**Chapter 5. Knowing Thyself: The Cloud Native Maturity Matrix Tool**

Cloud native is no longer the exclusive domain of massive tech-forward pioneers like Google, Uber, and Netflix. Cloud native tools and tech have matured to the point where the vast benefits of going cloud native have become available to companies of any size, from every sector. You are likely reading this book because you realize that cloud native’s very tangible benefits—the ability to iterate and produce results quickly without getting bogged down in infrastructure setup and maintenance—are now realistically within your grasp.

Having read this far, however, means you’ve also gotten the news that achieving a cloud native transformation is not as simple or as easy to do as those peppy conference presenters like to make it sound. Transformation patterns certainly help facilitate the process, but that requires awareness and understanding of your current organizational context in order to know which patterns apply, and in what order.

So how *do* you go about launching a successful migration, patterns and all?

The first crucial step, as we saw in [Chapter 2](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch02.html#the_human_challenge_of_cloud_native), is Know Thyself. This means truly understanding your company’s existing architecture, processes, and organizational culture. You must evaluate these crucial but often overlooked realities before you can tackle the technical aspects of a migration—that is, if you want a successful outcome. Enterprises that just grab for the new tech without first understanding how it will fit into and function within their existing organizational culture (spoiler alert: it won’t) are setting themselves up for wasting time and resources on an ultimately ineffective effort. Or, worse, complete failure.

**Mirror, Mirror, on the Wall…**

Realistically, this is easier said than done. It’s hard to objectively observe and evaluate your own systems and culture from the inside as an active participant. Honestly it’s not the easiest thing to do from the outside, either. Take the many interwoven forces and factors at play and then combine those with the unique circumstances any enterprise brings to a migration: it’s a complex tangle.

Cloud service providers are no help in this area. Amazon Web Services, Google Cloud Services, Azure, et al., are doing their level best to anticipate and solve every onboarding problem a business may encounter, but thus far none of them offers any help with the crucial first step (i.e., that “Know Thyself” task). They don’t even acknowledge its existence. Why would they? Their business model centers upon getting you to sign up and use their systems, not analyzing your own existing one.

Thus, the temptation to just pick a provider and get on with the show is utterly understandable. In the early stages it will also likely feel like a successful move. However, the complexity inherent within cloud native’s distributed systems architecture is relentlessly exponential. An “unexamined” organization will inevitably reach a point where its existing systems and culture will clash with—and, ultimately, short-circuit—its transition attempt.

Though still new and evolving, cloud native has been around long enough for those of us in the sector to observe and identify common elements appearing in successfully completed migrations, as well as frequent points when attempts stall or even fail outright. Having cataloged some things that help drive a successful migration, and some that definitely do not, the next logical step was to create an assessment tool.

**For Mature Audiences Only: The Maturity Matrix**

Maturity models can be a useful and effective mechanism for evaluating status in all kinds of systems. When it comes to software development, unfortunately, traditional maturity models often lack context and/or fail to provide pragmatic guidance for undertaking their recommended steps. Even when they do try to help with actual implementation, maturity models too often either (a) oversimplify reality or (b) prescribe a one-size-fits-all progression path.

Since every organization has unique circumstances, challenges, and needs, we developed a detailed and accurate yet flexible and, above all, pragmatic version for organizations seeking a cloud native transformation.

We distilled four years’ worth of observation and experience gained from helping clients onto the cloud and used it to build this assessment tool. We call it the Cloud Native Maturity Matrix, a unique framework for evaluating and understanding your company where it is right now. The matrix is a pragmatic process you can apply to map your current maturity, identify gap analysis, and discern where they should focus their efforts in order to reap the most benefits. This is a model which has proved very useful for our clients.

Organizations tend to call upon cloud consultancy services in one of two ways. Helping a company commence a brand-new cloud migration and getting it right from the start is our preferred mode. But, as is happening more and more frequently, we are also called in to rescue stalled implementations. Either way, our approach starts the same with each and every client.

We spend two days on site evaluating their unique circumstances and specific needs by taking them through the Cloud Native Maturity Matrix. Together, we create an accurate snapshot of the enterprise along nine different axes. We then use this to define, analyze, and describe organizational status and begin to map the migration process. This data—constantly re-assessed as the process moves forward—allows us to customize transformation goals and monitor progress while working to keep all the different aspects involved in a migration moving forward smoothly and in alignment.

In other words, the Maturity Matrix is how we create the custom map for each company’s unique migratory path to the cloud. And it’s also how we monitor the process to remain on track.

The concepts that form the Maturity Matrix framework are essential background knowledge for talking about cloud native transformation patterns. Understanding what we assess, and why, is key to understanding your own global organizational context—so you can identify the applicable patterns and use them properly for your own transformation design.

**Staying in Sync**

In this chapter we are going to walk you through the nine different areas on the Maturity Matrix and how to identify your organization’s status in each one. These areas are:

* **Culture:** The way individuals in your organization interact with one another
* **Product/Service Design:** How decisions are made within your organization about what work to do next (e.g., which new products to build or what new features to add or improvements to make to existing ones)
* **Team:** How responsibilities, communication, and collaboration works across and between teams in your organization
* **Process:** How your organization handles the execution of work and assigned projects
* **Architecture:** Describes the overall structure of your technology system
* **Maintenance and Operations:** How software is deployed and then run in a production environment in your organization
* **Delivery:** How and when software from your development teams gets to run in your live (production) environment
* **Provisioning:** The processes by which you create or update your systems in your live production environment
* **Infrastructure:** The physical servers or instances that your production environment consists of—what they are, where they are, and how they are managed

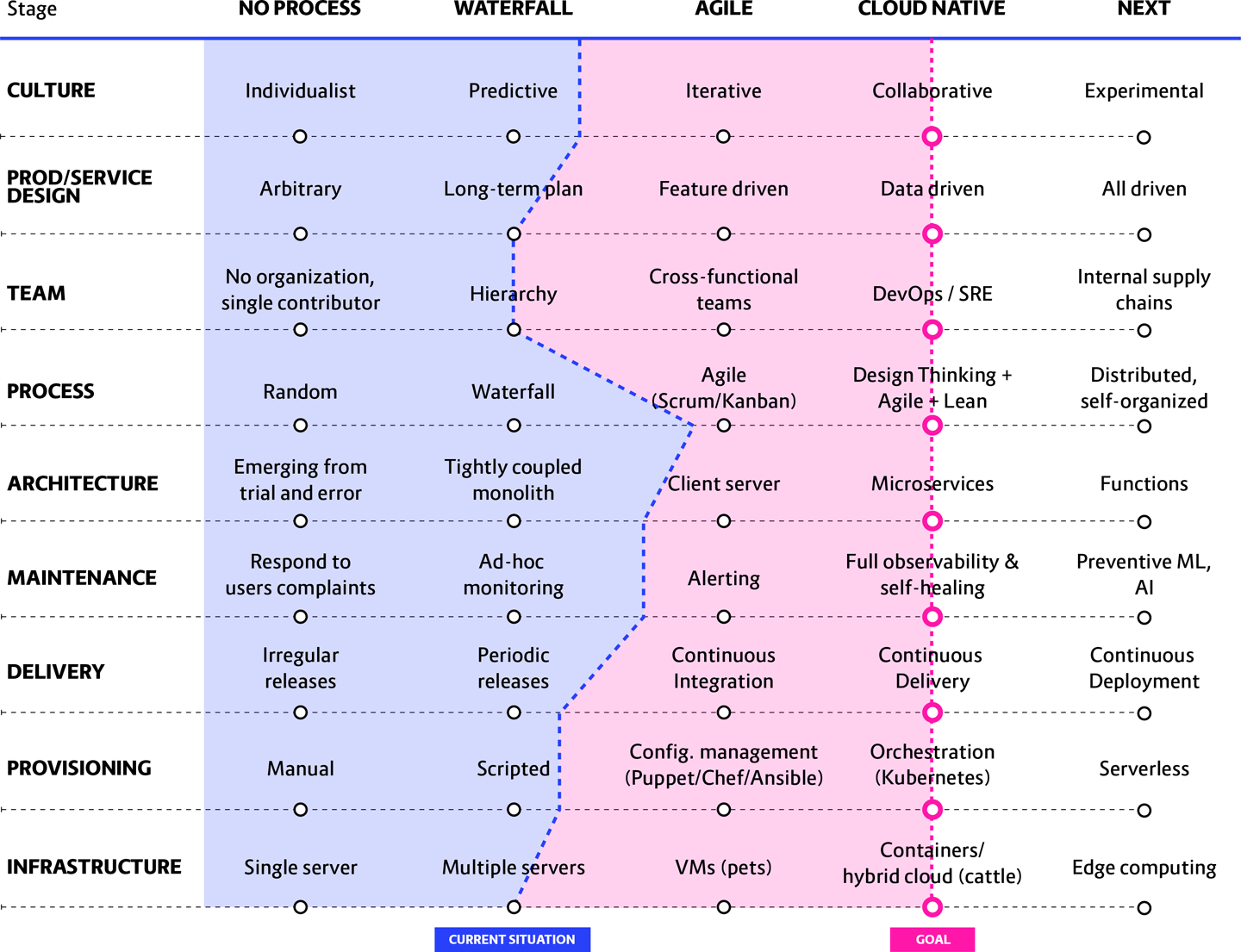
Once assessed, each area’s status is mapped on the corresponding Maturity Matrix axis. We then literally “connect the dots” by drawing a line through each area’s current status point, from Culture all the way to Infrastructure.

Graphing status in this way provides an instant, powerful, and above all easy to grasp visual representation of your company’s state. It also clearly demonstrates the distance that we will need to close in each area in order to achieve full cloud native functioning.

[Figure 5-1](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch05.html#cloud_native_maturity_matrix_results) shows Maturity Matrix results from a real-world client assessment.

In the sample Maturity Matrix in [Figure 5-1](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch05.html#cloud_native_maturity_matrix_results), we see both the current situation for an actual company, as well as the progression points necessary for a cloud native transformation. For example, Culture has progressed somewhat beyond Waterfall, while Process has nearly reached Agile.

This allows us to identify the worst bottleneck (i.e., the least developed areas) and focus our initial migration efforts there, so as to immediately begin to increase the flow. In this example matrix, we would be looking at Infrastructure, Product/Service Design, and Teams as first priorities.



**Figure 5-1. Cloud Native Maturity Matrix results from an enterprise assessment/discovery, with the cloud native “goal line” defined**

This, however, does not mean that other areas stay on hold while one or more bottlenecks are addressed. It’s OK for different teams to progress at different rates—especially if some of these teams are preparing the ground for an easier overall transition.

It is important to note that a company doesn’t need to go through intermediate stages in order to reach cloud native—that is, if they are in Waterfall, they can jump directly to cloud native without going through Agile first.

Transitions progress gradually, and different teams move forward at different rates. Aligning the Maturity Matrix does not mean moving in lockstep to maintain some inflexible, perfectly even line during transition. It’s more about staying in sync: making sure that each of the axes is adequately and appropriately addressed, working together holistically and in context with the entire complex system.

**Applying the Matrix**

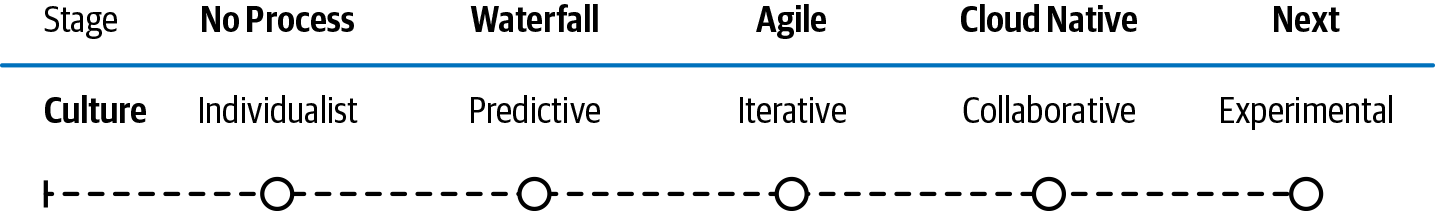
The Cloud Native Maturity Matrix is typically administered by trained facilitators over the course of a few days spent on-site with an enterprise and its employees. However, it is still an extremely useful thought experiment to work through it on your own to try to identify where your organization currently falls.

As we have seen, the Maturity Matrix is divided into nine separate areas (or axes), each one an individual and essential member of an integrated, interdependent system. Each axis is further divided into four specific stages of organizational development a company may currently occupy: no process, Waterfall, Agile, and cloud native. (An additional “Next” category is also included to show possible directions that could happen in the future, given current trends and tech developments.) Over our next few sections we will examine what exactly organizational status typically looks like in each of these same query stages, compared across each of the nine axes. We move across them from left to right, from older/less agile states in a progression toward resilient and responsive cloud native. But the Cloud Native Maturity Matrix does not end with a successful migration onto the cloud! As we’ve discussed, cloud native is not only focused on what to do now—it is just as much about building in the ability to easily adapt to whatever comes *next.*

**Culture**

The Maturity Matrix begins with Culture because it is the toughest transition axis to progress—no matter the organization. Culture is abstract, hard to transform, and evolving it is a slow process. The other axes are faster and easier to achieve because, ultimately, they are mainly code and planning. Changing culture also requires a lot of buy-in across the entire organization, while the other axes can generally function in a more independent way.

We discussed culture in depth in [Chapter 2](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch02.html#the_human_challenge_of_cloud_native), but here is a quick overview. [Figure 5-2](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch05.html#culture_axis_of_the_cloud) shows the range of Culture indicators we investigate in a Maturity Matrix assessment.



**Figure 5-2. Culture axis of the Cloud Native Maturity Matrix**

**No process:** Individualistic

There is no set or specified way to interact with peers, superiors, or subordinates. Instead, communications are rooted in personal preferences. This is a common culture for startups but is unsustainable as you scale up.

**Waterfall:** Predictive

A Predictive organization has a strong preference for long-term planning and firm deadlines. The goal is to deliver a complex system exactly as specified; delivering it fast is not a priority. These organizations tend to suppress experimentation or introducing new ideas because these are inherently unpredictable. Typically there are large amounts of documentation; procedures for changes, improvements, and daily tasks; segregation of teams by specialization; tools for every situation; and regular (e.g., weekly), lengthy planning meetings. Delivering a complex system exactly as specified and on time is a complex and difficult endeavor.

This culture is common in medium-to-large enterprises.

**Agile:** Iterative

An Agile organization chooses smaller and simpler goals, which it aims to deliver as fast as possible. Agile organizations tend to focus on the short term rather than following a long-term plan. Communication is often by short, daily meetings. Emphasis is on fast responses and quick fixes, which can lead to a “hero culture” where individuals regularly display superhuman efforts to keep everything on track. They commonly use the Scrum project management methodology, with inter-team communication by Scrum Masters and other coordinators. Agile organizations normally have wide responsibilities *within* cross-functional teams but narrow responsibilities *for* each team.

This culture is common throughout startups and enterprises of all sizes.

**Cloud native:** Collaborative

A **Collaborative** organization tends to have big but not deeply specific goals (i.e., there may be a wide vision but without a detailed specification or a fixed delivery date). This culture embraces learning and consistent, continuous improvement over predictability. Emphasis is on self-education, experimentation, and research. Results are coldly assessed based on field data.

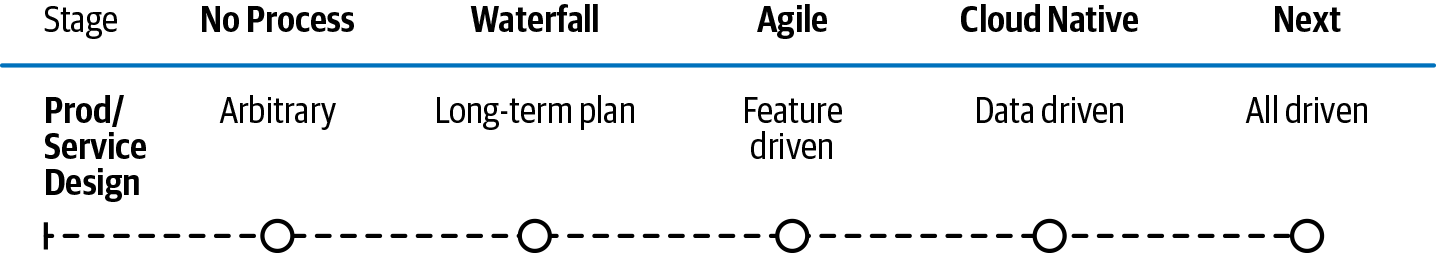
A collaborative culture is crucial for companies operating in areas of high uncertainty or fast change.

**Next:** Generative

We predict the next type of organization will be a **Generative** one. An extension of a collaborative organization, in a generative organization IT will co-create solutions as equal partners with the business.

**Product/Service Design**

This is the place where we assess just what it is you do and how you go about doing it. We evaluate whether you are organized around long-term planning, delivering a tightly coupled product on a slow and deliberate schedule—or whether you iterate rapidly in shorter sprints, ideally using customer feedback to drive the changes. [Figure 5-3](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch05.html#product_solidus_service_design_axis) shows the range of Product/Service Design situations we look for in a Maturity Matrix assessment.



**Figure 5-3. Product/Service Design axis of the Cloud Native Maturity Matrix**

**No process:** Arbitrary

An arbitrary design process is fad/wild-idea driven, somewhat random, and not deeply discussed. It is a common way to operate in startups where ideas usually come from the founders. On the upside, it can be highly creative. On the downside, it may result in partial features or an incoherent product.

**Waterfall:** Long-term plan

This design process focuses on collating and assessing product feature requests by customers, potential customers (via sales), users, or product managers. Individual features are then turned into team projects and multiple features are combined into large releases that happen every six to twelve months. This process is a very common model for larger enterprises.

**Agile:** Feature Driven

A feature-driven design process speeds things up by allowing small new features to be selected with less planning. The aim is that these more modest features will be delivered to clients every few weeks or months in small batches. A feature-driven organization focuses on fast change often without an overarching long-term plan.

**Cloud native:** Data Driven

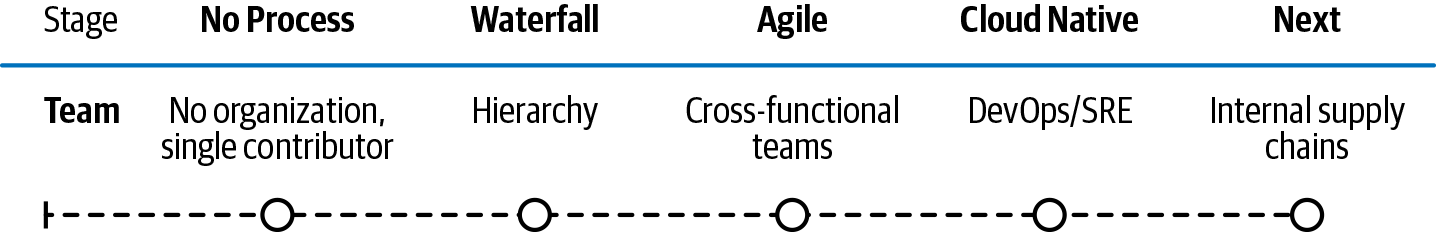
The final say on which features stay in a product is based on data collected from real users. Potential new features are chosen based on client requests or designs by product owners without a long selection process. They are rapidly prototyped and then developed and delivered to users with copious monitoring and instrumentation. They are assessed against the previous features (better or worse?) based on A/B or multivariate testing. If the new feature performs better, it stays; if worse, it is switched off or improved.

**Next:** AI Driven

In the future, humans will be cut out of this process entirely! AI-driven systems will make evolutionary tweaks and test themselves with little developer interaction.

**Team**

Does your enterprise take a top-down, “Do what the boss says” approach, likely with highly specialized teams? Or one that is more cross-functional, composed of teams where each member has specific skills? Possibly you have progressed all the way to DevOps—an effective approach that takes advantage of cloud native architecture. [Figure 5-4](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch05.html#team_axis_of_the_cloud) shows the range of Team structures we look for in a Maturity Matrix assessment.



**Figure 5-4. Team axis of the Cloud Native Maturity Matrix**

**No process:** No organization, single contributor

In this type of organization we find little structure, typically one or possibly a few independent contributors with no consistent management. This is most commonly found in small startups.

**Waterfall:** Hierarchy

Organized via considerable top-down order, both within and between the teams. Decisions are made by upper managers, and implementation is done by specialized teams (making it difficult to move individuals between teams). There will be separate teams of architects, designers, developers, testers, and operations. Inter-team communication is generally through tools like JIRA or via managers. Historically, this has been the most common structure of large organizations.

**Agile:** Cross-functional teams

A cross-functional organization has less specialization across teams and more cross-capability *within* teams. For example, development teams will often include testing and planning capabilities. Scrum Masters, Product Owners, etc., facilitate communication between teams. However, a hierarchical organizational structure remains outside the teams themselves.

**Cloud native:** DevOps/SRE

Traditionally, developers/engineers have been responsible for building software and then handing it off to the operations team for deployment. A DevOps team joins the two in a single team capable of designing and building applications as part of a distributed system, and also operating the production platform/tools. Across the organization, each team has full responsibility for delivering an individual set of microservices and supporting them. DevOps teams typically include planning, architecture, testing, dev, and operational capabilities.

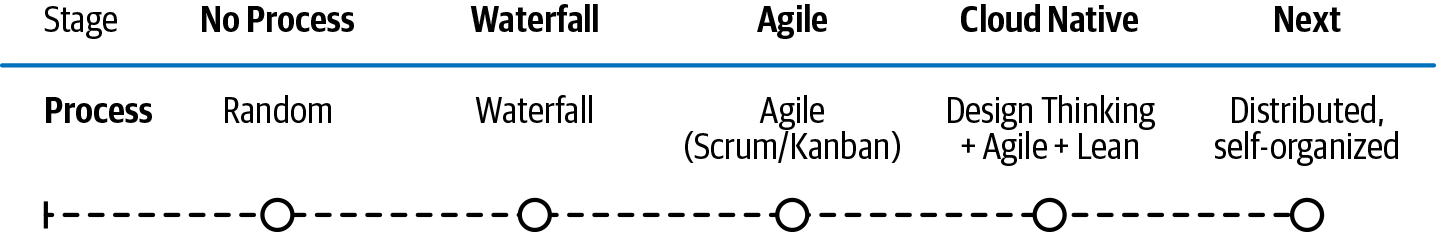
There will still often remain a separation of tasks. For example, it is common to see a platform DevOps team in charge of building the cloud native platform, while site reliability engineering (SRE) or first-level support teams respond to incidents (and spend the rest of their time working on automation to prevent them from happening in the first place). However, there is considerable collaboration between those teams and individuals can easily move between them.

**Next:** Internal supply chains

In an Internal Supply Chain organization, each service is a separate product with full tech and business generation responsibilities in the teams—much as many ecommerce teams have been managed for a decade.

**Process**

Does your enterprise do long-term planning up front and then follow with execution? Or do you change things responsively and on the fly? Currently, Scrum/Kanban is what we find most enterprises using. Cloud native and CI/CD require the next jump in speed: now developers need to be able to deliver every day—and do so independently from other developers. [Figure 5-5](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch05.html#process_axis_of_the_cloud) shows the range of process approaches we look for in a Maturity Matrix assessment.



**Figure 5-5. Process axis of the Cloud Native Maturity Matrix**

**No process:** Random

In a random organization there is no change-management process, just random changes made at will. There is often no consistent versioning. This is common in many small companies with only a couple of engineers.

**Waterfall:** Waterfall

In a Waterfall organization, the product development process is tightly controlled through up-front planning and change management processes. A sequential process is followed of planning, execution, testing, and (finally) delivery. There is usually an Integration stage before delivery where work from different streams is combined.

The process is run by managers; every handover is well documented and requires forms and procedures.

**Agile:** Agile (Scrum/Kanban)

Product development is run in sprints using an Agile technique such as Scrum or Kanban. Documentation is limited (the product is the documentation), and teams are heavily involved in their own management through daily consultation. There is usually considerable pressure to deliver fast and no defined provision for experiments or research. Limited or no changes are allowed during sprints to protect the delivery deadlines.

**Cloud native:** Design Thinking + Agile + Lean

Design Thinking and other research and experimentation techniques are used for de-risking large and complex projects. Many proofs of concept (PoCs) are developed to compare options. Kanban is often then used to clarify the project further, and finally Agile methods like Scrum can be applied once the project is well understood by the entire team. Highly proficient organizations might choose to follow the Lean model.

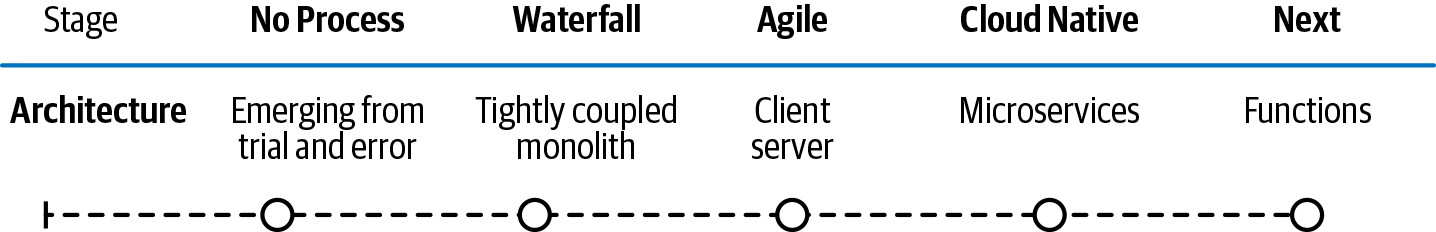
This relatively new approach is very effective in situations of high uncertainty or where the technology is changing rapidly.

**Next:** Distributed, self-organized

In the future, self-organized systems will be highly experimental. There will be less up-front design. Individuals or small teams will generate ideas, which will then form the seeds of a new product or feature. Once implemented, these will be iterated and improved on automatically by the platform.

**Architecture**

Is your enterprise trying “batteries included” to provide everything needed for most use cases—the Tightly Coupled Monolith? Or perhaps you have reached the next step in the evolutionary architecture chain, Client–Server. The cloud native goal is to use microservices architecture where a large application is built as a suite of modular components or services. Microservices enable development teams to take a more decentralized (non-hierarchical) approach to building software. Microservices enable each service to be isolated, rebuilt, redeployed and managed independently. [Figure 5-6](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch05.html#architecture_axis_of_the_cloud) shows the range of Architecture approaches we look for in a Maturity Matrix assessment.



**Figure 5-6. Architecture axis of the Cloud Native Maturity Matrix**

**No process:** Emerging from trial and error

In an architecture described as emerging from trial and error, there are no clear architectural principles or practices. Developers just write code independently, and all system-level communication is ad-hoc. Integrations between components tend to be poorly documented, unclear, and hard to extend and maintain.

**Waterfall:** Tightly coupled monolith

A tightly coupled monolith is an architectural model where the entire codebase is built as one to five modules, with many developers working on the same components. A layered architecture (database, business logic, presentation layer, etc.) is common. Although interfaces have been defined, changes in one part often require changes in other parts because, typically, the code is divided into components with very strong coupling.

Delivery is done in a coordinated way, all together, and typically the monolith is written in a single programming language with strong standardization on tooling. The application is usually vertically scalable (you can support more users by adding more resources on single server). The design and maintenance of the monolith is usually led by a system architect or her team, many of whom are not hands-on developers.

**Agile:** Client–server

This architecture is the most basic form of distributed system. The client–server model partitions tasks or workloads between service providers—the servers—which deliver requested resources to the service-seeking clients.

Like a monolith, in a client–server architecture multiple teams work on services at once, and all services need to be deployed together. However, because the network-induced separation provides a degree of decoupling, it is usually possible to at least some degree to develop on the system in parallel (one group handles the client part, one the server).

**Cloud native:** Microservices

Microservices architecture is highly distributed. It comprises a large number (usually more than 10) of independent services that communicate only via well-defined, versioned APIs. Often, each microservice is developed and maintained by one team. Each microservice can be deployed independently, and each has a separate code repository. Hence, each microservice team can work and deploy in a highly parallel fashion, using their own preferred languages and operational tools and datastores (such as databases or queues).

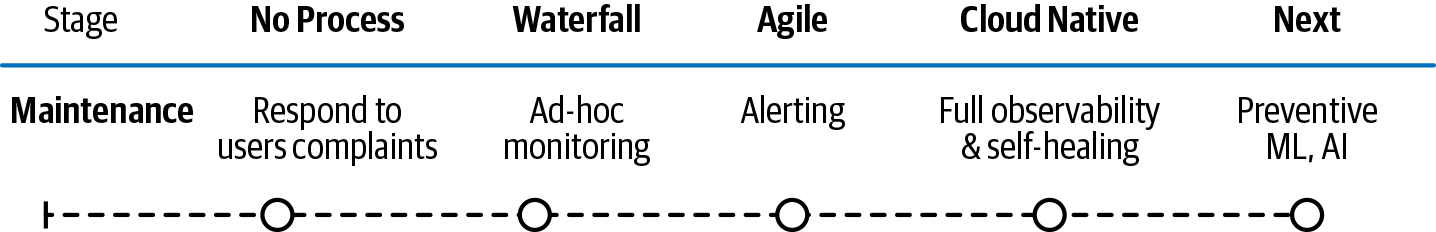
Because the system is distributed and components are decoupled, not only from each other but from other copies of themselves, is it easy to scale the system up by deploying more copies of each service. Operationally, microservice deployment must be managed in a fully automated way.

**Next:** Functions-as-a-Service/Serverless

A Functions-as-a-Service (FaaS, also known as Serverless) architecture is one where no infrastructure needs to be provisioned. Each piece of business logic is in separate function, which is operated by a fully managed Function-as-a-Service such as AWS’ Lambda, Azure Functions, or Google’s Cloud Functions. No operations tasks such as up-front provisioning, scaling, or patching are required. There is a pay-as-you-go/pay-per-invocation model.

**Maintenance**

On this axis we assess how you monitor your systems and keep them running. It’s a broad spectrum, from having no process whatsoever to full automation with little or no human intervention. No Process/Ad Hoc means every now and then going in to see if the server is up and what the response time is. (And the somewhat embarrassing fact is, a lot of folks still do just that.) Alerting means having some form of automation to warn when problems arrive, but it is nowhere near fast enough for this new world because once a problem is alerted, a human being still needs to intervene. Comprehensive monitoring and full observability, where system behavior is observed and analyzed so problems can be predicted (and prevented) in advance, rather than responded to when they do happen, are an absolute necessity for cloud native. [Figure 5-7](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch05.html#maintenance_axis_of_the_cloud) shows the range of Maintenance approaches we look for in a Maturity Matrix assessment.



**Figure 5-7. Maintenance axis of the Cloud Native Maturity Matrix**

**No process:** Respond to users’ complaints

The development and operations teams are alerted to most problems only when users encounter them. There is insufficient monitoring to flag issues in advance and allow engineers to fix them before the majority of users will hit them. System downtime may only be discovered by clients, or randomly. There is no alerting.

For diagnosing issues, administrators usually need to log in to servers and view each tool/app log separately. As a result, multiple individuals need security access to production. When fixes to systems are applied, there is a manual upgrade procedure.

This is a common situation in startups or small enterprises, but it has significant security, reliability, and resilience issues, as well as single points of failure (often individual engineers).

**Waterfall:** Ad-hoc monitoring

This consists of partial, and mostly manual, monitoring of system infrastructure and apps. This includes constant monitoring and alerting on basic, fundamental downtime events such as the main server becoming unresponsive.

Live problems are generally handled by the operations team and only they have access to production. Usually, there’s no central access to logs, and engineers must log in to individual servers for diagnosis, maintenance operations, and troubleshooting. Formal runbooks (documentation) and checklists exist for performing manual update procedures; this is very common in larger enterprises but still does not completely mitigate security, reliability, and resilience issues.

**Agile:** Alerting

Alerts are preconfigured on a variety of live system events. There is typically some log collection in a central location, but most of the logs are still in separate places.

Operations teams normally respond to these alerts and will escalate to developers if they can’t resolve the issue. Operations engineers still need to be able to log in to individual servers. Update processes, however, may be partially or fully scripted.

**Cloud native:** Full observability and self-healing

In full observability and self-healing scenarios, the system relies upon logging, tracing, alerting, and metrics to continually collect information about all the running services in a system. In cloud native you must observe the system to see what is going on. Monitoring is how we see this information; observability describes the property we architect into a system so that we are able to discern internal states through monitoring external outputs. Many issue responses happen automatically; for example, system health checks may trigger automatic restarts if failure is detected. Alternatively, the system may gradually degrade its own service to keep itself alive if, for example, resource shortages such as low disk space are detected (Netflix is famous for this). Status dashboards are often accessible to everyone in the business so that they can check the availability of the services.

Operations (sometimes now referred to as “platform”) engineers respond to infrastructure and platform issues that are not handled automatically. Live application issues are handled by development teams or system reliability engineers (SREs). The SRE role may be filled by individuals embedded in a DevOps team or separated into a dedicated SRE team.

Logs are all collected into a single place. This often includes distributed tracing output. Operations, developers, and SREs all have access to the logging location. They no longer have (or need) security access to production servers.

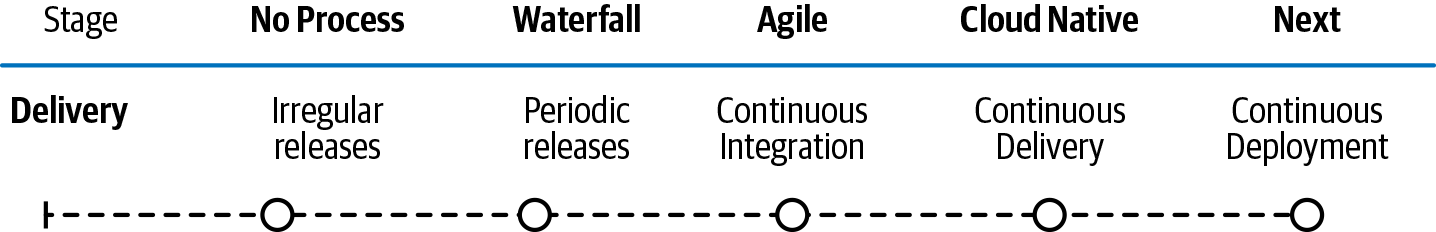
All update processes are fully automated and do not require access by individual engineers to individual servers.

**Next:** Machine learning (ML) and artificial intelligence (AI)

In the next generation of systems, ML and AI will handle operational and maintenance processes. Systems learn on their own how to prevent failures by, for instance, automatically scaling up capacity. Self-healing is the optimal way for systems to be operated and maintained. It is faster, more secure, and more reliable.

**Delivery**

Delivery is really all about how quickly you can get things out and in how automated a fashion. The Maturity Matrix moves from traditional major version releases every six to 12 months to Agile’s more rapid iterations of weekly to monthly releases. Reaching cloud native grants the ability to release daily, or even multiple times per day. [Figure 5-8](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch05.html#delivery_axis_of_the_cloud) shows the range of delivery approaches we look for in a Maturity Matrix assessment.



**Figure 5-8. Delivery axis of the Cloud Native Maturity Matrix**

**No process:** Irregular releases

In many small organizations, irregular software releases (new function or fixes) are delivered into production at random times based on IT or management decisions about the urgency of the change. For highly urgent issues, like fixes for production problems, changes are delivered by developers directly to production ASAP.

This is a common situation for startups and small enterprises.

**Waterfall:** Periodic scheduled releases

Many organizations have periodic scheduled releases, for example every six months. The contents of these (usually infrequent) releases becomes extremely important and is the result of long planning sessions. Extensive architectural documents for each release are produced by enterprise architects; no coding is done before the full architecture is ready. Once the release contents are agreed on, any change is subject to a change approval board (CAB). A key driver behind infrequent releases is the need to perform expensive manual testing of each release prior to deployment.

Highly sequential processes are followed for each release:

1. System and software requirements are captured in a product requirements document.
2. Analysis is performed, resulting in documented models, schema, and business rules.
3. Design of the software architecture is completed and documented.
4. Coding is done: the development, proving, and integration of software (i.e., merging the work done by different teams).
5. Testing of that integrated new code is performed, including manual tests.
6. The installation and migration of the software is completed by the operations team.

After the release, the Operations teams support and maintain the completed system.

**Agile:** Continuous Integration (CI)

Continuous integration describes an organization that ensures new functionality is ready to be released at will—without needing to follow a strict release schedule (although a formal release schedule may still be followed). It often results in more frequent releases of new code to production.

A tech organization using CI typically has:

* A single codebase (aka source repository) that all developers add their code to. This ensures that merging and integration happen constantly rather than occasionally. That tends to make merging much easier.
* A fully automated build process that turns new code into runnable applications.
* Automated testing of all code as part of the build. This forces developers to fix bugs as they go along (which, again, is easier).
* A requirement for developers to add their new code to the single repository every day, which forces them to merge and fix bugs incrementally as they go along.
* A way to deploy code to test or production hardware in an automated fashion.

**Cloud native:** Continuous Delivery (CD)

Continuous delivery describes an organization that ensures new functionality is released to production at high frequency, often several times per day. That does not mean the new functionality is exposed to all users immediately. It might be temporarily hidden or reserved for a subset of experimental or preview users.

With CD we typically see:

* A so-called “deployment pipeline” where new code from developers is automatically moved through build and test phases.
* Automatic acceptance (or rejection) of new code for deployment.
* Thorough testing of functionality, integration, load, and performance happens automatically.
* Once a developer has put their code into the pipeline, they cannot manually change it.
* Individual engineers do not have permission to change the production (live) servers.

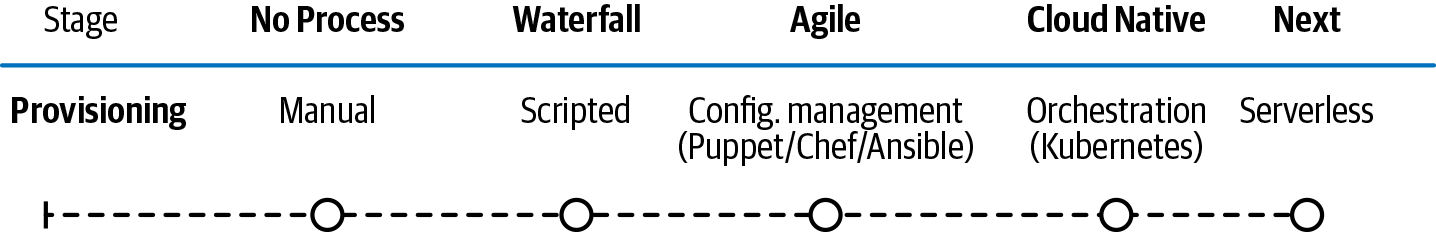
Cloud native organizations typically combine integration and deployment processes, known as “CI/CD,” to drive continuous improvements to their systems. They also run tests on their production systems using methods such as “chaos engineering” (a way of forcing outages to occur on production systems to ensure those systems recover automatically) or live testing for subsets of users (e.g. “A/B testing”).

**Next:** Continuous Deployment

The next evolution of delivery is continuous deployment, where we see fully automatic deployment to production with no approval process—just a continuous flow of changes to customers. The system will automatically roll back (uninstall) new changes if certain key metrics are negatively impacted, such as user conversion.

**Provisioning**

How do you create new infrastructure and new machines? How quickly can you deploy everything, and how automated is this process? Provisioning is the Maturity Matrix axis where we are happiest to see a company leading the other eight areas! [Figure 5-9](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch05.html#provisioning_axis_of_the_cloud) shows the range of delivery approaches we look for in a Maturity Matrix assessment.



**Figure 5-9. Provisioning axis of the Cloud Native Maturity Matrix**

**No process:** Manual

In a manual system, a developer (who is also your operations person) logs in to a server and starts apps manually or with rudimentary scripting. Servers are accessed using primitive file transfer mechanisms like FTP.

This is a common situation in startups. It is slow, labor-intensive, insecure, and doesn’t scale.

**Waterfall:** Scripted

Developers build an app and hand it over to the operations team to deploy it. The ops team will have a scripted mechanism for copying the application and all its dependencies onto a machine to run. They will also have a scripted mechanism for configuring that machine, or they may have pre-configured virtual machines (VMs).

In this case, because the development team “throws their app over the wall” to operations, there is a risk that the development team built and tested their app using different tools, versions, or environments to those available to or used by the ops team. This can cause an application that worked fine for the dev team to fail to work when operations puts it on their test or live servers. This introduces confusion when issues are subsequently seen: is there a bug in the app delivered by dev, or is it an issue in the production environment?

**Agile:** Configuration Management (Puppet/Chef/Ansible)

In a system with configuration management, applications are developed to run on specific hardware or virtual machines. Tools like Puppet, Chef, or Ansible allow operations engineers to create standardized scripts, which are run to ensure a production system is configured exactly as required for the application provided by development. This can be done at will (i.e., fast) but there is limited automation (mostly a human still needs to press a button to run the scripts).

Developers often deploy on their local test environments with different, simpler tooling. Therefore, mismatches can still occur between developer environments and production ones, which may cause issues with the live system. However, this is less common and faster to resolve than with more ad-hoc scripting.

**Cloud native:** Dynamic Scheduling/Orchestration (Kubernetes)

Applications in production are managed by a combination of containerization (a type of packaging that guarantees applications are delivered from development with all their local operational dependencies included) and a commercially available or open source orchestrator such as Kubernetes.

The risk of a mismatch between development and live environments is reduced or eliminated by delivering applications from Dev to Ops in containers along with all of the app’s dependencies. The Ops team then configures Kubernetes to support the new application by describing the final system they want to produce in production. This is called declarative configuration.

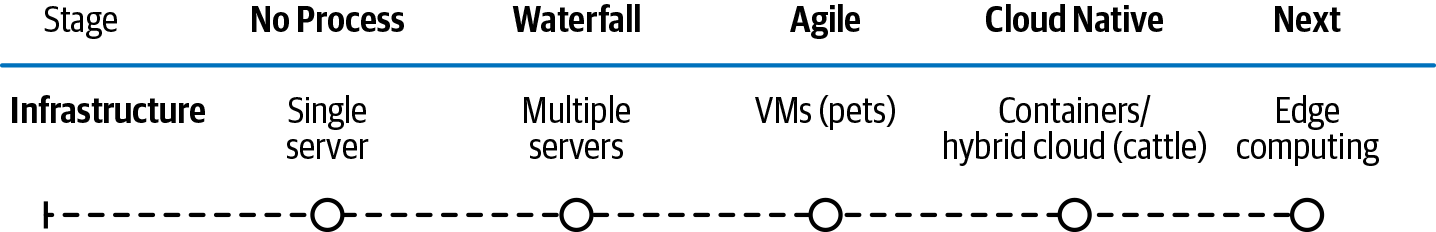
The resulting system is highly resilient, automated, and abstracted. Neither engineers nor the apps themselves need to be aware of hardware specifics. Everything is automatic. Detailed decision making about where and when applications will be deployed is made by the orchestrator itself, not a human.

**Next:** Serverless

All hardware maintenance and configuration is done in a fully automated way by your cloud provider’s platform. Code is packaged by developers, submitted to the serverless service(s), and can potentially be distributed and executed on many different platforms. The same function can run for testing or live. Inputs, outputs, and dependencies are tightly specified and standardized. Serverless is rapidly being adopted across the cloud native ecosystem and is well on its way to becoming standard cloud native best practice.

**Infrastructure**

Everyone knows this one: single server to multiple servers to VMs running in your own data center. Then shifting to Hybrid cloud for a computing environment that mixes on-premises infrastructure with private and/or public cloud services best tailored to your company’s specific needs and use case. [Figure 5-10](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch05.html#infrastructure_axis_of_the_cloud) shows the different Infrastructure options we look for in a Maturity Matrix assessment.



**Figure 5-10. Infrastructure axis of the Cloud Native Maturity Matrix**

**No process:** Single server

In a single server environment you run all of production on a single physical machine. This may be an old desktop sitting under a desk in the office. You have no failover servers (resilience), and you deploy to your server using copy-and-paste file transfers. You probably have some rudimentary documents to describe the setup.

**Waterfall:** Multiple servers

A multiple servers (physical) infrastructure will handle a moderately complex application. You can have a sophisticated system of multiple interacting applications, for example front ends and a clustered database. Redundancy ensures that if one machine fails, another will take over. This is probably all sitting in a simple co-located data center.

Your operations team may use manual problem solving, and it might take days or weeks to provision new infrastructure because it’s hard to get more rackspace! Compute, storage, networking, and security are usually managed separately and require separate requests to ops. New infrastructure is ordered through a ticketing system and provisioned by ops.

**Agile:** VMs (“pets”)

A VM-based environment is similar to a multiple-servers environment in that you have a set of machines and manual server setup. (VMs are sometimes referred to as “pets,” due to the small number of machines and the personal relationship that arises from needing to interact with each of them regularly.) However, this is made easier by using standardized virtual machine images. You use virtualization software such as VMware to help manage your virtual machine instances. You get better resource utilization (and therefore an effectively larger system for your money) by running multiple VM instances on each physical server.

Your operations team uses manual or semi-automated provisioning of new infrastructure resources. Your VMs are “mutable”—engineers can log on to them and change them by, for example, installing new software or fixes. Each machine is maintained separately, and it would be painful if one died (hence, “pets”). It will generally take hours or days to provision new infrastructure, mainly due to handovers between Dev and Ops teams.

**Cloud native:** Containers/hybrid cloud (cattle)

Here, individual machines don’t matter: they are called “cattle” because there is a big herd and they are interchangeable. There is usually full automation of environment creation and maintenance. If any piece of infrastructure fails, you don’t care—it can be easily and almost instantly recreated.

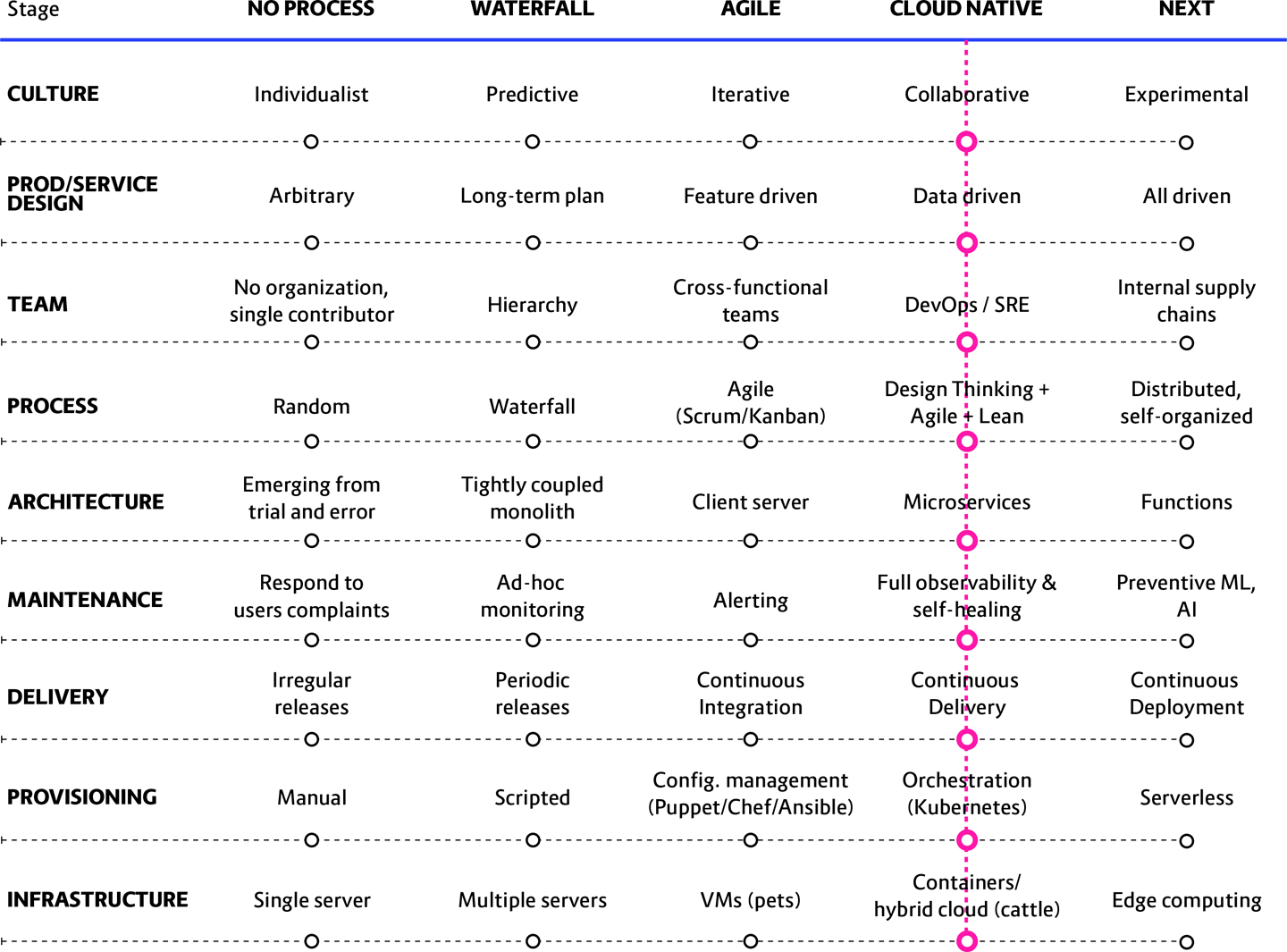
Unlike VMs, these are never directly provisioned; they are accessed only through automated processes exposed through APIs. This automation means new infrastructure takes minutes, even seconds, to provision. Containers are used for application packaging, which makes it easier to run those applications anywhere, including different “hybrid” cloud environments, whether public or on-premises.

**Next:** Edge computing

The next evolution for infrastructure is edge computing. Decentralized computer processing as the edge of your network. Edge computing takes applications, data, and computing power (services) out of a centralized location and distributes them to locations closer to the user. (Kind of like microservices for compute loads, really.) Edge computing returns results fast and works well in applications where, for example, adequate data is available locally.

**Connecting the Dots**

OK! Now you have read through the nine Maturity Matrix axes and identified, at least roughly, your company’s current practices, from Culture to Process to Infrastructure. Now it is time to copy your answers from each individual axis onto the full blank matrix in order to graph your own real-time status. [Figure 5-11](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch05.html#the_blank_cloud_native_maturity) provides a blank version of the matrix, or you can visit [*https://container-solutions.com*](http://bit.ly/37L69Ij) to find a full-size downloadable version.



**Figure 5-11. The blank Cloud Native Maturity Matrix template, ready for you to fill in and then connect your very own dots**

It’s very simple: match your answer from the section above, where we explained each individual axis, to draw a point on the corresponding place on the full blank matrix. Then literally connect the dots by drawing a line through each status point. Graphing status in this way gives instant valuable feedback and provides a powerful visual of your company’s current state.

At this point we are using the Maturity Matrix specifically in a “know thyself” capacity. You have mapped where you are right now and where you want to go—a crucial first step in preparing for a migration.

However, there is much more we can do with this useful data! See [Chapter 9](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#patterns_for_development_and_process), Antipatterns and Common Challenges, for problems that commonly arise during transformation initiatives and what the Maturity Matrix looks like in each scenario. (It’s kind of like a field guide for what not to do.) It is very useful to match your results against typical scenarios that we have observed at other enterprises seeking to migrate their legacy systems and transform into a true cloud native entity. If they are strikingly similar, well, then we have patterns for fixing exactly that problem!

**The Axes, United**

The cloud native approach empowers enterprises to design their product exclusively around the user, with no concern for the needs of the underlying system. This lets them deliver better products with less risk, which is the true heart of cloud native. That they also can now deliver them faster and cheaper is a pleasant corollary outcome.

By undergoing the Cloud Native Maturity Matrix assessment, a company generates an intelligent, flexible, and constantly updatable status check. Granted this understanding and perspective, companies can begin to plan their cloud native transition with knowledge—and confidence that they will be able to avoid common pitfalls along the way.

We will revisit the Maturity Matrix again in [Chapter 13](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch13.html#common_transformation_challenges), when we use it to identify common problems arising during the cloud native transformation process. These typical scenarios occur often enough that they are a valuable source of useful patterns (and even antipatterns) for constructing a proper migration path. It is wise to examine them first as a valuable lesson of what *not* to do in a cloud transformation and then as an aid in custom-mapping the right path.