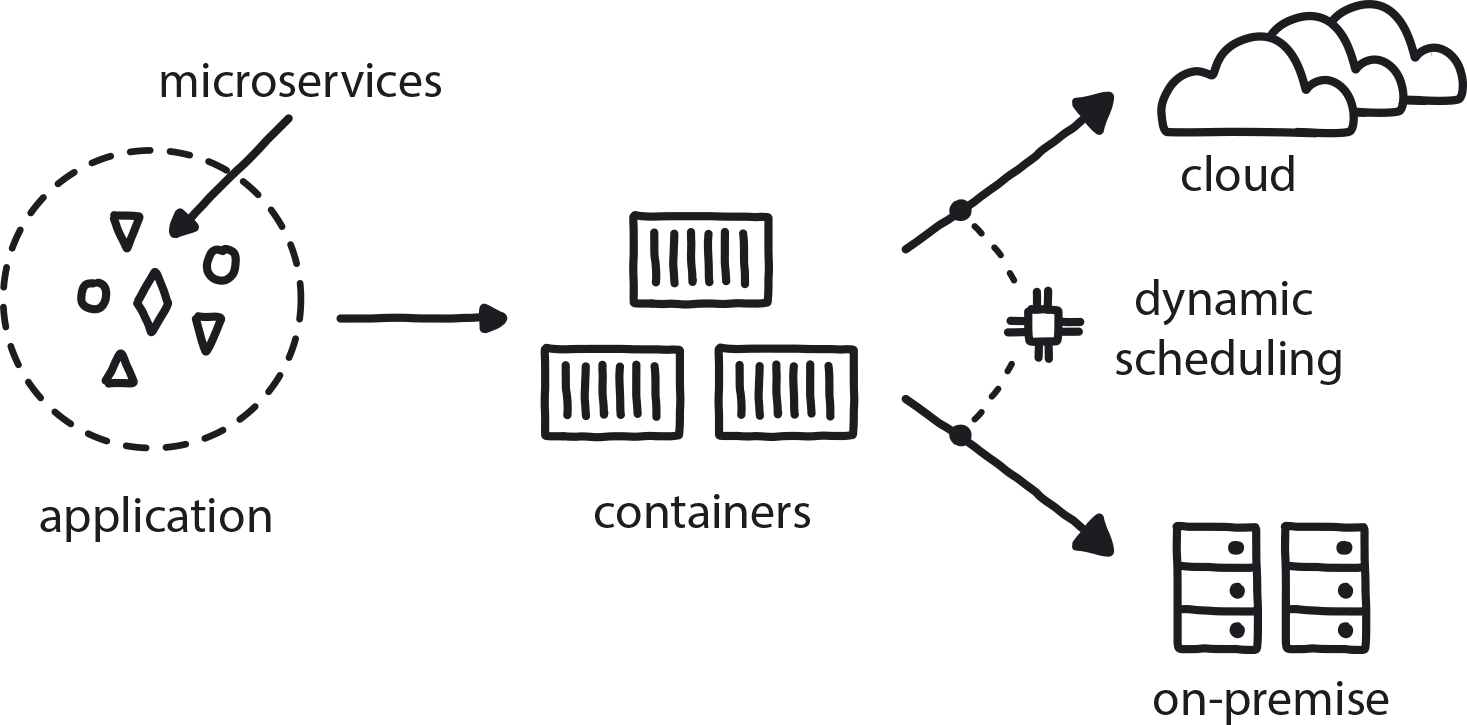
“Automation is good, so let’s automate all the things!” Yes, automation is good, but automating the wrong things is not good.

**Related Patterns**

* [Delayed Automation](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#delayed_automation)
* [MVP](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#mvp_platform)
* [Observability](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#observability)
* [Dynamic Scheduling](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#dynamic_scheduling)
* [Build-Run Teams](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#build_run_teams_devops)
* [Automated Testing](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#automated_testing)

**Pattern: Dynamic Scheduling**

*An orchestrator (typically Kubernetes) is needed to organize the deployment and management of microservices in a distributed container-based application to assign them across random machines at the instant of execution (*[*Figure 10-5*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#dynamic_scheduling01)*).*



**Figure 10-5. Dynamic Scheduling**

An application has dozens of independent microservices, and the development teams want to deploy each of them multiple times a day on a variety of private and public cloud platforms.

**In This Context**

**The traditional hardware approach assumes “This app runs on this server”—which is not practical in a distributed system running on the cloud. Attaching specific microservices to specific pieces of hardware significantly compromises the stability and resilience of the system and leads to poor use of the hardware. Every time you want to improve something you need someone to understand how it is all related, so they can safely make the change.**

**The market demands that companies deliver value to clients very quickly, in a matter of hours or even minutes. However, the traditional deployment of applications to specified static servers using manual or semi-automatic procedures cannot support the growing demands of the development teams to deploy each component separately on multiple environments once, or even more times, per day.**

* Software systems become more distributed overall and are required to run on many platforms.
* Dynamic scheduling tools (i.e. container orchestrators like Kubernetes, Nomad, and Docker Swarm, among others) are becoming mature and available for general use.
* Small pieces of applications can fail at random times.
* Advanced technology companies deploy thousands of times a day to large and varied array of development, testing, and production environments.
* In a system with hundreds of components, designating where each one runs increases complexity in the code because this must be specified in the build and is impractical.
* Predicting how a distributed system will behave in runtime is practically impossible.
* All major public clouds have managed dynamic scheduling as a service.

**Therefore**

**All application scheduling needs to be done using an orchestration system in a fully automatic way.**

Dynamic schedulers function as container management, in the form of a platform for automating deployment, scaling, and operations of application containers across clusters of hosts. This helps to achieve much more efficient hardware use as the scheduler understands the latest state of the system and can squeeze many components to the same piece of hardware. It also helps to achieve much higher resilience by adding health checks and elements of self-healing and abstracts away the target hardware to simplify the development.

* Cross-functional teams need to understand how to use such tools effectively, and they need to become part of the standard development process.
* Dynamic scheduling also handles stability and resilience by autoscaling and restarting failing applications.

**Consequently**

**Developers build distributed systems and define how components will run and communicate with one another once they are deployed.**

* + Applications can scale up and down.
* + Non-functional parts can be restarted and healed automatically.
* − In a distributed system you invest in mechanisms for disaster recovery and disaster maintenance, which carry a high cost.
* − Maintenance is more complex.
* − Even when letting the public cloud handle it for you, Dynamic Scheduling is still exponentially more complex than running a single server.

**Common Pitfalls**

Trying to put Kubernetes (or any other orchestration tool) in place before making sure that all other pieces of the architecture are completely in place: containers, microservices, CI/CD, full monitoring. Trying to use a dynamic scheduler on its own will significantly increase the complexity and maintenance cost of the infrastructure while producing minimal value.

**Related Biases**

*Bandwagon effect*

The tendency to do something because many other people are doing it, and everybody is talking about Kubernetes these days. Ask for it by name, even if you aren’t exactly sure what it does or if it fits your use case!

*Pro-innovation bias*

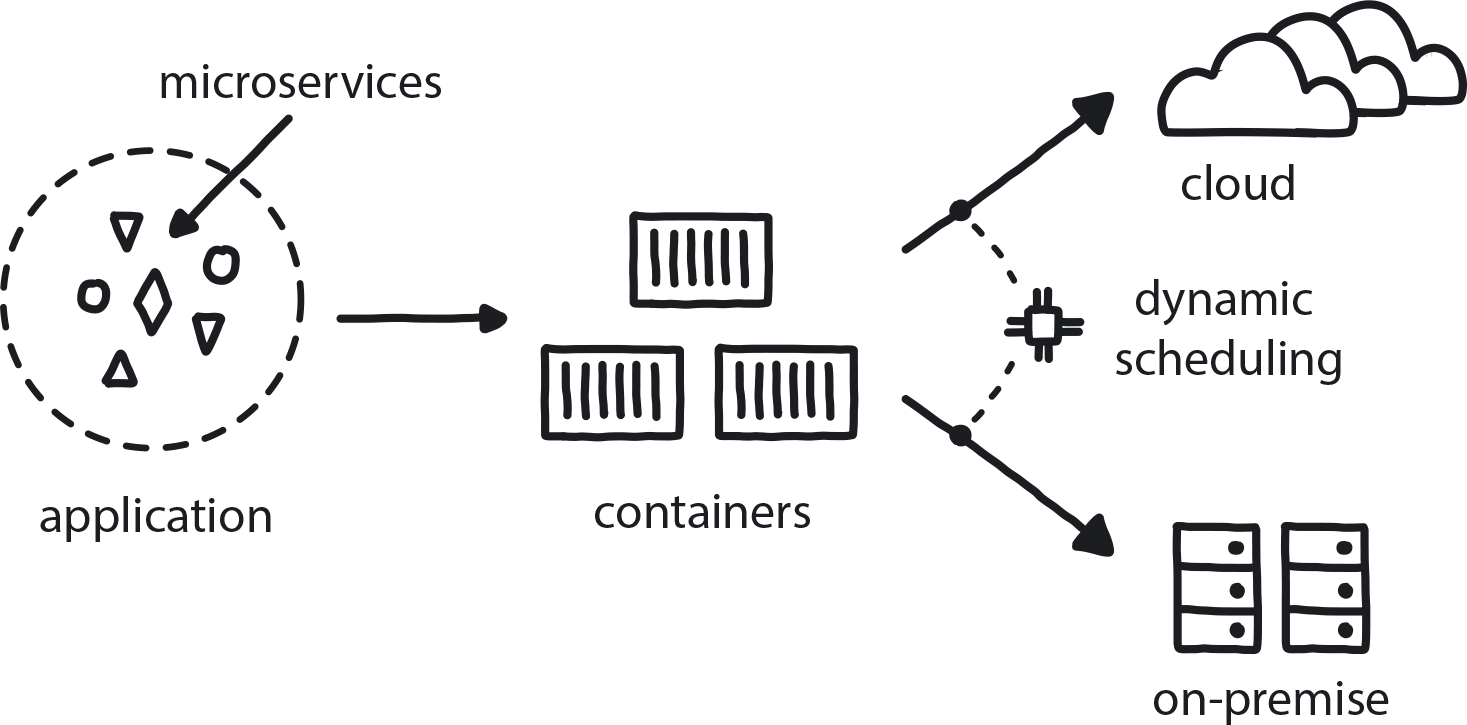
Having excessive optimism toward an innovation’s usefulness and applicability because it is new and cutting edge, without understanding its limitations or implementation context. Basically, thinking that Kubernetes is going to solve all of your scheduling problems for you.

**Related Patterns**

* [Microservices Architecture](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#microservices_architecture)
* [Distributed Systems](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#distributed_systems)
* [Containerized Apps](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#containerized_apps)
* [Platform Team](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#platform_team)
* [Build-Run Teams](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#build_run_teams_devops)

**Pattern: Containerized Apps**

*When an application is packaged in a container with all its necessary dependencies, it does not rely on the underlying runtime environment and so can run agnostically on any platform (*[*Figure 10-6*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#containerized_apps01)*).*



**Figure 10-6. Containerized Apps**

The team is building a distribution app that can scale up and down. Separate teams are meanwhile building microservices, and they need to take them quickly through a variety of dev and test environments all the way to production. There are a variety of different runtime environments, such as personal laptops, during development and different private and public clouds for testing, integration, and production.

**In This Context**

**Speed of delivery can be slow if an app depends on uniformity among different environments. Maintaining consistency of different tools and their versions on many machines across many on-premises and/or public clouds is difficult, time-consuming, and risky.**

* All stable environments tend to drift eventually.
* It’s impossible to manage thousands of machines manually.
* Differences in environment are difficult to find and to fix.

**Therefore**

**Package up an application’s code and everything it needs to run (all its dependencies, runtime, system tools, libraries, and settings) as a consistent unit so the application can be distributed to start up quickly and run reliably in any computing environment.**

This minimizes dependency on the runtime environment. Build once, run everywhere, and distribute quickly and easily.

* Use industry standards like Docker.
* Ensure proper versioning and simple and quick distribution.
* All dependencies included.
* All environments use the same basic tooling.
* Ensure that containers can run on local or any other environment.

**Consequently**

**Every part of the app is packaged in a container and can be easily started anywhere.**

* + Container tools like Docker improve efficiency.
* − Large numbers of separate containers increase system complexity.
* − Managing container images requires extra effort.

**Common Pitfalls**

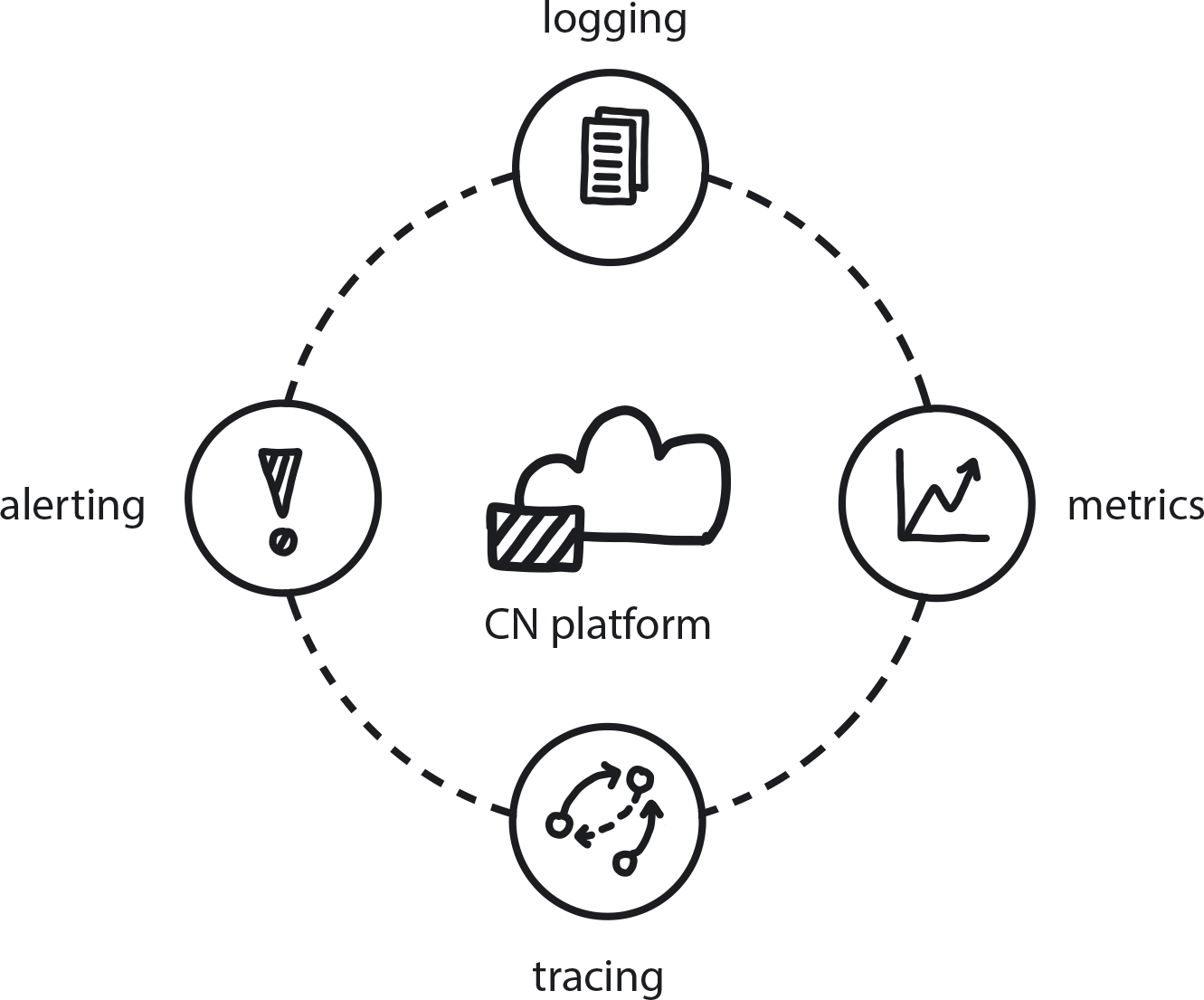
Building or managing container images manually, or not managing the versions on the released container images. This can lead to differences in different environments during the development of the runtime, which will introduce instability and lower the quality of the application.

**Related Patterns**

* [Microservices Architecture](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#microservices_architecture)
* [Reproducible Dev Environments](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#reproducible_dev_environments)
* [Distributed Systems](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#distributed_systems)

**Pattern: 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 (*[*Figure 10-7*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#observability01)*).*



**Figure 10-7. Observability**

Teams are moving to microservices, and there are more and more pieces—the number of components is growing. Traditional responsive monitoring cannot recognize service failures.

**In This Context**

**Traditional systems practice assumes the goal is for every system to be 100% up and running, so monitoring is reactive—i.e., aiming to ensure nothing has happened to any of these components. It alerts when a problem occurs. In traditional monitoring if a server fails, you will have an event; response, even if automatic, is manually triggered. This assumption is not valid for distributed systems.**

* A distributed system is by definition not 100% stable—when you have so many pieces, some of them will go up and down at random times.
* Resilience is built into the system to handle the assumption that eventually everything in the system will go down at some point.
* The number of components is always increasing while the number of people working on the application remains reasonably stable.
* Always a cost: if you get something, you must pay for it in some way. Here it is complexity, and you must manage it.
* Manual response is never fast enough in a cloud native system.

**Therefore**

**Put in logging, tracing, alerting, and metrics to always collect information about all the running services in a system.**

* Switch from a centrally planned system to distributed self-governing system.
* Continually analyze availability of services and behavioral trends of the individual services and entire system.
* Instead of focusing on specific pieces of hardware, focus on functional behavior of components.
* Consistently collect as many metrics as possible.
* Analyze the trends.
* Create an interface to make system state accessible to all stakeholders: anyone involved in system maintenance or development who needs to understand the system at any given time must be able to observe the system’s behavior.

**Consequently**

**There is a continuous overview of the state of the system, visible to anyone for whom this is relevant information.**

* + Analytics and proactive monitoring can be used to discover trends, which can be used to predict failure.
* − Any response to a single specific failure is extremely difficult.

**Common Pitfalls**

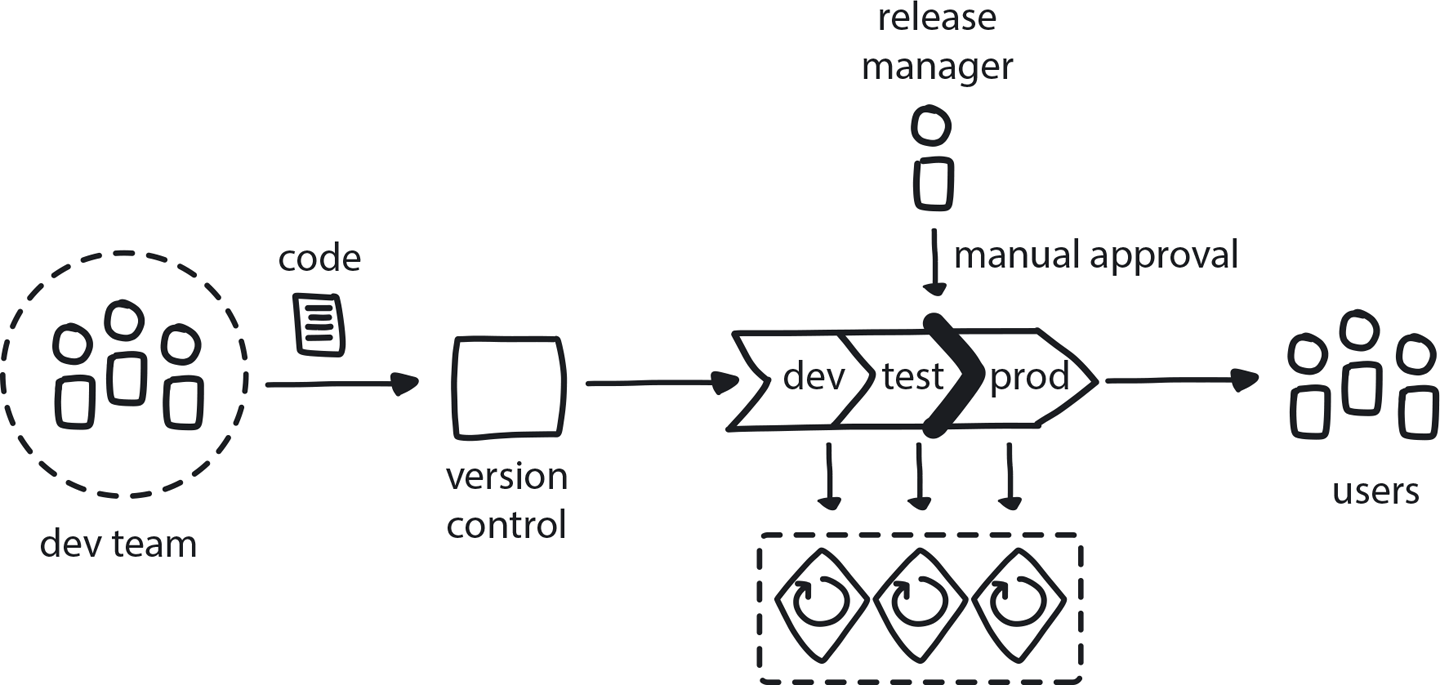
Lack of standardization through allowing multiple teams to do things how they like, so 10 different teams get to production in 10 different ways. Trying to create observability in this situation—to record, analyze, and reconcile behavior insights in a meaningful way from so many sources—is basically impossible.

**Related Patterns**

* [Measure What Matters](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch07.html#measure_what_matters)
* [Avoid Reinventing the Wheel](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#avoid_reinventing_the_wheel)
* [Public Cloud](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#public_cloud)
* [Full Production Readiness](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#full_production_readiness)
* [Learning Loop](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch07.html#learning_loop)

**Pattern: Continuous Delivery**

*Keeping a short build/test/deliver cycle means code is always ready for production and features can be immediately released to customers—and their feedback quickly returned to developers (*[*Figure 10-8*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#continuous_delivery01)*).*



**Figure 10-8. Continuous Delivery**

Teams are doing Continuous Integration, so each change is automatically built and tested separately in a very short time. Developers commit changes at least once a day. Teams are building distributed systems using microservices architecture.

**In This Context**

**A full quality test of a distributed system can be done only when the entire system is fully deployed and used. Teams try to reduce complexity and increase quality through infrequent and coordinated multiple-team deployments to the test environment, and then handing off to the operations team for more extensive testing and, finally, delivery.**

**Large-scale deployments at long intervals are usually difficult and time-consuming, so teams tend to minimize their number. This leads to painful integrations between services, reduced independence of teams, slower time to market, and—eventually—lower quality and higher costs for building the product or service.**

* There is more incentive to automate an operation if it happens more frequently (if something is painful do it more often).
* Some issues cannot be caught by automation.
* People gain trust in an action when it’s executed frequently and rarely fails.
* People lose trust quickly if quality is low or reporting unreliable.
* Small changes delivered by the teams themselves are easy to fix or revert.
* Recent changes are still fresh in developers’ minds.

**Therefore**

**Deliver every change to a production-like environment where all the services are delivered continuously and independently.**

Reduce the pain by using full automation coupled with increased speed and frequency of releases.

* No manual handovers on the way to deployment.
* Full automation is now in place.
* Test automation is now in place.
* Products are always in a releasable state.
* Experiments are more effective, as getting feedback from real customers is fast and painless.

**Consequently**

**The business can release to customers anytime.**

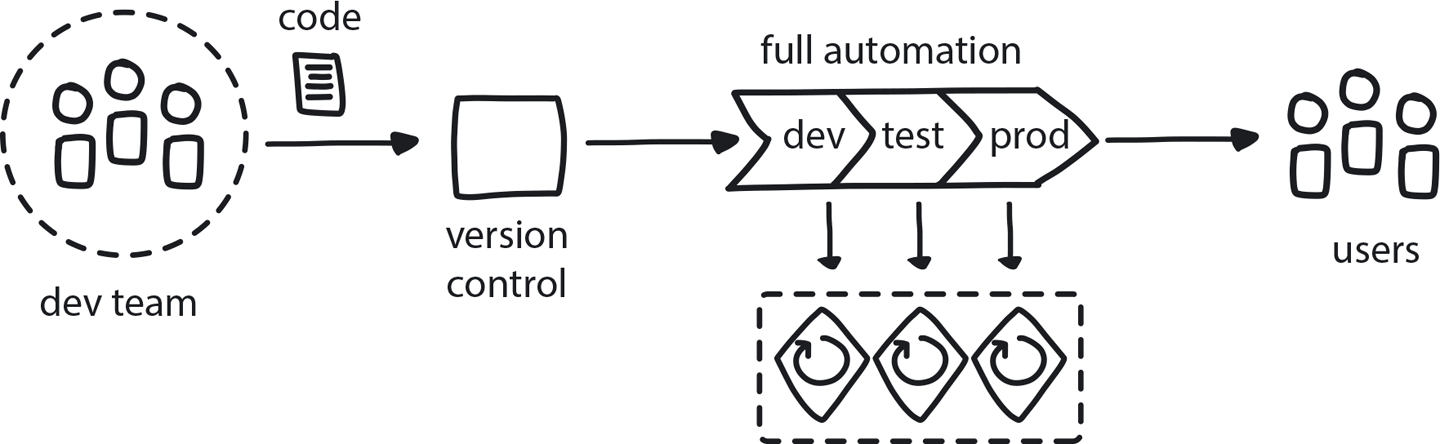
* + A feature can be tested very quickly with customers.
* − Some failures will sneak in through automation.
* − Very high level of automation is critical; any manual steps slow the process and introduce problems.

**Related Patterns**

* [Continuous Integration](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#continuous_integration)
* [Continuous Deployment](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#continuous_deployment)
* [Learning Loop](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch07.html#learning_loop)
* [Build-Run Teams](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#build_run_teams_devops)
* [Automated Testing](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#automated_testing)
* [No Long Tests in CI/CD](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#no_long_tests_in_ci_solidus_cd)

**Pattern: Continuous Deployment**

*Continuous Deployment automatically and seamlessly pushes code that has been accepted in the Continuous Integration/Continuous Delivery cycle into the production environment (*[*Figure 10-9*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#continuous_deployment01)*).*



**Figure 10-9. Continuous Deployment**

The team is using Continuous Delivery and building a distributed system using microservices. There is no regulation nor any other restriction on fully automated delivery.

**In This Context**

**Changes are frequently delivered to a production-like environment in a quick and automated way, but then they get stuck there.**

Many teams using CI/CD still have a human gate at the end, meaning that before changes can go live they may need to wait for the Ops team to do extensive testing, whether automated or manual, and/or wait for manual deployment. This means changes are not tested in a real live environment and carry high deployment risk due to potential differences in tools and processes. Time to market is slower; release manager delays releases due to risk and complexity.

* Any handover point slows down the process and introduces risk.
* Some industries have regulations that define roles during development and deployment (for example, release must be manually approved by a designated person of responsibility, as is common in financial sector regs).
* Frequent delivery better helps perceive customer needs.

**Therefore**

**Create a fully automated deployment process to take all changes to production all of the time. Use gradual deployment strategies and prepare for failure mitigation.**

Every change committed to the source code repository needs to be tested and ready for delivery to production at any given moment based on business needs and according to the relevant regulations.

* Build in full automation throughout system.
* No team or service dependencies are allowed.
* Make sure there is very good overview of the content and risks to the release manager.
* In case of regulatory restrictions that require a manual delivery step, automate everything before and after the release review point and make the release point as simple as pressing a button. Provide sufficient information to the release manager to allow fast and sound decisions.

**Consequently**

**Speed of delivery to customers is very high, and the changes are flowing to the customers daily or even hourly. Products or services are continuously providing higher value to the customers while allowing the developers to get real live feedback very quickly and learn and improve the products even further.**

* + Experiments can be done quickly and changes rolled back if necessary.
* − Some issues can still sneak in.
* − You have less control over feature release cadence.

**Common Pitfalls**

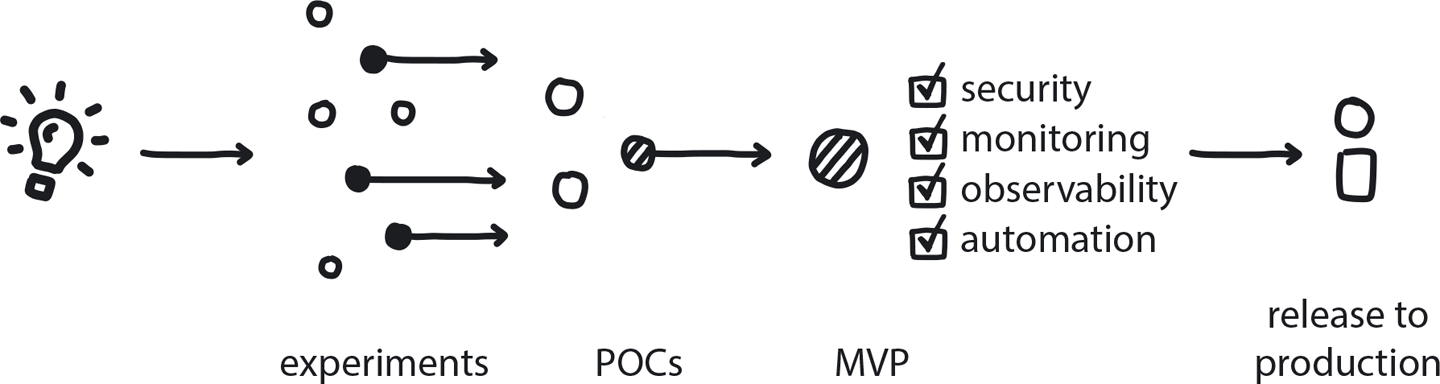
Attempting Continuous Deployment without an adequate level of automation. Every manual step or handoff inserts a significant speed bump in the process.

**Related Patterns**

* [Continuous Integration](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#continuous_integration)
* [Continuous Delivery](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#continuous_delivery)
* [Automated Testing](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#automated_testing)
* [Learning Loop](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch07.html#learning_loop)

**Pattern: Full Production Readiness**

*Make sure your platform is fully provisioned with CI/CD, security, monitoring, observability, and other features essential to production readiness before you try to take it live (*[*Figure 10-10*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#full_production_readiness01)*).*



**Figure 10-10. Full Production Readiness**

Core Team is stepping up the cloud native platform. The new platform and the first few applications moving over to it are scheduled for full production delivery very soon.

**In This Context**

**Too many companies try to rush their newly built cloud native platform into production before it is really ready. Typically the container scheduling platform is installed, but it’s still missing essential automation elements around maintenance and software delivery. This leads to poor quality of the running system and necessitates long delivery cycles. Many times the platform vendor will confirm production readiness of their tool—but without considering the whole system of delivery and maintenance cycles that are an equally crucial part of production.**

* Typical cloud native platforms today include only a partial platform and require significant additional configuration and automation.
* A typical full cloud native platform includes 10 to 20 tools.
* Once the main platform is fully running, there is pressure to deliver.
* In case of poor maintenance automation, platform teams could be overloaded with support tasks as teams onboard.

**Therefore**

**Before going to production all major elements of the platform, as well as those of any applications initially migrating to it, need to be in place. This includes having a scheduler, observability, security, networking, storage, and CI/CD. Also, at least basic maintenance automation must be in place.**

* System is automated.
* Maintenance is automated or at least documented.
* A bare and basic MVP version of a platform is not enough: there must also be observability, CI/CD, security, etc.

**Consequently**

**The new platform is functional and maintainable, and major tasks are automated. The Platform Team can continue extending the platform while providing satisfactory support for the development teams using it.**

* + All major elements of the platform are in place.
* − May delay release to production.

**Common Pitfalls**

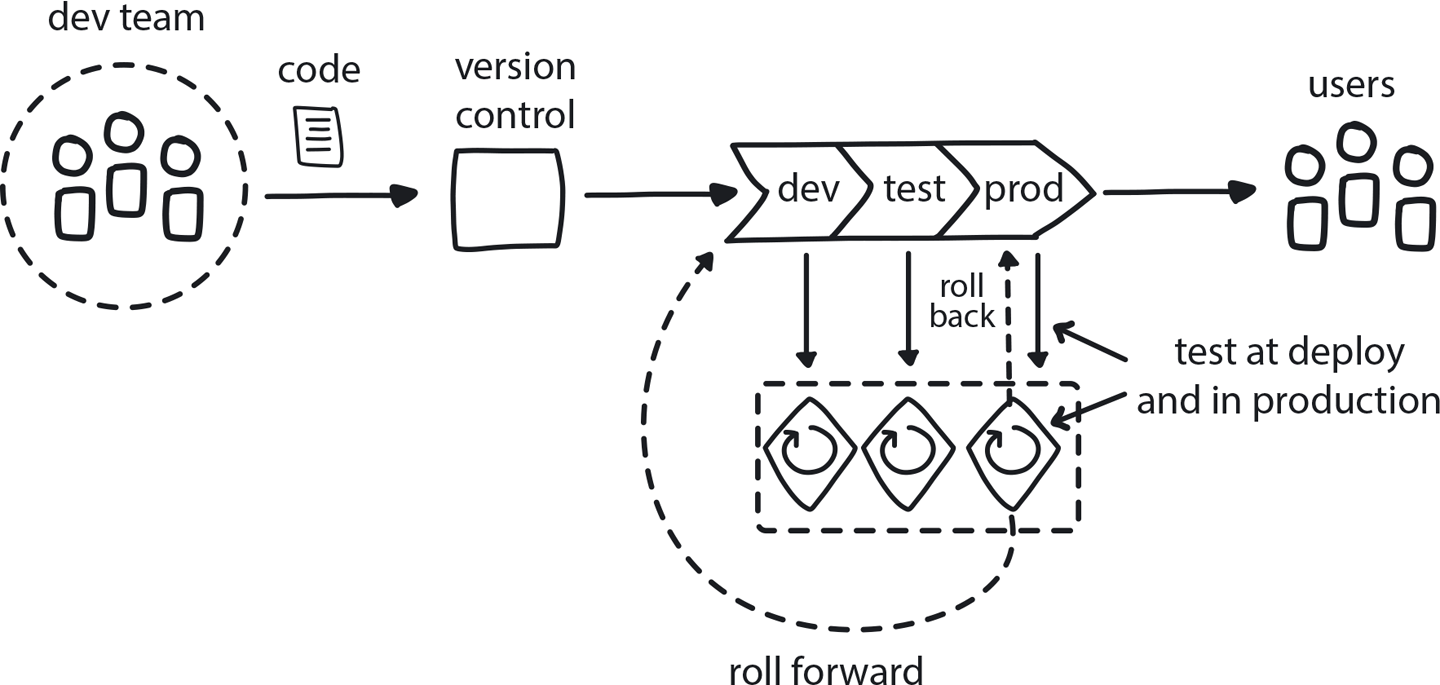
The most common pitfall is going to production without having a fully functional and automated CI/CD setup. This often happens because the Platform Team is pressured to deliver to production by the management, so they make a plan to go live and then complete the CI/CD setup when they have a bit of free time. Unfortunately, as we all know, that bit of free time will never materialize; the pressure will stay high and will only grow after the users are exposed to the new system. Further, the lack of CI/CD automation will be the biggest time consumer, demanding most of the Platform Team’s time as the Build-Run development teams introduce more applications. This leads to a vicious circle of not having enough time to automate delivery because we’re too busy deploying applications manually.

**Related Patterns**

* [MVP Platform](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#mvp_platform)
* [Observability](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#observability)
* [Secure System from the Start](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#secure_system_from_the_start)
* [Automated Testing](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#automated_testing)

**Pattern: Risk-Reducing Deployment Strategies**

*Employ release tactics to decrease the chance of problems happening when changes are introduced into the production system (*[*Figure 10-11*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#risk_reducing_deployment_strategies01)*).*



**Figure 10-11. Risk-reducing Deployment Strategies**

You have a cloud native system with high availability requirements. Teams are building microservices and releasing them independently and often.

**In This Context**

**Deployment is like a plane taking off: it’s the most dangerous part of flying. System failures typically don’t happen during normal operation but rather during maintenance and updating. This is because change always carries risk, and these are times when you are introducing change into the system.**

**Deployments are the most frequent and common type of change, and if not treated carefully can introduce a lot of instability. In cloud native, deployment happens *very* frequently, so this is an important area of concern.**

* Maintenance tasks are risk points.
* Deployment frequency in cloud native is high.
* Impact of changes to a complex distributed system is impossible to predict and can lead to cascading errors.
* Modern web users expect 100% availability and high levels of system responsiveness.

**Therefore**

**Use a variety of advanced release strategies to reduce the impact to end users when deployments occur.**

There are multiple ways to more safely release a new version of an application to production, and the choice is driven by business needs and budget. Options that work well for cloud native are:

* **Re-create**: First the existing version A is terminated, then new version B is rolled out.
* **Ramped** (also known as rolling-update or incremental): Version B is slowly rolled out and gradually replaces version A.
* **Blue/Green**: Version B is released alongside version A, then the traffic is switched to version B.
* **Canary**: Version B is released to a subset of users, then proceeds to a full rollout.
* **Shadow**: Version B receives real-world traffic alongside version A until stability and performance are demonstrated, and then all traffic shifts to B.

When releasing to development/staging environments, either a re-create or a ramped deployment is usually a good choice. If releasing directly to production, a ramped or blue/green deployment is a good potential fit but proper testing of the new platform is necessary. Blue/green and shadow strategies require double resource capacity and so are more expensive, but reduce the risk of user impact. If an application has not been thoroughly tested or if confidence is lacking in the software’s ability to carry a production load, then a canary or shadow release should be used.

Shadow release is the most complex, especially when calling external dependencies with mutable actions (messaging, banking, etc.). But it is particularly useful in common cloud native situations like migrating to a new database technology due to the ability to monitor live system performance under load while maintaining 100% availability.

(If your business requires testing of a new feature against a specific demographic filtered by parameters like geolocation, language, device platform, or browser features, then you may want to use the A/B testing technique. See the *A/B Testing* pattern.)

* Dynamic scheduling used in conjunction with other technologies supports a variety of deployment strategies to make sure infrastructure remains reliable during an application update.
* Use redundancy and gradual rollout to reduce risk.
* Limit the blast radius by deploying small parts of the system independently.

**Consequently**

**Risks to the system during deployment/maintenance are anticipated and managed.**

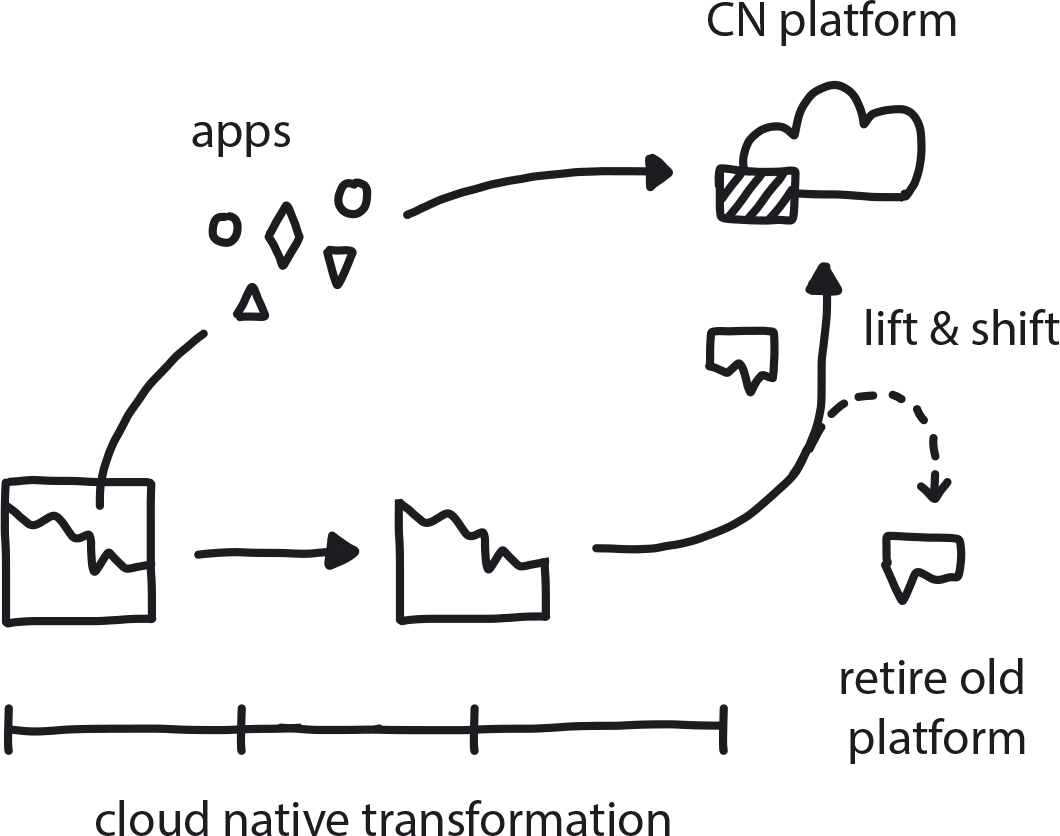
* + Deployments are frequent, and the Dev teams are confident.
* + Users get high availability and seamless responsiveness.
* − These setups are complex, and cost can be high.
* − The complex nature of these techniques can introduce new risks.

**Related Patterns**

* [Dynamic Scheduling](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#dynamic_scheduling)
* [A/B Testing](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#a_solidus_b_testing)
* [No Long Tests in CI/CD](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#no_long_tests_in_ci_solidus_cd)
* [Reproducible Dev Environments](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#reproducible_dev_environments)

**Pattern: 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 (*[*Figure 10-12*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#lift_and_shift_at_the_end01)*).*



**Figure 10-12. Lift and Shift at the End**

Most of your existing apps have been re-architectured and moved to the new platform. Some old stuff remains, but it is stable and doesn’t change too often.

**In This Context**

**Companies often first approach a cloud native transformation with the belief that it simply means migrating their existing system and processes onto the cloud. This is so common that it’s earned the name “lift and shift” and it categorically does not work if you try to do this at the beginning of a transformation. An organization has to change not only its technology but also its structure, processes, and culture in order to succeed with cloud native.**

When organizations do a lift and shift at the very beginning, they expect it will be a quick and easy project to execute, since the applications themselves do not change significantly. In most cases, however, it ends up being a large and expensive initiative requiring triple the anticipated time and budget. This is due both to the complexity related to the underlying platform and the new knowledge required for operating on the cloud. Such a painful experience may lead to delays or even total cancellation of the following refactoring initiatives.

* An expensive project to rebuild/refactor an app may prevent further improvements on app.
* “Moving to cloud” may look like a quick win.
* Public cloud vendors may offer incentives for you to move—lift and shift—inappropriately early in the transformation initiative.

**Therefore**

**Move functioning and stable legacy apps as-is from data centers to the new cloud native platform without re-architecture at the very end of the initiative.**

An organization has to change not only its technology but also its structure, processes, and culture in order to succeed with cloud native. Once the cloud native transformation is close to the end, however, it could be valuable to move any remaining parts of the old system to the new platform. This avoids the extra cost of supporting the legacy platform as well as gaining at least some benefits of the new platform for the old applications.

At the end of a successful refactoring, when the last few, rarely changing applications remain, the team continues to refactor them all, ignoring the piling costs and few benefits coming from such refactoring.

* Keep old tech like VMS and monoliths intact on the new platform.
* Spend minimal effort.
* Create easy ways to strangle remaining old apps.
* Lift and shift only apps that almost never change but cannot be retired easily.

**Consequently**

**You can retire the old platform, shut down the servers, and end the datacenter contracts.**

* + Benefits are gained from some features on new platform.
* − But this is a missed opportunity to update/re-architect these legacy apps/services, which will likely remain the same forever.
* − Some teams are stuck with old tech.

**Common Pitfalls**

By far the most common is to try to move the entire existing system exactly as it is onto AWS or some other cloud provider. All your processes and procedures remain exactly the same, only now they’re on cloud-based infrastructure instead of on premises. This kind of “lift and shift in the beginning” is a large effort that typically requires far more expense and effort than originally estimated. Furthermore, though there are always plans to continue refactoring to improve and update the new system, the team feels fatigue from change and becomes apprehensive of starting any other major refactoring project on the same set of applications.

Another, less common, pitfall is to insist on keeping two different platforms for years just to run a few remaining old applications that were never refactored in cloud native.

**Related Biases**

*Status quo bias*

If this way of doing things has always worked for us, why change anything now?

*Default effect*

When lifting and shifting an entire system onto a new public cloud platform, simply accepting the tools and options as packaged/configured by the provider instead of finding the best ones for your specific needs and circumstances.

*Bandwagon effect*

Rushing to move operations to the cloud because other companies in your sector are doing it and you want to do it too.

**Related Patterns**

* [Strangle Monolithic Application](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#strangle_monolithic_application)
* [Lean for Optimization](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#lean_for_optimization)
* [Architecture Drawing](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#architecture_drawing)

**Summary**

In this chapter we introduced patterns around cloud native infrastructure. The intent is to first expose readers to the patterns themselves before applying them—fitting them together in the transformation design outlined in [Chapter 11](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch11.html#applying_the_patterns) and [Chapter 12](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch12.html#applying_the_patterns_a_cloud_native). There we show how a company like WealthGrid can apply patterns step by step, from start to finish, as a transformation design. The design moves the company through four stages to emerge successfully transformed into a flexible, responsive, and above all confident organization, able to work both proficiently and innovatively as needed.

Once familiar with the patterns and ready to move on to applying them in a design, this chapter—along with the other chapters presenting patterns for strategy, organization/culture, and development/processes—functions as a more in-depth resource for referencing and working with individual patterns.

New cloud native patterns are emerging constantly. To continue sharing them and extending the cloud native pattern language we have established here, please visit [www.CNpatterns.org](http://www.cnpatterns.org/).

This is where to find the latest pattern developments, but also an online community for discussing and creating new patterns. We are inviting people from across the industry, thought leaders and influencers but most importantly everyday engineers and managers—those out there working elbows-deep in cloud native code and architecture—to contribute and participate. Hope to see you there!