**Chapter 9. Patterns for Development and Process**

A cloud native transformation is an exciting time for an organization. Who doesn’t want to work with the hottest new technologies using cutting-edge approaches? Introducing these new technologies and ways of working, however, can also knock you seriously off balance at first. You’re not only changing everything you do, but how you go about doing it—which can create a whole slew of problems. These can include slow delivery, reduced product/service quality, and difficulties in both team and project management, not to mention any brand-new complications that may be unique to your own transformation circumstances.

Cloud native processes are still being fleshed out because cloud native itself is still so emergent. This is not yet a beaten path so much as one that’s being actively created by the people walking it. What we do know, however, is that it’s critical to make sure that the foundation is right and the system architecture can support future growth, extension, and constant change.

The patterns in this chapter address how to approach designing, building, and delivering your business’s products or services in this new paradigm. This is where we look at the architecture and processes that support cloud native’s fast, dynamic, and responsive delivery model: microservices, continuous integration, and other process-oriented tools and methods that empower teams to be independent, proactive, and self-sufficient while delivering rapid, iterative changes on a daily basis. They are not inherently superior to other ways of building software, either singly or when harnessed together. What they do represent is a better set of tactics and processes to deal with uncertain situations. What we are giving you here is a practical way to implement them.

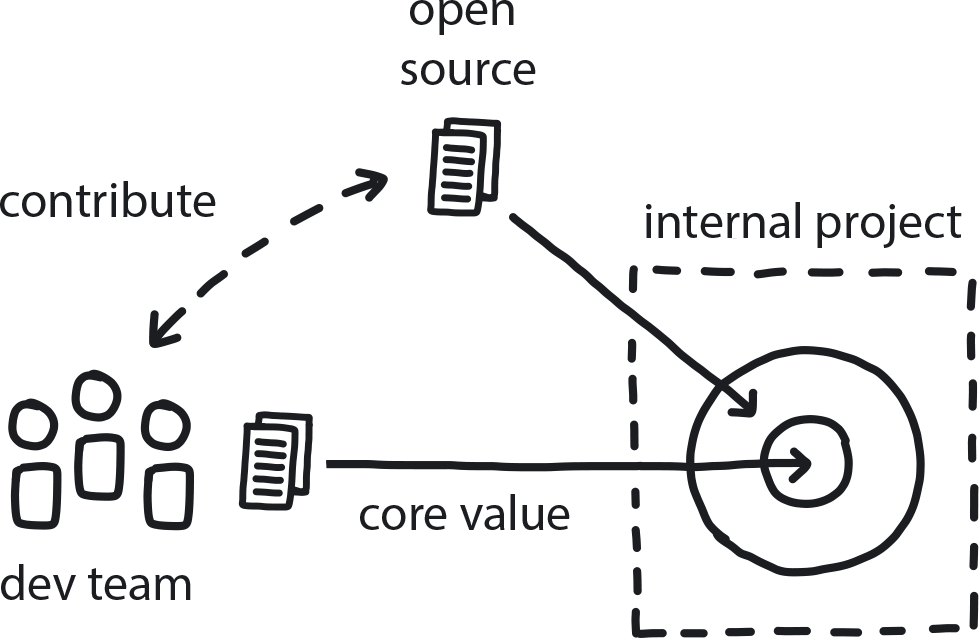
*The following patterns describe and address cloud native development and process. They are presented in an order that we believe will be useful or helpful for the reader, but there is no right (or wrong) order for approaching them: patterns are building blocks for a design and can be combined in different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

* [Open Source Internal Projects](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#open_source_internal_projects)
* [Distributed Systems](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#distributed_systems)
* [Automated Testing](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#automated_testing)
* [Continuous Integration](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#continuous_integration)
* [Reproducible Dev Environments](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#reproducible_dev_environments)
* [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)
* [Microservices Architecture](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#microservices_architecture)
* [Communicate Through APIs](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#communicate_through_apis)
* [Reference Architecture](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#reference_architecture)
* [Architecture Drawing](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#architecture_drawing)
* [Developer Starter Pack](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#developer_starter_pack)
* [Demo Applications](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#demo_applications)
* [Secure System from the Start](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#secure_system_from_the_start)
* [Strangle Monolithic Application](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#strangle_monolithic_application)
* [Delayed Automation](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#delayed_automation)
* [Avoid Reinventing the Wheel](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#avoid_reinventing_the_wheel)
* [A/B Testing](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#a_solidus_b_testing)
* [Serverless](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#serverless)

**Pattern: Open Source Internal Projects**

*Use open source solutions for any software need that is not directly related to the company’s core business value (*[*Figure 9-1*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#open_source_internal_projects01)*).*



**Figure 9-1. Open Source Internal Projects**

The company is building a lot of software, but most of it covers generic needs—only a minor percentage is related to delivering the actual core business products/services.

**In This Context**

**When a project is strictly internal, there is a tendency to cut corners to save time. Meanwhile, the open source community is constantly coming up with new tools to solve business use cases in the cloud native world.**

Internal projects that are not in a company’s core business area take time away from that essential work. Furthermore, they rarely get the priority to be built at the highest quality, and always get lowest priority for maintenance. Over time they become outdated and quality suffers, while innovation is limited or lost. Meanwhile, the rest of the market goes full-speed ahead.

* When something is invisible to the public, there are few incentives to make it nice.
* There are many other teams outside of the company having similar challenges.
* Good devs are attracted to interesting tech.
* Internal projects rarely get budget for cosmetic or procedural improvements.
* Many major cloud native tools are open source projects (Kubernetes!).

**Therefore**

**All software that does not address company core business (“secret sauce”) can be open sourced from the start.**

* Use open source software (OSS) whenever possible.
* New projects should be OSS by default.
* Use OSS governance and development practices even for internal projects.
* Always give back by contributing to the open source products you choose.
* Promote and market OSS projects.

**Consequently**

**If there is a gap in functionality, instead of building a new solution internally, use existing open source projects and contribute back to them. Alternatively, create your own open source solution and invite others to use, contribute to, and improve it.**

* + Code quality is higher, as the project is more visible.
* + Contributions from other people help the project continually improve.
* + Contributing to OSS boosts the company’s tech reputation.
* − Lose some control.
* − Competitors can use it too.

**Common Pitfalls**

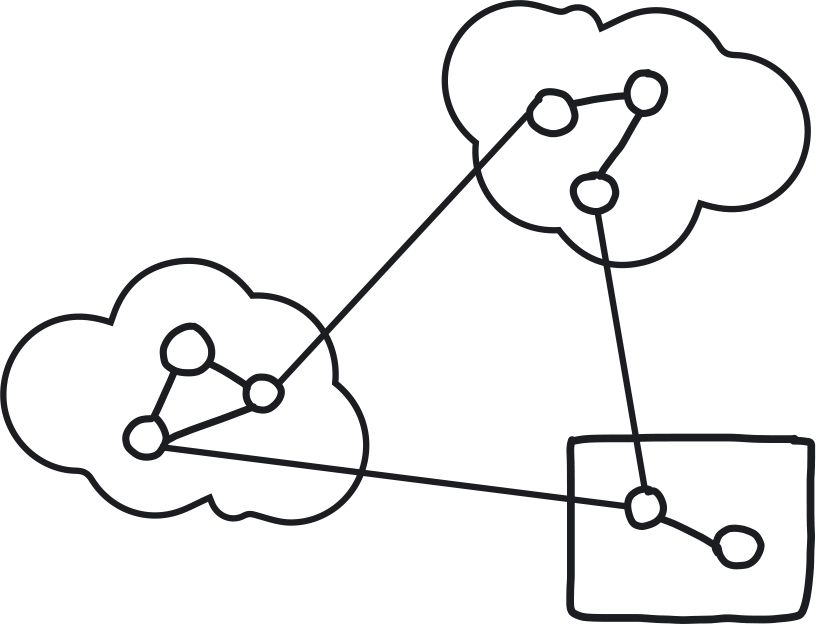
Using existing open source projects, such as Kubernetes, Linux, and many others, without ever contributing back. Open source projects are the best when a variety of developers and companies embrace them and help to improve and extend them.

**Related Patterns**

* [Avoid Reinventing the Wheel](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#avoid_reinventing_the_wheel)

**Pattern: Distributed Systems**

*When software is built as a series of fully independent services, the resulting system is, by design, fast, resilient, and highly scalable (*[*Figure 9-2*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#distributed_systems01)*).*



**Figure 9-2. Distributed Systems**

System complexity has grown beyond the capabilities of the key architects to understand it, but additional growth is still required.

**In This Context**

**Once the system has grown beyond the capacity of a single architect/engineer to understand, it becomes difficult and time-consuming to add functionality. With the growth of the old software systems, more people join the team and the system constantly collects technical depth, which leads to fragility and unpredictable side effects that come with every change. This creates fear of adding new functionality and stagnates development.**

* Human mental capacity for grasping complex systems is finite.
* Monolithic systems require someone with full understanding of the complex relationships among components to make/approve any changes to the system.
* Almost all software systems start out as small monoliths.

**Therefore**

**Build the software system as a number of independent components (microservices) running on different computers and communicating through APIs. Development, delivery, and scheduling of each component is completely independent, and any component can fail without affecting the others.**

Distributed systems are much more complex to initially architect and implement, but once that initial work is invested they are much simpler to grow—and evolve with improvements, new features, and changes.

* Split the system into small pieces (microservices components).
* Define APIs.
* Use more, but simple, computers that are less expensive and easy/fast to provision. Public clouds are most effective.

**Consequently**

**Higher complexity through many decoupled components makes a more resilient and scalable system.** Each component can become more complex until it is divided into smaller independent pieces to allow the system to grow indefinitely in scale and complexity.

* − At this level of complexity truly no one is capable of fully understanding the system, which makes it difficult to maintain.
* + High levels of automation and good observability help prevent problems.

**Common Pitfalls**

People fail to recognize cloud native as a true paradigm shift and attempt to treat the transition as simply a new installation of an Agile toolset. They try to implement cloud native using the ways they have always worked, so the new system—if they can even get it functioning—begins to resemble their existing monolith due to Conway’s law.

**Related Biases**

*Illusion of control*

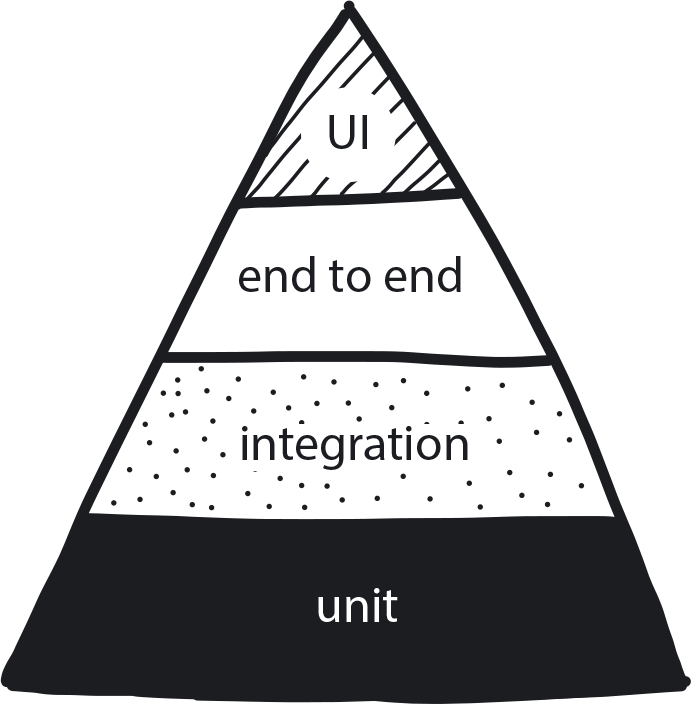
The tendency to overestimate one’s degree of influence over other external events is common in very complex systems, like distributed systems, and especially so in the uncertain circumstances of a cloud migration. Engineers think they know how to build microservices, and managers think they know what it takes to do DevOps. But in reality it is only an illusion of control. Many complex and emergent processes are difficult to even steer, much less control. Sometimes we need to embrace some uncertainty to ultimately get results.

**Related Patterns**

* [Architecture Drawing](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#architecture_drawing)
* [Reference Architecture](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#reference_architecture)
* [Microservices Architecture](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#microservices_architecture)
* [Automated Infrastructure](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#automated_infrastructure)
* [Serverless](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#serverless)
* [Strangle Monolithic Application](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#strangle_monolithic_application)

**Pattern: Automated Testing**

*Shift responsibility for testing from humans (manual) to automated testing frameworks so the quality of the released products is consistent and continuously improving, allowing developers to deliver faster while spending more of their time improving features to meet customer needs (*[*Figure 9-3*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#automated_testing01)*).*



**Figure 9-3. Automated Testing**

CI and CD are in progress. Legacy code is being refactored to microservices, and the team is aiming to deliver changes a few times a day.

**In This Context**

**Humans are too slow and inconsistent to be a blocking factor in the pipeline for deployment to production.**

Any human handover or task performed by a human will significantly reduce the number of changes a team can deliver and increase the time required to deliver them.

* Humans can never be as fast as computers.
* Poor testing quality undermines trust in the delivery process, and testing is a waste of time.
* People tend to delay and combine changes if each delivery is risky.
* When tests are slow, people tend to avoid running them as frequently as they should.

**Therefore**

**Automate all the testing required to take any product change to production.**

Most functionality should be tested using fast and local unit tests, integration tests can ensure that components are working well together, and only a small portion of the test coverage needs to be on the system UI levels. All long-running and manual tests should be gradually refactored and automated, and they should not block consistent flow of changes to production.

* Use testing pyramid.
* Long tests should not block release.
* Manual and long-running processes happen only in background.
* Continuously add and change tests.
* Consider test-driven development.
* Add advanced in-product testing like A/B, canary, blue/green, etc.

**Consequently**

**The team can trust that the delivery process will catch most issues and that changes will flow to production quickly.**

* + The team is ready to deliver changes and take the risks.
* + Developers write tests, which gives deeper insight into the code.
* − If there is a team in charge of manual testing, they may need to be retrained for new responsibilities.

**Common Pitfalls**

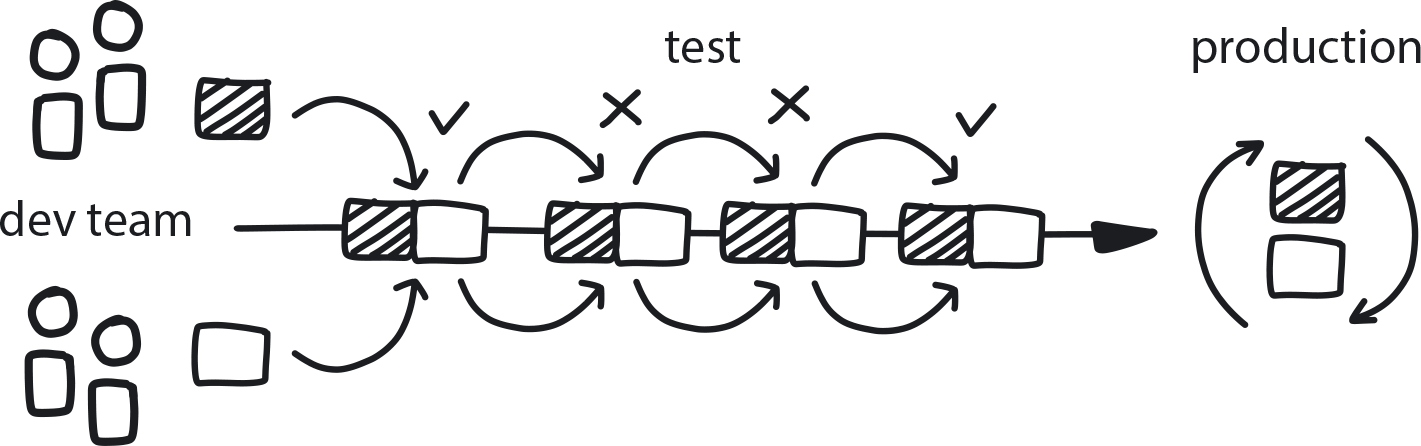
Automating everything except the actual release. Any manual step slows the process significantly, but some businesses (like finance/banking) are required by law to have a responsible administrator manually approve changes to go live. Where such manual approval is legally regulated, everything before and after the approval should still be fully automated.

**Related Patterns**

* [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)
* [Measure What Matters](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch07.html#measure_what_matters)
* [Observability](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#observability)
* [A/B Testing](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#a_solidus_b_testing)
* [Self-Service](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#self_service)
* [Build-Run Teams](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#build_run_teams_devops)

**Pattern: Continuous Integration**

*Frequent integration of small iterative changes speeds overall delivery and improves the quality of the code (*[*Figure 9-4*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#continuous_integration01)*).*



**Figure 9-4. Continuous Integration**

Many developers are working within the same codebase and need to integrate their changes.

**In This Context**

**When a team of developers works on a set of features that integrates only when all features are finished, the integration process tends to be very complex. The codebase change is large, and in the meantime other devs have integrated separate large changes that can further complicate the integration. To increase productivity, devs often delay interim integration—which leads to a single “big bang” integration just prior to release. A minor bug or conflict that could have been easily caught in an interim integration can now end up delaying the entire release.**

* Memory of the change you made fades with time, so delayed integration can increase difficulties.
* Chance of conflicts is smaller when the change is small.
* Frequent execution of the same task creates incentives for automation.
* It is very easy to lose trust in the system if reports are not available.

**Therefore**

**All developers integrate their changes at least once per day.**

Integration of all changes is done on a main codebase for each microservice. Code differences are small, less than one day of work, which leads to simpler integration. The main codebase is continually rebuilt and tested to ensure that every developer has functioning and up-to-date code to work with, which minimizes unexpected conflicts with any other newly integrated code.

* Introduce test automation and unit tests.
* Build each change and test it immediately.
* Immediately fix any broken build.
* Commit to the same mainline on the codebase.
* Must have good reporting.
* Use feature toggling.

**Consequently**

**Integration is a nonevent. Products are always in a releasable state.**

* + Code is always good-quality, tested, and functional.
* + Collaboration is easier.
* + Minor bugs and conflicts are caught before they cause major problems.
* − There is some overhead.

**Common Pitfalls**

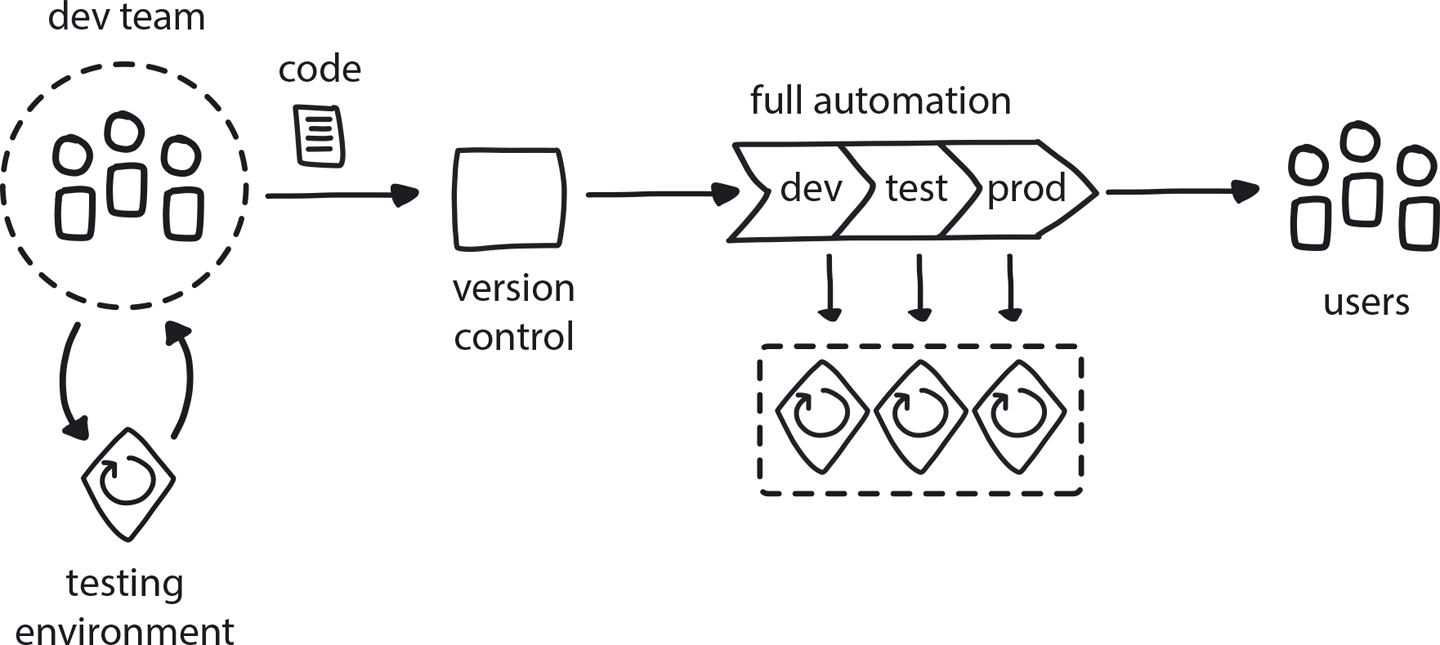
Many teams consider having Jenkins or some other continuous integration build tool running and then doing a full product build upon every change as a fully functional continuous delivery (CD). In reality, this is just a single element of the proper CD setup. The main goal is to have all the changes integrate quickly and very close to the actual time when the change was introduced. This requires all the team members committing their code to the same branch, a good and reliable test suite, fast uncompromising response to any failures that arise in the CD build, and more. Full CD can significantly boost code quality and team agility, while partial implementation will typically provide only marginal real value while creating the illusion of success.

**Related Patterns**

* [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)
* Decide Closest to the Action
* [Build-Run Teams](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#build_run_teams_devops)

**Pattern: Reproducible Dev Environments**

*Developers need to test their daily work in an environment that is easy to spin up and that matches production tooling as closely as possible (*[*Figure 9-5*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#reproducible_dev_environments01)*).*



**Figure 9-5. Reproducible Dev Environments**

Developers are building containerized microservices and deploying them to the containerized platform. Each microservice is built by a team, and microservices are interpreted into the larger system. There are many devs on many teams.

**In This Context**

**Shared environments and databases are difficult to keep in good shape and create dependencies that lead to delays.**

**When developers can’t create their own test environments, they may avoid running proper tests before submitting the code or run them on shared environments that may affect the work of their teammates. This affects other developers by making interpretation more difficult.**

**Differences between development environments and the eventual production environment may lead to the introduction of bugs that happen only in production and are related to those differences.**

**In all of these scenarios, product quality and developer productivity suffer.**

* Local testing reduces interpretation problems.
* If setup for the developer environment is too slow, devs will reuse the same environments.
* If not refreshed, local environments tend to undergo configuration drift.
* Shared environments create dependencies that lead to delays.
* Developers tend to create many test environments if it is easy and fast.
* Developers often have multiple changes that require testing at the same time.
* CI and CD are much more difficult to achieve without being able to test each change thoroughly.

**Therefore**

**Establish a fully automated and fast process to create development environments where devs can test-run their apps. Each developer should be able to have their own environment, or multiple environments, that resemble the eventual production environment.**

* Provide the same (or at least close to the same) tooling to deploy apps as in production.
* It is possible to do this on the cloud but will require cost management.

**Consequently**

**Each developer can run tests on their own without delays or disturbing the rest of the team.**

* + Productivity and product quality are high.
* − Could require a lot of hardware to create, at high cost.
* − If on cloud, devs may forget to switch them off and accidentally create large use charges.

**Related Biases**

*Bystander effect*

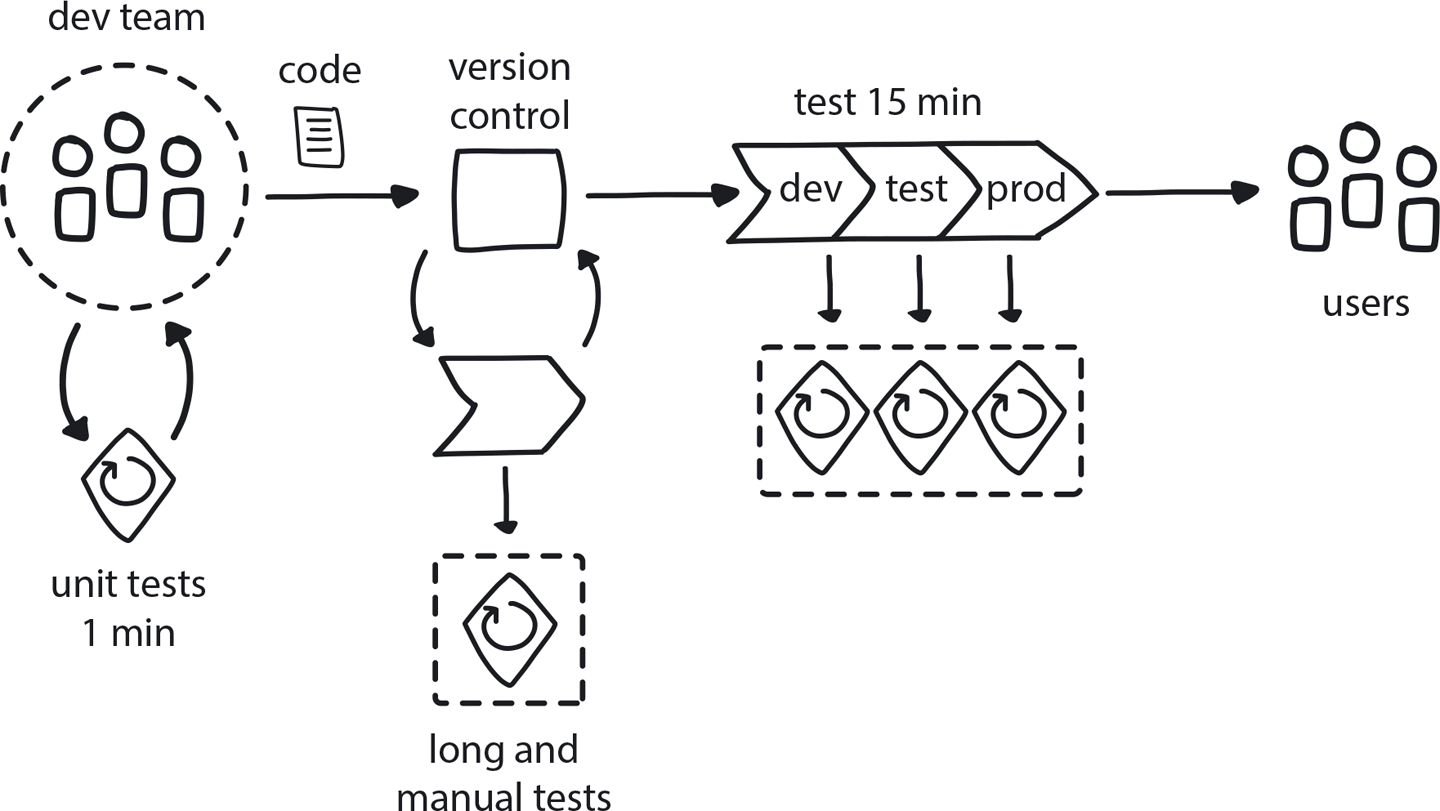
Doing nothing while hoping someone else will solve the problem.

**Related Patterns**

* [Containerized Apps](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#containerized_apps)
* [Risk-Reducing Deployment Strategies](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#risk_reducing_deployment_strategies)
* [Build-Run Teams](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#build_run_teams_devops)
* [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)
* [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)
* [Decide Closest to the Action](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#decide_closest_to_the_action)
* [Self-Service](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#self_service)

**Pattern: No Long Tests in CI/CD**

*Execute non-critical long-running tests in the background so they don’t block delivery to production (*[*Figure 9-6*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#no_long_tests_in_ci_solidus_cd01)*).*



**Figure 9-6. No Long Tests in CI/CD**

CI/CD is in place, and most tests are automated. Some are taking hours or even longer, and others require manual execution.

**In This Context**

**Delivering changes quickly is a major goal of the cloud native approach to delivering software.**

**Long-running performance tests, reliability tests, and manual and other types of full-system tests can take too long and delay delivery for hours—rendering CI/CD less valuable. Instead of dozens or even hundreds of times a day, delivery frequency gets reduced to just a few times per day. Similarly, fixing a bug or problem goes from taking a few minutes to instead requiring several hours.**

* Manual intervention can create very long delays.
* Frequent integrations require a fast build/test cycle.

**Therefore**

**Run your fastest tests earliest in the process. Schedule all tests that are manual, or which take longer than a few minutes to run, outside of the normal delivery process. If you have a test that is critical for functionality, however, it should be run as a blocking test.**

Mitigating risk is an equally important cloud native goal, which is why automated testing is a core tenet of the architecture. These two things need not conflict. There are strategies that allow adequate testing while enabling teams to still release quickly and constantly. Short-running tests can be incorporated into the pre-deployment process, while long-running tests can be executed in the background without blocking delivery to production.

* Run tests periodically post-delivery, and if a problem is found, either roll back the change or fix the problem and roll forward in a new release.
* Run long tests in parallel.
* Split the test automation into smaller test segments.

**Consequently**

**Testing does not delay or disturb delivery. Quality of the products is kept high by the right balance of ever-changing tests.**

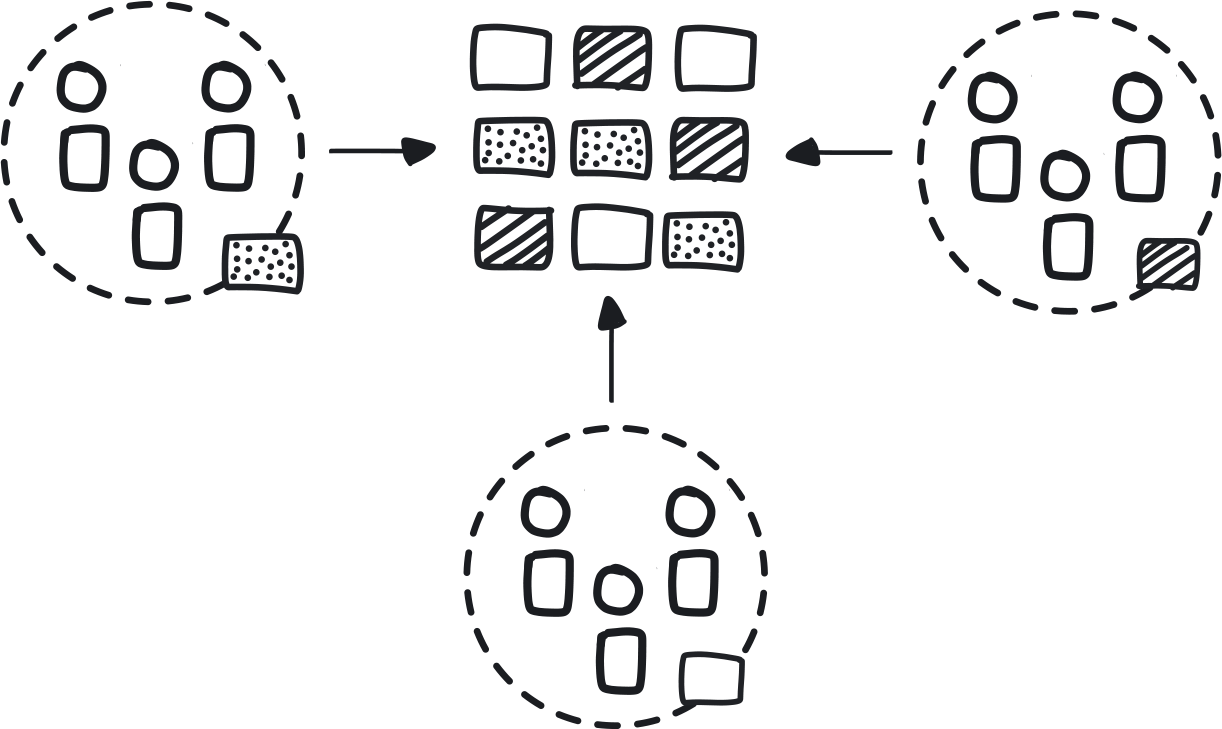
* + Non-blocking long-running tests reveal problems without slowing velocity.
* + Release is quick and easy.
* + Devs can deliver many times a day.
* − Some issues can carry through to production.
* − Requires strong roll back/roll forward protocol and procedures in place.

**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)
* [Continuous Deployment](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#continuous_deployment)
* [Automated Testing](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#automated_testing)
* [Reproducible Dev Environments](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#reproducible_dev_environments)

**Pattern: Microservices Architecture**

*To reduce the costs of coordination among teams delivering large monolithic applications, build the software as a suite of modular services that are built, deployed, and operated independently (*[*Figure 9-7*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#microservices_architecture01)*).*



**Figure 9-7. Microservices Architecture**

A company has decided to move to cloud native and is looking at ways to speed up feature development and to optimize their use of cloud resources. The size of the development/engineering staff can range from a few tens, for a small to medium business, up to a few thousand for a large enterprise.

**In This Context**

**Delivery of large monolithic applications developed by large teams require long and complex coordination and extensive testing, leading to longer TTM (Time to Market). Hardware use by such applications is inefficient, which leads to wasted resources.**

* People tend to delay painful moments; since integration and delivery are typically painful, their frequency tends to decrease as system longevity increases.
* Larger monolithic systems are increasingly difficult to understand as they grow in size and complexity.
* Monoliths are easier to work with than modular applications as long as they are small enough to be understood by each developer.
* Tiny monoliths (not big ones) are often the quickest, simplest solution to relatively easy problems.
* Conway’s law: architecture tends to resemble the organizational structure.

**Therefore**

**Split applications into smaller, loosely coupled microservices that can be built, tested, deployed, and run independently from other components.**

* Small and independent teams work on separate modules and deliver them with only limited coordination across the teams.
* Independent components allow different teams to progress at their own pace.

**Consequently**

**New systems are created from many small and independently built components with a complex web of connections.**

* + Faster-moving teams are not held back by slower ones.
* + Teams can choose the most appropriate tools for delivering their particular service.
* − Independence and freedom of choice are achieved, but with the tradeoffs of reduced standardization and certain types of reusability.

**Common Pitfalls**

All-or-nothing thinking: trying to build all the components at once, instead of concentrating on getting one service working well before moving on to the next.

Moving to microservices first—before establishing containerization, automation, or CI/CD. So you get hundreds of mini monoliths running around that all need to be deployed manually.

Not restructuring how teams are organized, so you are still building all your microservices in single-delivery cadence.

**Related Biases**

*Bandwagon effect*

The tendency to do something because many other people are doing it. When a hot technology is getting talked up at all the conferences or when Gartner puts certain tech on its chart, everyone decides to adopt it even without understanding how it relates to their use case.

*Pro-innovation bias*

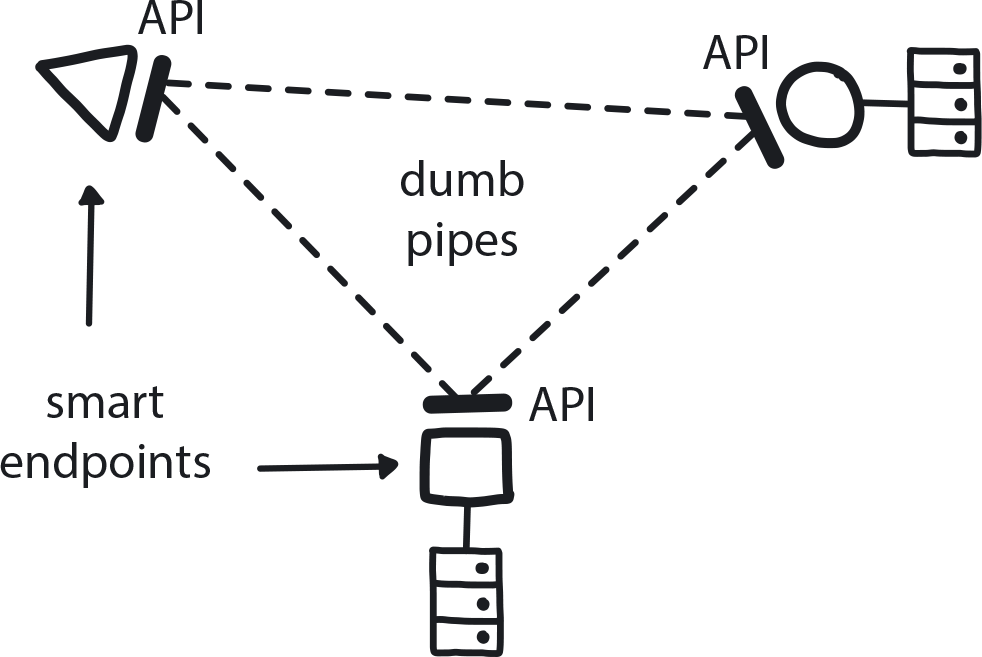
Having excessive optimism toward an innovation’s usefulness and applicability because it is new and cutting edge, without understanding its limitations or implementation context.

**Related Patterns**

* [Build-Run Teams](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#build_run_teams_devops)
* [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)
* [Avoid Reinventing the Wheel](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#avoid_reinventing_the_wheel)
* [Communicate Through APIs](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#communicate_through_apis)
* [Dynamic Scheduling](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#dynamic_scheduling)

**Pattern: Communicate Through APIs**

*In a highly distributed system, microservices must communicate with one another via stable and strongly segregated APIs (*[*Figure 9-8*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#communicate_through_apis01)*).*



**Figure 9-8. Communicate Through APIs**

A company is building a microservices application. Some teams work on a single microservice, others on multiple microservices. Teams are independent and aim to reduce interteam dependencies on both technical and organizational levels.

**In This Context**

**If APIs among microservices are not well-defined and fully segregated, they will require tighter coupling in development and/or delivery. This in turn introduces dependency, both service-to-service and among teams on an organizational level. This process essentially undoes the move to decouple the monolithic app in the first place, as it leads to coordinated development for the delivery of multiple services and requires very tight collaboration across teams.**

**This reduces the speed and agility of the organization and effectively re-creates the original monolithic architecture and organizational structure.**

* Tight coupling begins with a simple decision to share data directly.
* Conway’s law: a software application’s architecture will evolve to mirror the organizational structure of the company producing it.
* A single team working on multiple microservices may take shortcuts and introduce tight coupling.

**Therefore**

**Microservices should communicate with one another only through the network, using simple, consistent, and stable APIs.**

* Build stable APIs with backward compatibility.
* Place most of the service logic within the service itself, keeping the API simple and easily maintainable.
* Smart endpoints, dumb pipes (most of the business logic is in the microservices themselves and not in the APIs).
* Ensure each microservice has no direct access to data of other microservices.
* Make sure there is version control and version management for APIs.

**Consequently**

**Microservices are kept decoupled and independent.**

* + Old microservices can easily be replaced when needed as long as APIs are preserved.
* − Communication through network can be slower and is more complex to architect.

**Common Pitfalls**

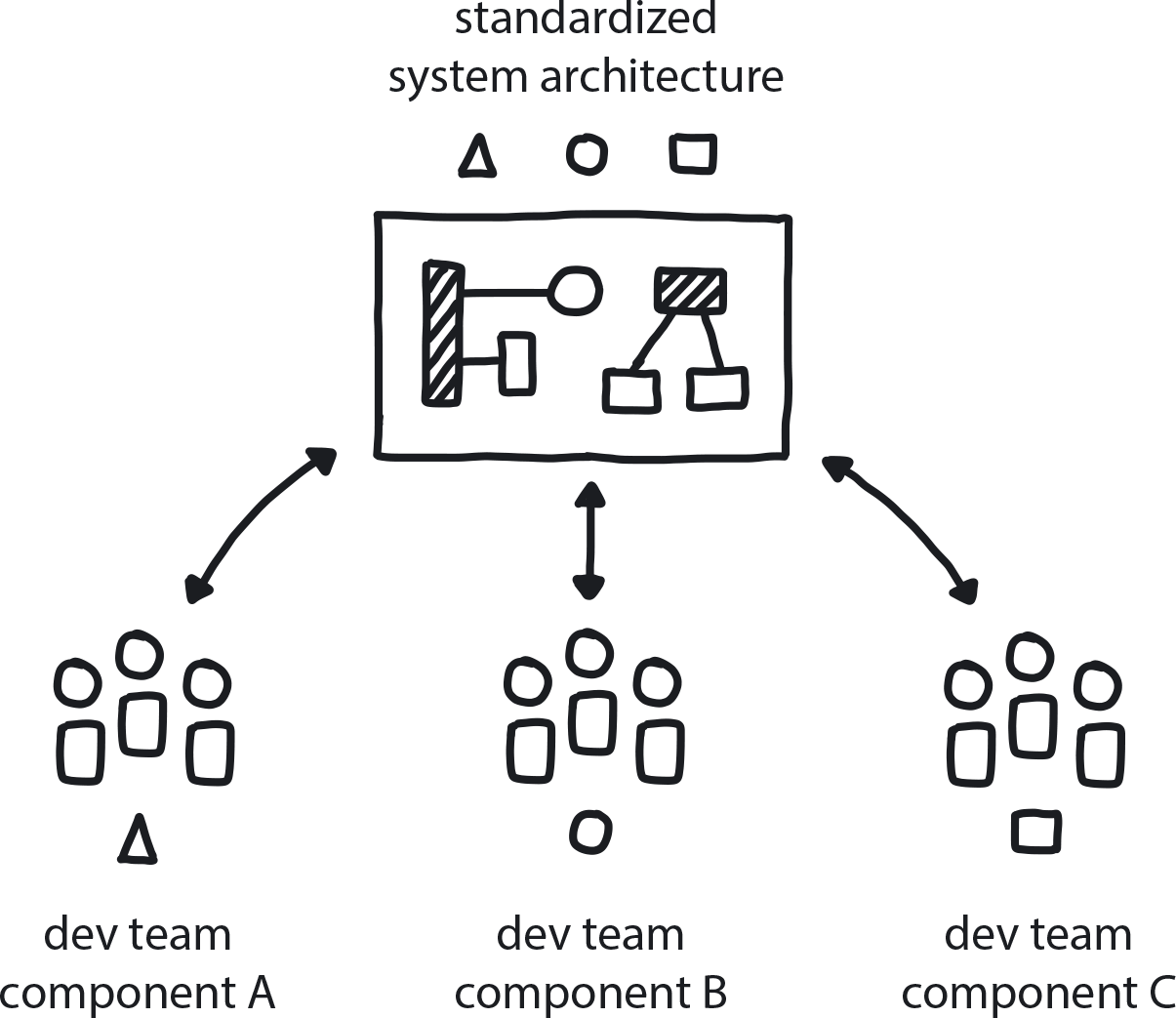
Some teams choose to do the quick and easy thing rather than the right thing at the beginning, thinking there will always be a chance to go back in later to refactor for APIs over a network. Even if this refactoring does eventually happen, which is not reliable, dependencies grow very quickly. It will be a time-consuming task to detangle such a mess, almost certainly more work than was saved by not doing well-defined, strongly segregated APIs in the first place.

**Related Patterns**

* [Microservices Architecture](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#microservices_architecture)

**Pattern: Reference Architecture**

*Provide an easily accessible document laying out a standardized system architecture for all teams to use for building their applications/components. This ensures higher architectural consistency and lowers development costs via better reusability (*[*Figure 9-9*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#reference_architecture01)*).*



**Figure 9-9. Reference Architecture**

The Core Team is designing the setup of the initial platform and the migration of the first few applications that will test it. The rest of the teams will start migrating soon and will need to understand the platform architecture.

**In This Context**

**When moving to cloud native, teams have no experience and no clear reference on the right ways to architect cloud native systems. When a transformation proceeds without a proper plan for standardizing the platform, each team will likely choose very different architecture for the particular piece it is in charge of building.**

This makes interpretation of components difficult and maintenance more complex, and will make it more difficult to move developers across teams due to steep learning curves among the different platforms. Furthermore, in the absence of both knowledge and easy solutions, teams may revert to well-known ways (biases) and significantly diminish the value of the transformation.

* It’s easier to reuse existing architecture.
* Some teams would never extend the original version.
* Given full freedom, teams will come up with many different architectures.
* It’s difficult to consider the whole system from within one team.

**Therefore**

**Document the architectural principles to be used in the company, educate the teams on the standardized way to build software in this system, and create an architecture review process to ensure consistency among teams.**

Standardizing the architecture early on paves the way for more rapid adoption while preventing chaos. This is an extension of the Vision First pattern: rather than making a bunch of random decisions to move the initiative forward, people with an understanding of how to build distributed systems software have helped make proper technical decisions.

Providing good reference points coupled with clear architectural guidelines right from the start helps the teams to bootstrap the projects in better ways and may avoid costly mistakes.

* Make the architecture sufficiently high-level to allow flexibility for the teams to choose tools.
* Use demo apps as example implementations.
* Create a procedure to uniformly educate everyone who will be using the system.
* Review and help teams to improve their microservices to run optimally on the standardized platform.
* Include recommended languages and tools.
* Just because we are being creative and doing experiments doesn’t mean we should not be doing architecture.

**Consequently**

**Components are consistent across all teams and projects. There is a clear understanding regarding the platform and agreement over preferred application architecture styles. The current state of the platform is known, and it is open for improvements.**

* + Easier to improve and maintain.
* + Easier to onboard new devs.
* − May limit freedom of choice.

**Common Pitfalls**

Teams will just accept the standardized platform, with default settings, as given and never work to improve or extend it to suit their particular work.

**Related Biases**

*Default effect*

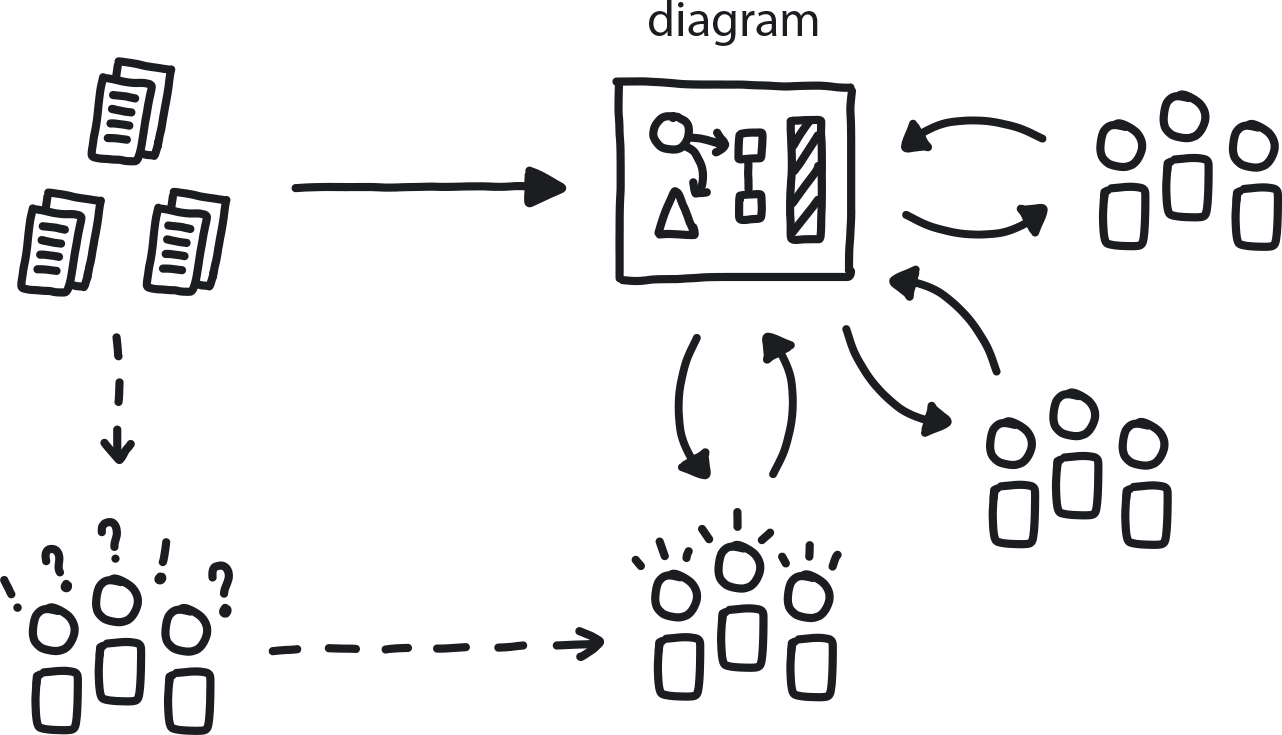
When given a choice between several options, people will tend to favor whatever default is already in place. This is why solutions to problems have to consider the correct/optimal defaults, since those will be adopted more frequently than any customized option. This is true both for the tools built into cloud platforms like Amazon Web Services, Azure, etc., as well as internal tools provided to employees. It’s why we have the Starter Pack pattern.

**Related Patterns**

* [Vision First](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch07.html#vision_first)
* [Architecture Drawing](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#architecture_drawing)
* [Communicate Through Tribes](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#communicate_through_tribes)

**Pattern: Architecture Drawing**

*A picture—or, in this case, a high-level outline sketch of your system’s basic architecture—can replace a thousand words, save time, and prevent misunderstandings (*[*Figure 9-10*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#architecture_drawing01)*).*



**Figure 9-10. Architecture Drawing**

The company is in the middle of cloud native adoption. Strategy is set, architecture defined. Now the company needs easy ways to discuss and improve the architecture.

**In This Context**

**When architecture is very complex and difficult to replicate—or, if as sometimes happens, a piece is entirely missing—a team can struggle to have a quick and effective conversation about technical solutions. Describing complex technical discussions using words can be time-consuming and confusing, resulting in many misunderstandings later on. This may lead to deviations in implementation, which can harm product quality and make maintenance more difficult.**

* Different people grasp information in different ways.
* Visualization helps people think more creatively.
* Common language (visual or verbal) helps save time and increase the volume of information during technical discussions.

**Therefore**

**Draw the high-level architecture on a whiteboard and teach all team members how to repeat the drawing.**

Use the drawing consistently in team discussions and in all internal documents. Create similar drawings for each subcomponent recursively.

* Create simple elements that are easy to draw.
* Limit the drawing to 20 elements or so.
* Create a digital version that resembles the drawings.
* Use the same graphical language for subcomponents.
* Keep a version of the drawings centrally available for easy reference, and keep it updated.

**Consequently**

**Everyone in the team can draw parts of the architecture in seconds. There is a common visual language as a basis for improved collaboration.**

* + Consistent visuals are used throughout the project.
* + Internalized understanding of the architecture’s components and how they relate to each other.
* − Standardized representation of the architecture can crimp creative thinking in the team and lead to conformity in early stages of the product development.

**Common Pitfalls**

Though this is a No Regret Move—easy and inexpensive yet extremely beneficial—few companies produce a single, unified official version of the architecture for circulation and reference. Instead, teams and individuals create their own versions, leading to a lot of confusion.

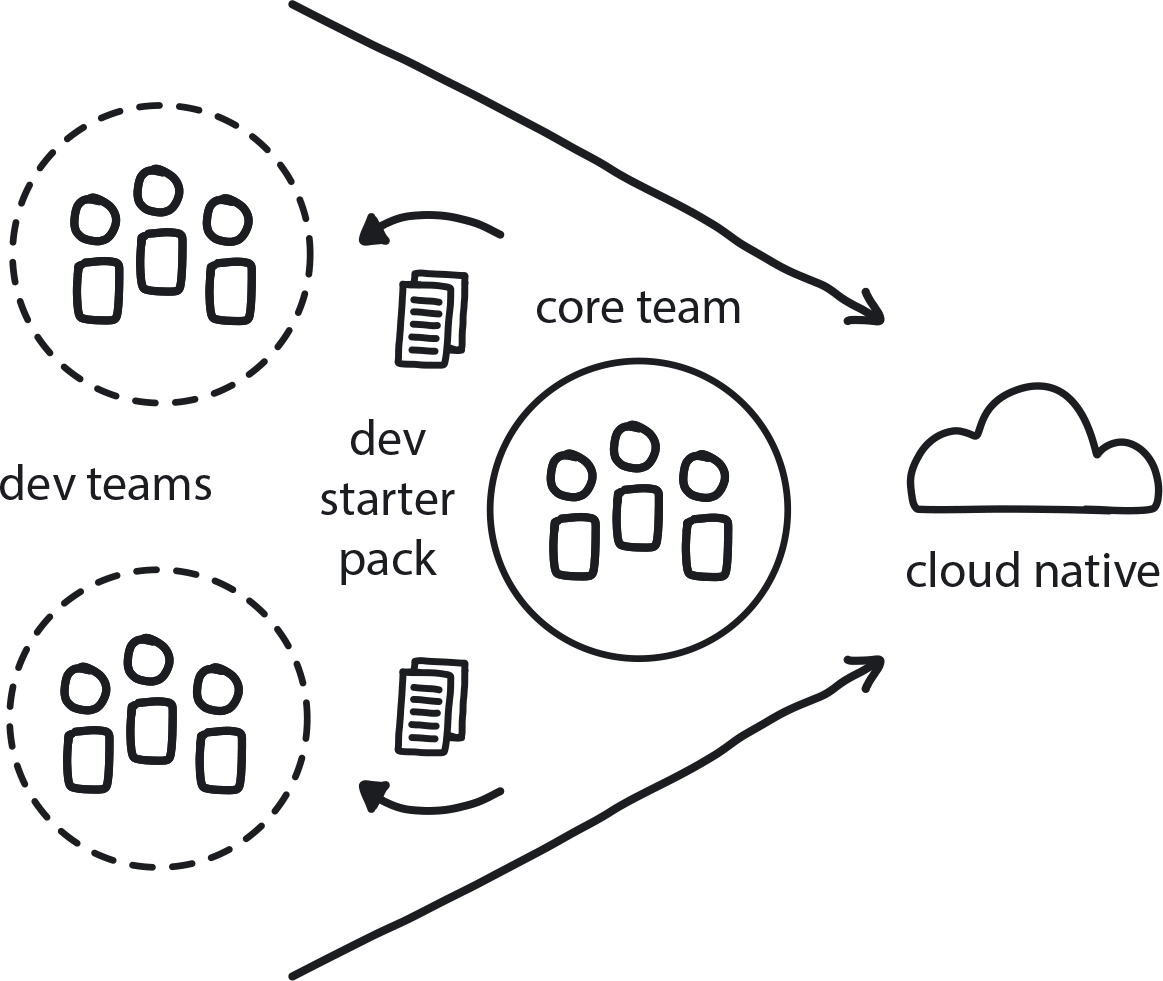
Another pitfall is when they do create one, but their visual representation of the architecture is so complex that no one on the team can understand it, much less replicate it.

**Related Patterns**

* [Distributed Systems](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#distributed_systems)
* [Design Thinking for Radical Innovation](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#design_thinking_for_radical)
* [Internal Evangelism](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#internal_evangelism)
* [Periodic Checkups](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch07.html#periodic_checkups)

**Pattern: 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 (*[*Figure 9-11*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#developer_starter_pack01)*).*



**Figure 9-11. Developer Starter Pack**

The new cloud native platform is approaching production-ready status, and it’s time to begin onboarding teams.

**In This Context**

**Teams onboarding to cloud native don’t know the tools or technologies and typically receive poor onboarding materials. At best, this leads to wasted time, and at worst, teams are forced to create their own cloud native practices, which are uninformed, not designed for this platform, and not uniform.**

* There are limited publicly available materials.
* People will use known techniques if they are not provided with clear guidance to new ones.
* If teams are onboarded with insufficient training, they will overload the support team with requests for help.
* People tend to accept default choices, so giving them good defaults increases overall quality.

**Therefore**

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

Optimally, new developers should be able to commit their first change and deploy it to the test environment on the first day following the onboarding.

* This cloud native “starter kit” of materials should include tool configurations, version control repository, CI/CD pipelines, demo applications for practice, target platform description, trainings, and more.
* All of this needs to be prepared before the next step of onboarding.

**Consequently**

**Cloud native practices are adopted as the Core Team has planned, and there is consistency.**

* + Less work and fewer problems for the Core Team after onboarding, because the newly onboarded developers have the tools and confidence to solve their own problems.
* − Less freedom for learning by doing for the dev teams.

**Common Pitfalls**

Teams will tend to anchor on using the provided starter pack as an ultimate solution instead of using this as a starting point and then innovating when better solutions are needed or become available. That is, they accept the starter pack as “this is just how you do it” and never explore alternatives.

**Related Biases**

*Curse of knowledge bias*

The Core Team has been doing this for a while and has thus lost touch with being new to cloud native technologies and tools. They know exactly what to do and can’t imagine why others don’t also understand/have this knowledge.

*Default bias*

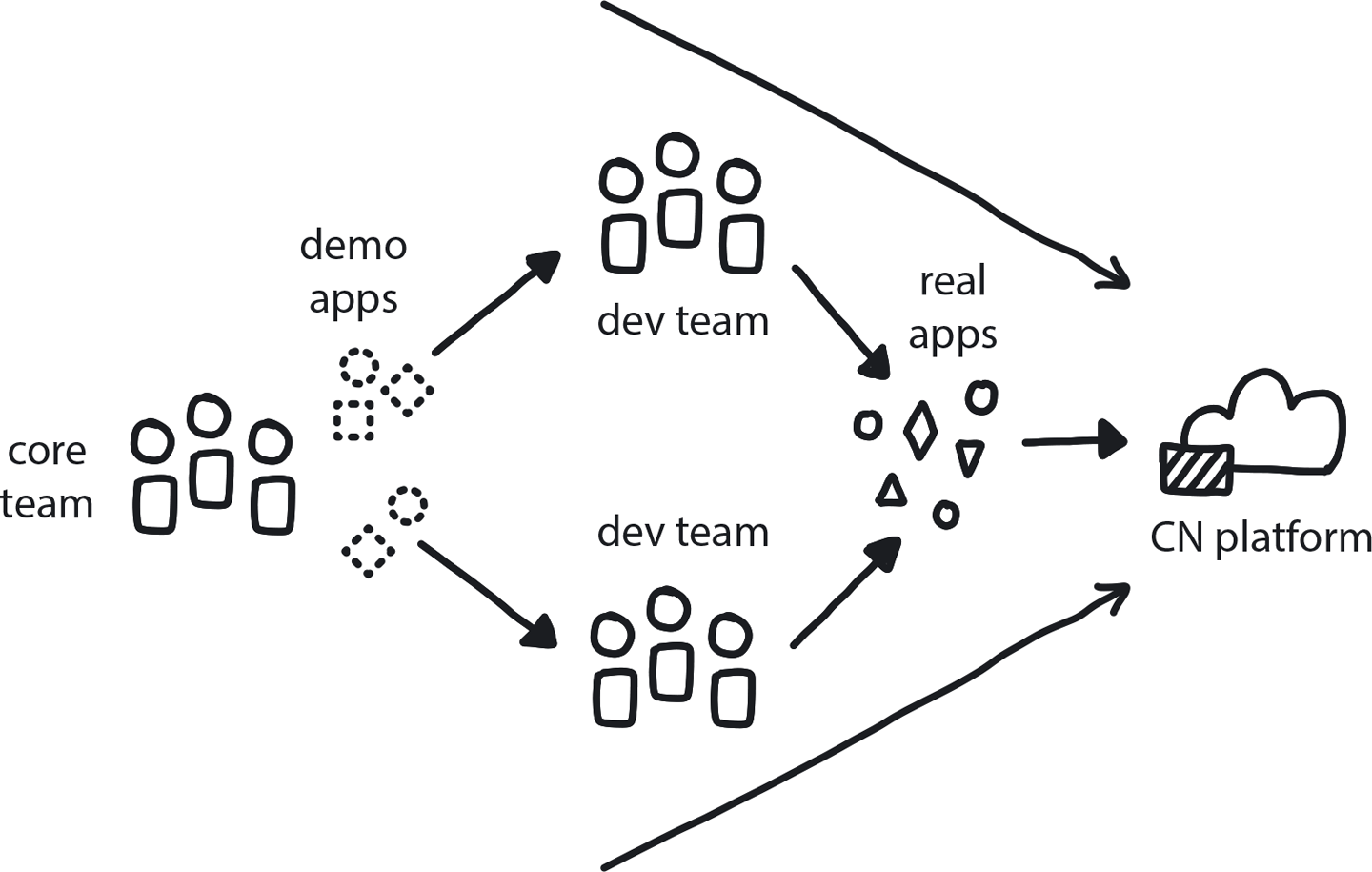
When given multiple options, the tendency is to choose the provided default. People accept what is provided and do not seek to adjust or customize it. We can use this as a nudge toward optimal choices by providing excellent default options.

**Related Patterns**

* [Gradual Onboarding](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#gradual_onboarding)
* [Reference Architecture](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#reference_architecture)
* [Demo Applications](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#demo_applications)
* [Full Production Readiness](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#full_production_readiness)

**Pattern: 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 (*[*Figure 9-12*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#demo_applications01)*).*



**Figure 9-12. Demo Applications**

The Core Team has built the initial platform and is ready to start onboarding the rest of the organization to cloud native. Developers have gone through platform trainings and soon need to start moving apps to cloud native. The level of cloud native experience is low.

**In This Context**

**Teams newly onboarded to cloud native have limited knowledge and no experience creating cloud native applications. They will tend to apply established skills and approaches carried over from previous experience in non-cloud native systems. This will lead to re-creating tightly coupled, interdependent applications—suboptimal architecture that conflicts with cloud native. This reduces overall quality for the apps they deliver and fails to capture cloud native’s development velocity benefits. Re-architecting apps later is much harder than building them the right way in the first place.**

* People tend to use known methods to solve new problems.
* Much easier to start from something rather than nothing.
* People learn by doing and from experiencing examples.

**Therefore**

**Build a number of simple, functional apps that fully fit cloud native practices.**

Make those apps known and available to new teams as they join the cloud native setup. Keep the demo apps up to date and adjust them to the latest best practices developed by the Core Team.

* Applications are basic but fully functional with a UI and a database, and built on microservices architecture with services communicating via APIs.
* Continuously improving—as the teams learn, they can incorporate new tools and methods to expand the application.
* Emphasize clean and high-quality code.
* Tests need to be automated/built in.
* The apps are to be delivered using CI/CD, and the delivery scripts are part of the applications.
* Always up and running—practice Build-Run Teams delivery workflow.

**Consequently**

**Teams moving to the new system have a way to practice their new skills and prepare to deliver a full enterprise application.**

* + Devs can start from the right place.
* + Core team can apply their knowledge.
* + Architecture is more consistent.
* − Demo apps could limit creativity (default effect).
* − Core Team spends time on writing demo applications.

**Common Pitfalls**

Moving to cloud native and trying to deliver a distributed architecture application by using old techniques/processes. Most often we see an application architected as microservices, but these are tightly coupled and delivered all together at the same time, essentially delivering a monolith of microservices.

**Related Biases**

*Law of the instrument*

An overreliance on familiar tools or methods, ignoring or undervaluing alternatives. “If all you have is a hammer, everything looks like a nail.”

*Default effect*

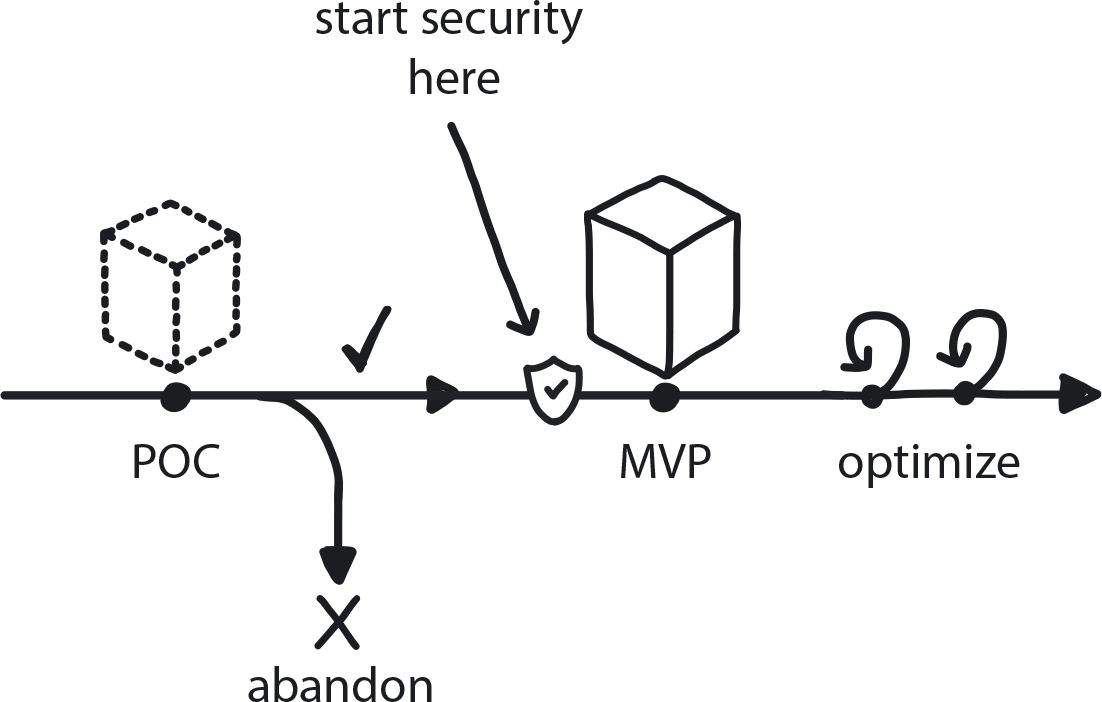
Whatever pre-selected options are given at the beginning tend to be kept as the default setting, instead of exploring better options.

**Related Patterns**

* [Developer Starter Pack](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#developer_starter_pack)
* [Gradual Onboarding](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#gradual_onboarding)
* [Learning Organization](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch07.html#learning_organization)
* [Build-Run Teams](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#build_run_teams_devops)

**Pattern: Secure System from the Start**

*Build security into the platform beginning with the earliest versions to ensure your distributed system is unbreachable by design (*[*Figure 9-13*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#secure_system_from_the_start01)*).*



**Figure 9-13. Secure System from the Start**

The company is moving to cloud native and in the process of building an MVP platform and starting to set up cloud native organizational structure. Distributed Systems, Microservices Architecture, and CI/CD are being used. The MVP is planned to go into production in just a few months.

**In This Context**

**Teams tend to delay setting up security until relatively late in a project. Cloud native, however, requires many new tools and security techniques, and teams are typically not proficient in working with distributed systems. Waiting to implement security features just before the platform is ready to go live leads to either poor security in production or significant delays while good security measures are finally taken.**

* Security in distributed systems cannot be ensured by perimeter security.
* It is more difficult to add security after the fact.
* The cloud native world requires many new and unfamiliar tools and methods.

**Therefore**

**Build the MVP as a highly secure system from Day One.**

Embed security practices in the Startup Pack and the Demo Application and run security tests as an integral part of the testing suite during the CI/CD process. This will allow the needed time to create security practices and provide examples for the teams onboarding to the platform.

* Provide good-quality and ongoing security training.
* Ensure that automated security testing is in place.
* Review the security of every tool in your cloud native system.
* Use best practices to secure containers, clusters, APIs, access rights, etc.

**Consequently**

**Security is a high priority from the start and baked in throughout the platform. The Build-Run teams and Platform Team have clear guiding principles for creating secure Distributed Systems.**

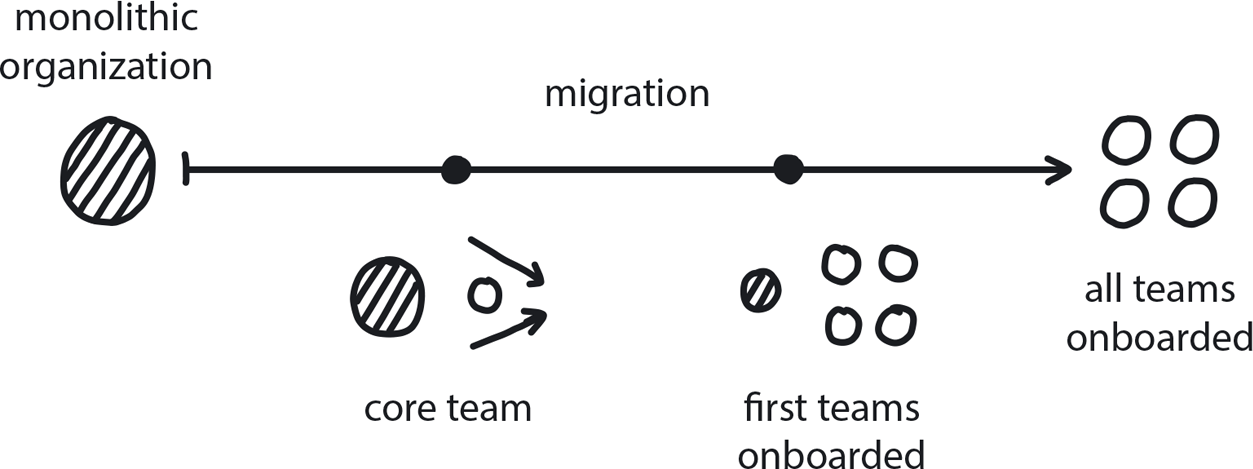
* + There is no extra cost to add security from the start.
* + The team is proficient in distributed security.

**Related Patterns**

* [Distributed Systems](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#distributed_systems)
* [Automated Testing](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#automated_testing)

**Pattern: 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 (*[*Figure 9-14*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#strangle_monolithic_application01)*).*



**Figure 9-14. Strangle Monolithic Application**

You have a monolith and are moving to microservices architecture. The new platform is ready or soon to be ready and you’re preparing the strategy for splitting the monolith into microservices.

**In This Context**

**Re-architecting a large monolith, built over many years or even decades, is a massive project that can take years. Some companies try to do it all at once, but rewriting a large monolithic application from scratch also carries great risk. You cannot start using the new system until it is developed and functioning as expected, but your company has little cloud native experience or knowledge to get this done. Building a new system from scratch will take a year or (likely) longer. While it is under construction there will be minimal enhancements or new features delivered on the current platform and so the business risks losing market share.**

**There is also a large risk of doing it all wrong in your first attempt. If the first project covers the entire application, then it will be very difficult to step back and start over due to sunk-cost fallacy—even if doing so is the best solution.**

* Teams don’t yet know how to split the monolith into microservices.
* The first time you do something you are going to make mistakes; it is a learning experience rather than an execution.
* Monoliths hide unexpected problems inside their huge size.
* A well-scoped migration can handle problems as they emerge, but if you are trying to do everything all at one time, they will cripple the initiative.
* 20/80 principle: it takes 20% of the time to get 80% finished, and then 80% of the time to finish the last 20% (and the last 1% will take as much time as the first 99%, so keep that 1% on the mainframe—see *Lift and Shift At the End).*

**Therefore**

**Once the cloud native platform is ready, take small pieces of the monolithic application and, one at a time, re-architect them and then move them to the new platform.**

The business value of new functionality is achieved much more quickly, and the cloud native architecture of loosely coupled services means future refactoring work will be simple. This is the cloud native version of Martin Fowler’s classic strangler pattern.

* Going piece by piece and over an extended period of time is key.
* First have the final functional platform in place.
* Give priority to pieces that change frequently and those that are easy to extract.
* Create demo apps.
* Document a simple way for migrating pieces to the platform to make the process consistent, replicable, and as quick and effortless as possible.
* Leave the things that are running but not changing at all behind on the old system, and move them at the very end.

**Consequently**

**There is a mixed environment of old and new applications working together. The team is getting better at re-architecting the pieces of the monolith.**

* + A plan is in place for moving pieces over time.
* − Some teams are still working in the old environment—the entire company is not all moving to Kubernetes on Day One.
* − Two different operational models are in place, which can create its own set of problems.

**Common Pitfalls**

Trying to do it all at once as a single massive re-architecture project: regrooming the monolith into dozens or even hundreds of pieces.

Lifting and shifting the entire monolith at the beginning, instead of remnants at the end.

**Related Biases**

*Pro-innovation bias*

The belief that new technology can fix all old problems.

*Planning fallacy*

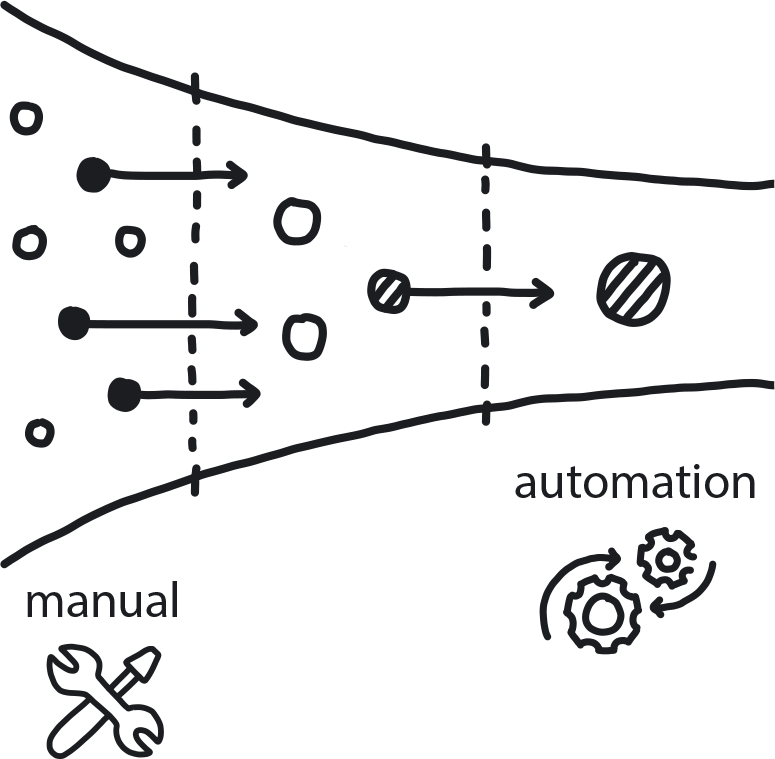
The human tendency to underestimate the time a project will require for completion. Especially operative in uncertain situations like moving to cloud native for the first time. We are eager to estimate the time and resources required, but we have no idea what it actually takes to move to cloud native. So some people estimate it as a few weeks of work when it often takes a year or longer.

**Related Patterns**

* [Lift and Shift at the End](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#lift_and_shift_at_the_end)
* [Architecture Drawing](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#architecture_drawing)
* [Create Demo Apps](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#demo_applications)
* [Manage for Proficiency](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#manage_for_proficiency)

**Pattern: Delayed Automation**

*Automate processes only after a problem has been completely solved and the solution has been run manually a few times (*[*Figure 9-15*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#delayed_automation01)*).*



**Figure 9-15. Delayed Automation**

The team is building a complex system that needs to support stress and a large and fluctuating number of users. The problem and the domain are not fully known. The solution is new and not easily uncovered.

**In This Context**

**Automation is essential for success in cloud native, but people tend to try to create a full and automated solution in the beginning before real pain points are uncovered (taking an academic approach rather than experimental). This leads to automation of the wrong thing when the problem is not fully understood. Or, paraphrasing Bill Gates, who describes this conundrum as “Crap in, crap out, only faster.”**

* Universities teach to solve the problem in the “right” way.
* Engineers prefer automation versus manual work.

**Therefore**

**Understand the problem well, create a solution, make it work, and only then automate, scale, optimize, and improve.**

Before automating anything, first solve the problem manually. The team needs to see the solution by doing it manually for a bit to experience and identify the pain points. Focus first on low-hanging fruit of automation (i.e., tasks that demand a lot of human time and are easy to automate).

* Run the process manually a few times.
* Create a blueprint (a document with steps).
* Do crude automation first (experiments, then an MVP version).
* Optimize and scale.
* Continually improve.

**Consequently**

**Only the right things get automated. All the important and time-consuming tasks get automated eventually.**

* + Scaled work becomes a well-understood process.
* − Process is manual for a while.

**Related Biases**

*Bandwagon bias*

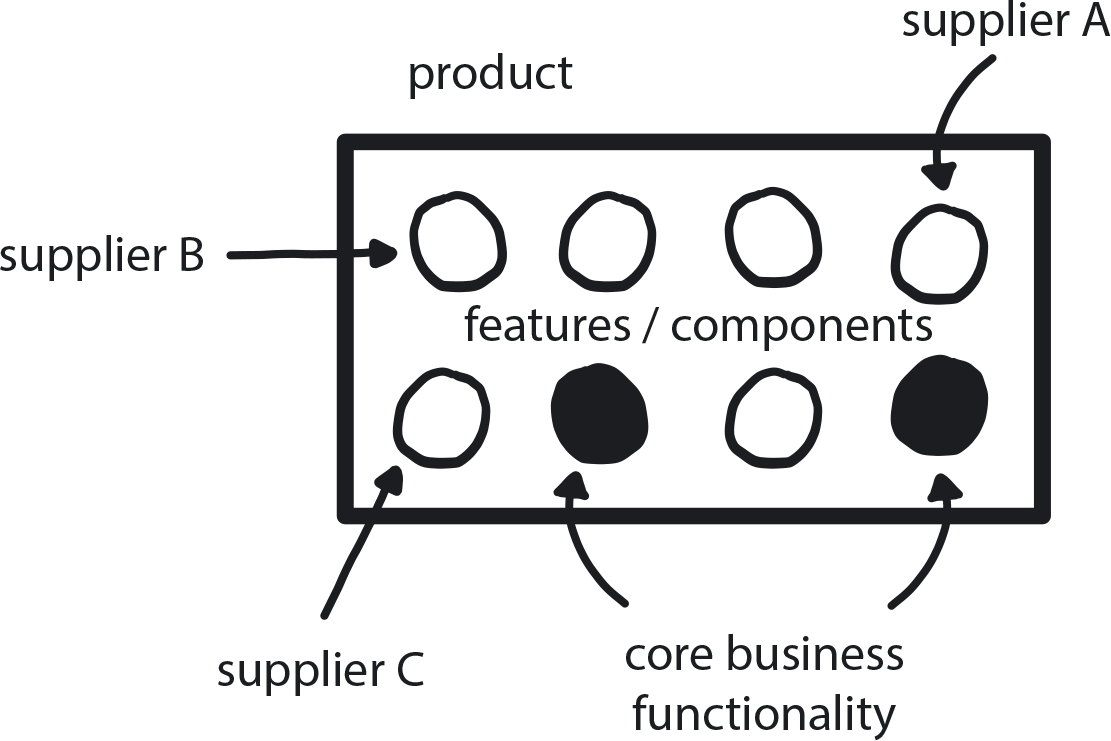
Everyone says automation is important, so we’d better do it immediately!

**Related Patterns**

* [Exploratory Experiments](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#exploratory_experiments)
* [PoCs](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#proof_of_concept)
* [MVP](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch08.html#mvp_platform)
* [Delayed Automation](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#delayed_automation)

**Pattern: Avoid Reinventing the Wheel**

*When possible, use open source or purchase commercial solutions for any need that is not your actual core business instead of trying to custom-build perfect tools (*[*Figure 9-16*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#avoid_reinventing_the_wheel01)*).*



**Figure 9-16. Avoid Reinventing the Wheel**

A team is in the middle of a cloud native transition and missing some functionality. Out-of-the-box solutions are available on the market, although the team is capable of creating its own.

**In This Context**

**Many development teams tend to create their own tools/solutions even when reasonable alternatives are available. Custom solutions are expensive and slow to build, difficult to maintain, and quick to become outdated. They don’t take advantage of developments in the industry, and the eventual cost is high.**

* Internal devs are most content to build core products.
* Tools that are not core business rarely get full attention.
* Everything that is not business logic or user interaction is not your core business.
* Every off-the-shelf product is a core business for the company or the community that makes it.
* Cloud native ecosystem is growing very fast.
* Many engineers think “they know better.”
* Open source attracts many devs.

**Therefore**

**Use existing tools whenever possible, even when the fit isn’t perfect.**

Whether commercial or open source, existing products are typically better quality, better maintained, and more frequently extended than anything you can build yourself. Spend most of the development time on your core business functionality. This will significantly increase the time and effort available for investment into the core business parts that separate your company from the competition, while making sure that the rest of the components are easily maintainable and up to the latest industry standards.

* Make use of third-party libraries, off-the-shelf products, and existing architectures when possible.
* Focus your internal resources on delivering your core business.
* Build only if nothing else is available; give preference to an open source solution unless it’s related to your core business.
* Seek the fullest possible solution; trying to fit together a variety of open source solutions, each addressing a separate business function, can lead to maintaining a complex environment of many moving parts.

**Consequently**

**The team can focus on core business.**

* + New functionality is constantly introduced with third-party product releases.
* + Quality of off-the-shelf products is typically higher.
* + There is external product/user support.
* + Easier to hire people when using common tools.
* − Some problems are too specific for any off-the-shelf solution to address.
* − Third-party products are often expensive.
* − Less control over functionality.

**Common Pitfalls**

Underestimating the cost of building and maintaining your own custom solutions. In the beginning it looks quick and easy, but developers are typically too optimistic in their estimates. Building your own tool always ends up being a long, difficult, and expensive initiative—much bigger than estimated in the beginning.

**Related Biases**

*Illusion of control*

Engineers think they know best what the company needs and will be able to build a better solution than what is available from outside vendors.

*Planning fallacy*

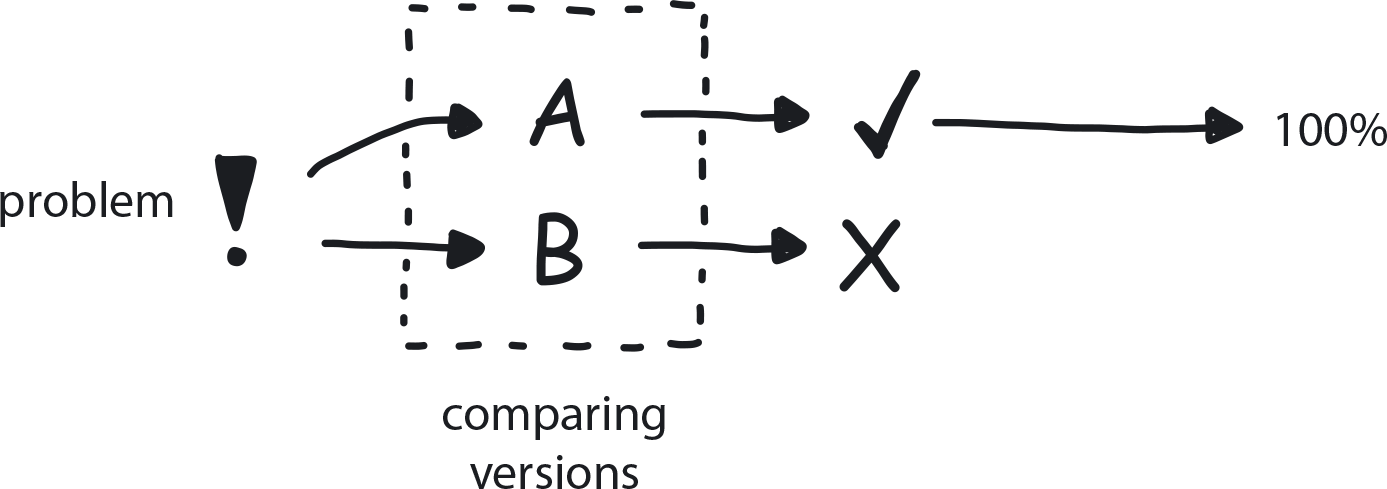
The human tendency to underestimate the time a project will require.

**Related Patterns**

* [Open Source Internal Projects](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#open_source_internal_projects)
* [Exit Strategy Over Vendor Lock-in](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch07.html#exit_strategy_over_vendor_lock_in)
* [Public Cloud](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#public_cloud)

**Pattern: A/B Testing**

*Comparing multiple versions of something (a feature, new functionality, UI, etc.) under real customer use conditions quickly gives useful data about which performs better (*[*Figure 9-17*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#a_solidus_b_testing01)*).*



**Figure 9-17. A/B Testing**

A company has a working cloud native infrastructure in place and is aiming to deliver a lot of useful functionality to its customers. Teams are proficient, and all the tech and processes are in place.

**In This Context**

**There is no practical way to predict how customers will respond to changes. In the absence of actual customer usage data, design and implementation decisions must be based on guesswork and intuition. Since our intuition is not perfect and full of biases, we may not get the best possible results.**

* People don’t always know what they need/want.
* Delivering fast without adjusting the product based on measurement and feedback will change nothing.
* It’s impossible to make logical decisions in an unknown environment.
* There are unlimited variations and combinations of possible solutions.

**Therefore**

**Prepare multiple versions of a solution to a challenge/problem and present them to randomized small portions of the client base. Measure the customer response in terms of value for them, and based on that choose the solution.**

* Famous Google example of testing 41 shades of blue for its toolbar[**1**](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#ch09fn1) to find which inspired the most consumer clicks—because, for Google, clicks equal revenue.
* The Obama campaign raised more money by using A/B testing to choose the most effective messaging.
* Need to provide businesspeople with the opportunity to run the A/B test experiments themselves, and an accessible way for them to do so.

**Consequently**

**You now have an easy way to test assumptions live in a real-world environment.**

Instead of making guesses or assumptions regarding which of two ideas, implementation strategies, etc., is better, a team can quickly put together a simple prototype with two or more versions and release them to small subsets of real customers. Based on customer response, the team can choose the more appropriate/preferred solution. This way many options could be tested while costs are ultimately saved because only the best option is ever fully implemented.

* + Customers see response to their needs.
* + If something doesn’t work, you can easily roll back to previous version.
* − Human insight might be sidelined when user response data is followed blindly.
* − Some innovative solutions take time to gain customer acceptance; A/B testing brings a risk of prematurely canceling such products.

**Common Pitfalls**

Taking a sequential approach by using an outcome as the basis for the next A/B test comparison. This “winner take all” approach risks premature elimination of the “‘losing” variable, which could be the better choice in a different context. You can’t know until you test this, of course! But by automatically rejecting the “losing” variable after each comparison, you lose the opportunity to experiment further with that variable. The solution is multivariate testing—instead of A/B comparisons, you have A/B/C or even A/B/C/D combinations of variables randomly exposed to different test groups.

**Related Biases**

*Confirmation bias*

Picking test variables that will “prove” your existing opinion, rather than genuinely exploring alternatives.

*Congruence bias*

If you have a preconceived outcome in mind and test results that confirm this, you stop testing instead of seeking further information.

*Information bias*

The tendency to seek information even when it cannot affect the outcome; in A/B testing this would lead to choosing meaningless variables to test.

*Parkinson’s law of triviality/“Bikeshedding”*

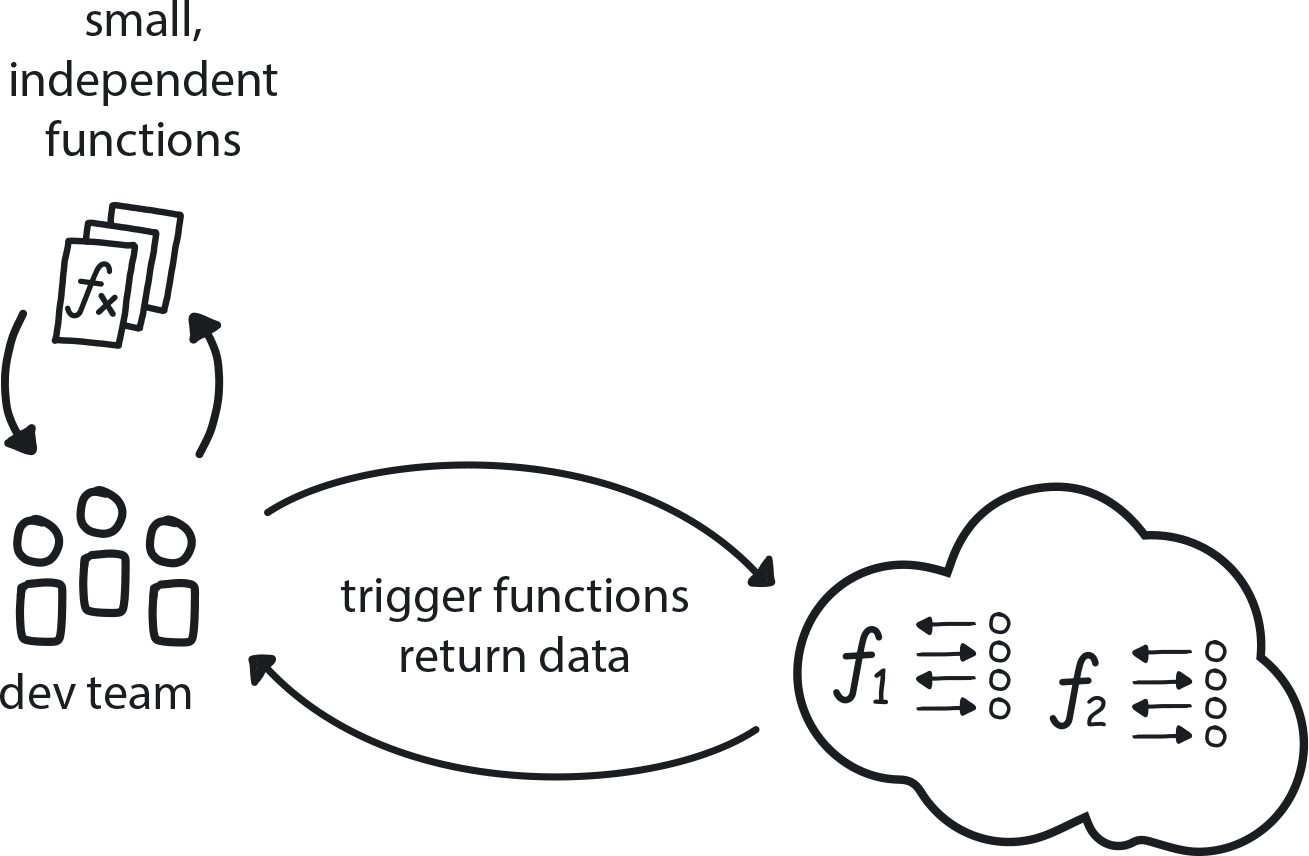
Choosing something easy but unimportant to test and evaluate over something else that is complex and difficult but meaningful.

**Related Patterns**

* [Measure What Matters](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch07.html#measure_what_matters)
* [Observability](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#observability)
* [Learning Loop](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch07.html#learning_loop)
* [Involve the Business](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch07.html#involve_the_business)

**Pattern: Serverless**

*The soon-to-arrive future is event-driven, instantaneously scalable services (functions) on the cloud (*[*Figure 9-18*](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#serverless01)*).*



**Figure 9-18. Serverless**

The team is building a highly scalable system or using tools that require integration maintenance tasks. Tasks are well-defined and repeatable and may require aggressive scaling for short bursts.

**In This Context**

**There are a lot of small tasks that can eat up a developer’s time: writing boilerplate code, setting up infrastructure, and of course later maintaining everything they created. Meanwhile, it’s very challenging to create scaling mechanisms that can respond in milliseconds. The first set of challenges leads to wasted developer effort plus extra costs related to setup and maintenance. The second typically results in over-provisioning of compute resources that have to be paid for whether they’re used every day or only on Black Friday.**

* Serverless is a recent and somewhat still emerging execution model for cloud computing.
* Some applications may change hardware requirements quickly and dramatically.
* Scaling up and down manually is difficult.
* Software components (microservices) become smaller all the time.
* Maintaining servers and/or container schedulers is an expensive task.

**Therefore**

**Package small pieces of code into fully independent executable functions that can be individually triggered on a serverless platform. Functions get one input source and return one output. Any number of functions can be executed in parallel. Functions are self-contained and repeatable.**

Thought leaders and experts in distributed systems believe serverless technologies are the next evolution of application infrastructure—the horizon that lies beyond microservices. Serverless architecture is “serverless” in that users never need to take care of, or even ever really think about, individual machines: infrastructure is fully abstracted away. Instead, developers simply pick from a nearly limitless menu of compute, network, and storage resources via managed services from public cloud providers. Serverless is truly pay-as-you-go, calculated according to actual real-time consumption instead of pre-purchased services based on best guesswork. While this makes for cost-efficient application development, the true benefit is velocity: developers finally get to focus on writing code instead of managing servers or sharding databases.

Currently there are many challenges to serverless adoption, such as operational control, the introduction of even greater complexity into (already highly complex) distributed systems, and effective monitoring.

For now this falls under the H2/innovation and H3/research categories in the Three Horizons pattern, but some companies on the leading edge of cloud native have already embraced serverless. Those able to dedicate skilled engineers to conquering its current challenges are able to dramatically reduce operational overhead and streamline the DevOps cycle even further, while increasing scalability and resilience.

Think of Serverless as basically like cloud native, with superpowers.

* Functions only consume resources while running.
* Very short startup time.
* Highly scalable.
* Cheap to use.

**Consequently**

**Some software tasks can be executed very fast at any scale; the rest of the system is containerized.**

* + Some tools/tasks are running in functions.
* + Running a function requires almost zero overhead.
* + Developers will never have to think about provisioning infrastructure ever again.
* − Creating a full serverless app is difficult due to current architectural limitations.

**Common Pitfalls**

Trying to achieve an advanced, and still emerging, technology like serverless before you have even established an initial solid cloud native platform based on current tech that is well-supported. Get good at orchestrating containerized microservice applications first, then worry about next steps.

**Related Biases**

*Bandwagon effect*

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

**Related Patterns**

* [Microservices Architecture](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#microservices_architecture)
* [Containerized Apps](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#containerized_apps)
* [Distributed Systems](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#distributed_systems)
* [Three Horizons](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch07.html#three_horizons_pattern)
* [Public Cloud](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#public_cloud)

**Summary**

In this chapter we introduced patterns around cloud native development and processes. 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 infrastructure—functions as a more in-depth resource for referencing and working with individual patterns.