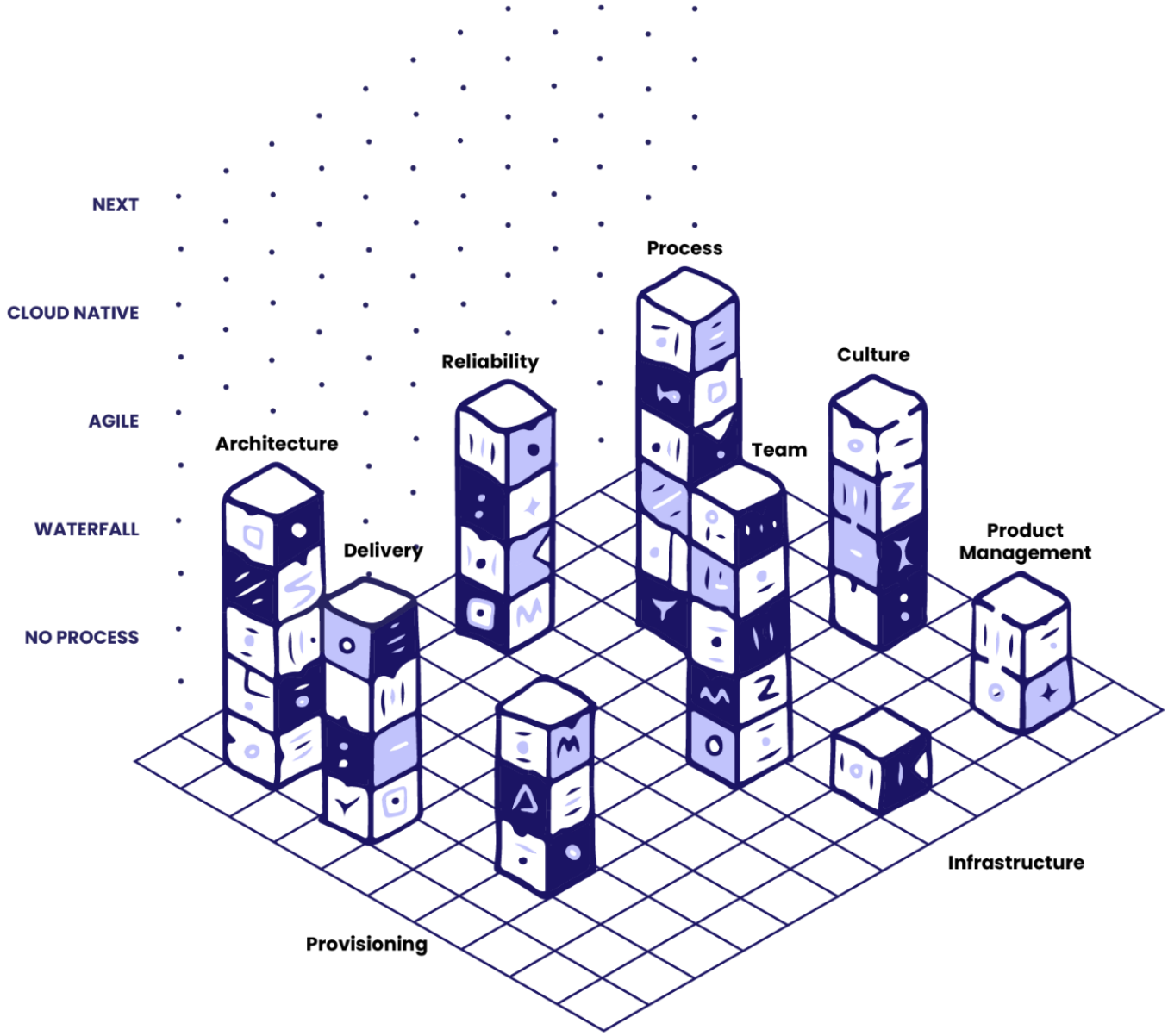
Container Solutions

What is the Cloud Native Maturity Matrix?



Introduction

Even if you are getting an inkling of WTF Cloud Native is, you might still be unsure about starting a journey toward it. Because Cloud Native is still new and mysterious. And if you're starting on a trip of any kind, it helps quell any anxieties you may feel if you know the answers to two questions:

- How far is our destination?
- Are we there yet?

At the core, Cloud Native is a method for optimising software systems for the cloud. It is an approach to systems architecture that harnesses the cloud's most powerful advantages—flexible, on-demand infrastructure, and managed operational services—using Continuous Delivery.[1]

But it's a lot more than just some cutting-edge technologies that can make things go faster. To work properly—and not be an expensive, time-wasting boondoggle—Cloud Native tech needs to be accompanied by other changes. Changes in your organisation's culture, for instance. And in how you align IT and business goals strategically.

Most enterprises consider a Cloud Native transformation because they want to drastically speed up their ability to build, test, and deploy software, reducing that time from months to days or even hours. Beyond sheer velocity, the ability to continually deploy and update applications without ever disrupting users is the

[1] Continuous Delivery automates the delivery of small, iterative changes to run on cloud-based infrastructure. It provides an automated way to push code changes to teams working outside the production pipeline, performs any necessary service calls to web servers or databases, and executes procedures when applications are deployed.

ultimate goal of a cloud transformation. Companies that can't do this will quite simply get left behind.

If their software is developed on a Waterfall[2] or even Agile[3] model, those companies will find that customer expectations have shifted by the time their application finally deploys—moving them even further behind the competition. With Cloud Native delivery processes, however, those companies can more easily keep pace with rapidly shifting technology and customer demand.

Going Cloud Native is a complex process that few companies have deep experience in navigating. The tech is new. There's a thin supply of engineers and developers who are fluent in it. And the path to transformation is different for nearly every organisation that sets out on the journey.

It makes sense that, in a young and quickly evolving sector, there were no maps to steer by.

So we made one.

Container Solutions have been guiding companies onto the cloud for seven years now, carefully observing and analysing each experience. We took the lessons we learned to develop the Cloud Native Maturity Matrix. It's an assessment tool for helping map the path between where an organisation currently finds itself—and where it wants to be.

---

[2] A commonly used model of software development based on a logical progression of steps that form the software development life cycle. One follows after the other in strict order, much as a waterfall cascades down from top to bottom. This method is often associated with lengthy release cycles.

[3] Software development principles focusing on iterative delivery, frequent user feedback, and collaboration over complex processes. Scrum is the most well-known development method integrating the Agile principles. For example, sprints in Scrum implement the Agile principle 'deliver working software frequently'.

Our Cloud Native Assessment uses interviews, workshops, and an assessment of the tech stack within an organisation to gather information which then is used to create a snapshot of that company along the Cloud Native Maturity Matrix's ten different axes. Some are technically oriented: infrastructure, maintenance, delivery, and security. Others assess people-oriented aspects: management process, team structure, and internal culture.

We use the gathered information to define, analyse, and describe an organisation's current status in each of the ten categories. That status is literally plotted out on the matrix, and, the gap between the company's current state and a Cloud Native state is easy to see. This data—constantly re-assessed as work progresses—allows us to make intelligent choices and monitor progress.

In other words, the Cloud Native Maturity Matrix lets us create a custom map for each company's unique path to the cloud.

What's new in this edition?

With this, our third iteration of the matrix, we've added a lane for security, as we're seeing organisations more tightly integrate their security operations into the rest of their operations, driven by "DevSecOps" approaches.

We've also seen wider adoption of FaaS as an alternative to microservices and Kubernetes since the last version was released, but we are still keeping this in our "next" categorisation as containers/Kubernetes and microservices remain by far and away the dominant approach.

We debated adding a lane for "sustainability" but whilst we are seeing signs of more organisations thinking about this, and we hope it will become mainstream with time, it remains the case that we have not yet come across an enterprise that went

Cloud Native because they wanted to be more green, and so adding a dedicated lane felt permature.

Likewise we considered adding a lane around “diversity and inclusion” but, again, it isn’t a driver towards being more Cloud Native. However there is no doubt that diversity matters, and there is a growing body of evidence that organisations with more diversity in terms of gender and under-represented minorities achieve higher team performance and better business outcomes. We strongly recommend that teams wanting to achieve high performance do their best to recruit and retain more women and under-represented minorities, including increasing neurodiversity, and foster an environment that is truly inclusive.

With this in mind, here is the latest version of matrix itself:

Maturity	NO PROCESS	WATERFALL	AGILE	CLOUD NATIVE	NEXT
Culture	Individualist	Predictive	Iterative	Experimental	Self-Organised
Product Management	Arbitrary	Long-Term Plan	Feature Driven (feature factory)	Data Driven	AI Driven
Delivery	Irregular Releases	Periodic Releases	Continuous Integration	Continuous Delivery	Continuous Deployment
Process	Informal	Top-Down	Agile (Scrum/Kanban)	Design Thinking + Agile + Lean	Distributed
Team	No Organisation, Single-Contributor	Siloed	DevOps	SRE	Internal Supply Chains
Architecture	No Guiding Principle	Monolith	Distributed Monolith	Microservices	Functions
Reliability	Inconsistent SDLC	Established Simple SDLC	Established Software Quality Model	Managed Reliability Fault Tolerant Platform	Automated Rollback & Automated Failover
Provisioning	Manual	Scripted	Config Management	Orchestration	Serverless
Infrastructure	Single Server	Multiple Servers	VMs	Immutable Infrastructure	Optimal Infrastructure
Security	Reactive / Ad-hoc	Review on Release	Regular Review Within SDLC	Continuous Security	Continuous Protection and Remediation

Cloud Native

Diagram 1: A blank Cloud Native Maturity Matrix, showing all ten categories.

What is the Cloud Native Maturity Matrix?

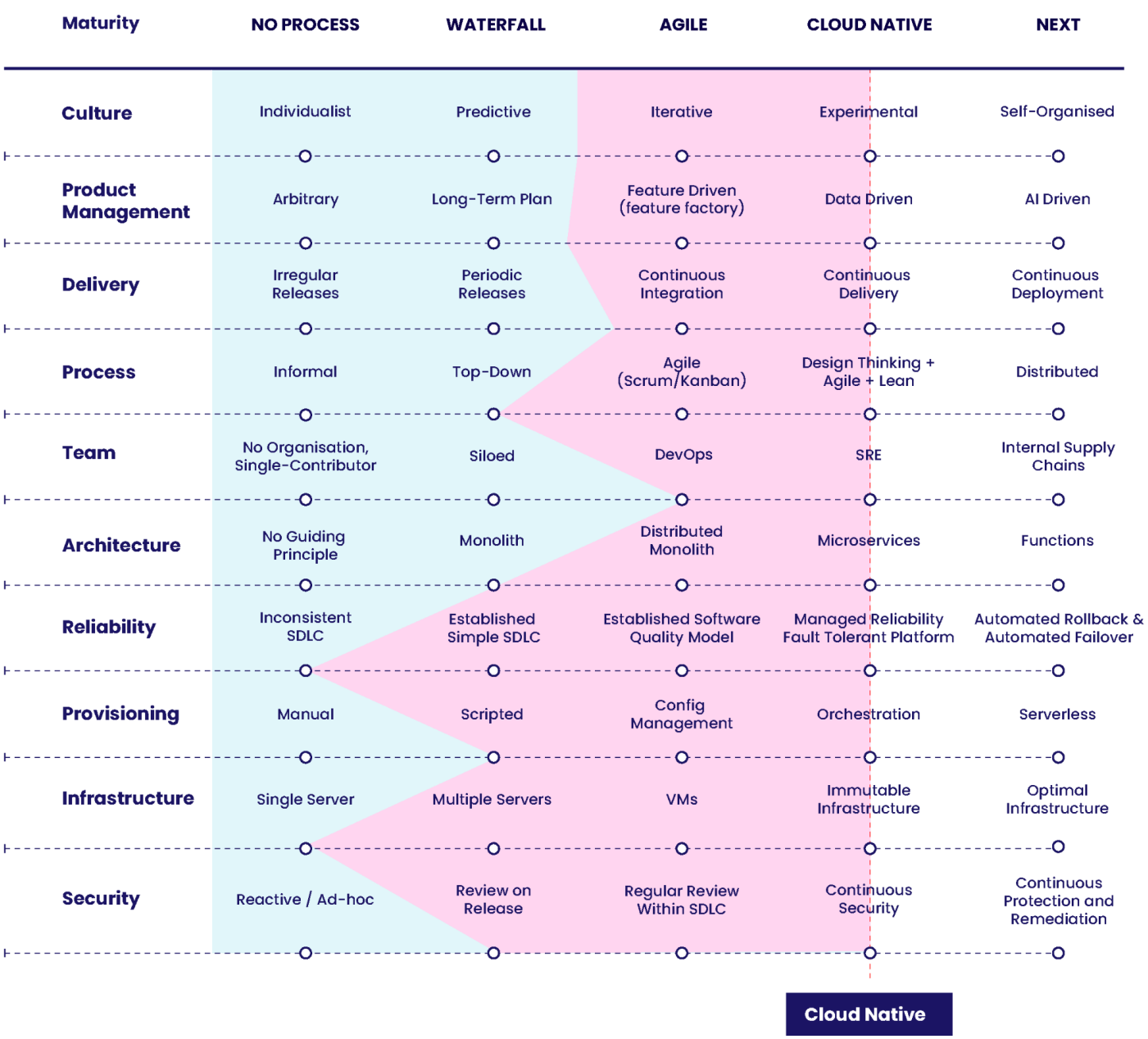


Diagram 2: An example of a completed Cloud Native Maturity Matrix; the red area shows the gap between the organisation's current state in each category and Cloud Native status.

This reference guide will help explain each category of the Cloud Native Maturity Matrix, and what Cloud Native status means in each area. We'll cover:

- Culture
- Product Management
- Delivery
- Process
- Team
- Architecture
- Reliability
- Provisioning
- Infrastructure
- Security

Culture

Your Culture describes the way individuals in your organisation interact with one another.

No Process: Individualistic

In an individualistic organisation there is no approved way to interact with peers, supervisors, or subordinates. Instead, communications are rooted in personal preferences. The communications processes often change when the people in a team change. This is a common situation for startups, but becomes chaotic and unsustainable as you scale up.

Waterfall: Predictive

A predictive organisation embraces long-term planning and commits to deadlines. The goal of a predictive business is to deliver what was agreed, on time. Often the delivery will be large and complex. Delivering as fast as possible or exploring novel new ideas are not priorities; in fact, exploring new ideas is often actively discouraged.

In such an organisation you would expect to see large amounts of documentation; procedures for changes, improvements, and daily tasks; segregation of teams by specialisation; tools for every situation; and regular, lengthy planning meetings.

Delivering an agreed specification on time is a difficult endeavour. Predictive companies need bureaucratic processes (for example, change control) and specialised team responsibilities (specific functions and technologies). This means formal handovers between teams—for example, from development to test to operations. It also requires modest cooperation within teams and between teams coordinated by full-time project managers.

Predictive organisations tend to suppress novelty because it is essentially unpredictable. They value rule following, encourage permission-seeking, and punish deviation. All of these behaviours are logical, given the desire to deliver complex systems exactly as specified.

This culture is common in medium to large enterprises.

Agile: Iterative

An Agile organisation has similarities to a predictive one, but it chooses smaller and simpler goals, which it aims to deliver as fast as possible. Agile organisations tend to focus on the short term rather than following a long-term plan. Communication is often by short, daily meetings.

Culturally, Agile organisations prefer fast responses and quick fixes, which may 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 individual teams responsible for their part of the backlog. Inter-team communication is by Scrum masters and other coordinators. They typically have high cooperation *within* teams but modest cooperation *between* teams. Risks are usually shared within teams, but not between them. Like in a predictive organisation, Agile organisations normally have narrow responsibilities within a team, and narrow responsibilities for a team.

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

Cloud Native: Collaborative

A collaborative organisation tends to have big goals, but less specific ones than a predictive organisation. For instance, there may be a broad vision but without a detailed specification or a fixed delivery date. This culture embraces learning and consistent, continuous improvement over predictability.

Typically, this culture involves teams with full responsibility for their services, tools, and processes during the entire lifecycle—from design to deployment. High levels of collaboration exist within teams and between teams. There is often constant communication, using team chat tools like Slack. This culture rewards self-education, experimentation, and research. Results are coldly assessed based on field data.

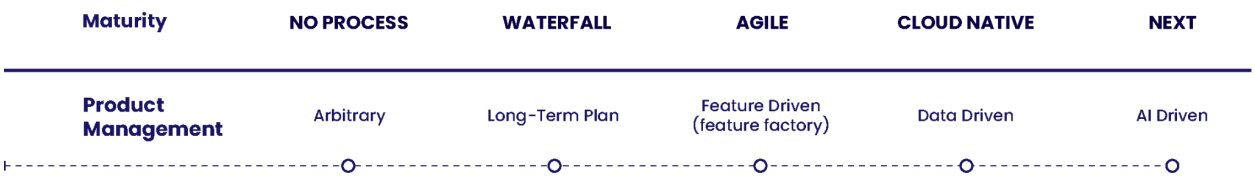
A collaborative culture is increasingly being adopted in companies operating in areas of high uncertainty or fast change.

Next: Experimental

We predict the next type of organisation will be an experimental one. In an experimental culture, people within an organisation are encouraged to try new ideas on a small scale, learn from their failures, and scale up their successes.

Product Management

The Design category describes how decisions are made within your organisation about what product work to do next. What decides which products to develop, or which improvements or new features are tackled next?



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

A long-term plan driven 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 12 months. This process is a common one 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 organisation focuses on fast change, often without an overarching long-term plan.

Cloud Native: Data-driven

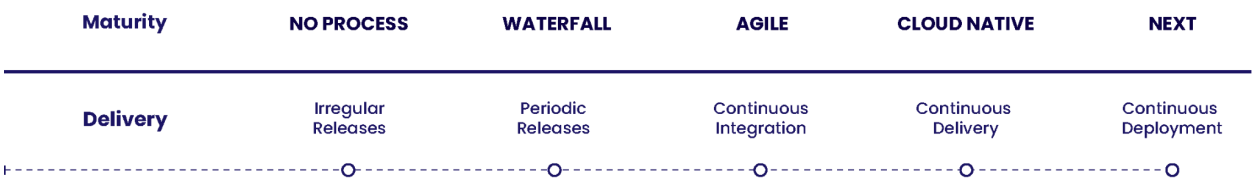
In a data-driven design process, the final say on which features stay in a product is based on data collected from real users. Potential new features are chosen based on client requests or designs by product owners without a long selection process. They are rapidly prototyped and then potentially developed 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: Artificial Intelligence-driven

In the future, significant interaction with AI-driven systems will help make product decisions both big and small based on data fed to the AI and questions asked of it with relatively little developer interaction.

Delivery

The Delivery process describes how and when software from your development teams gets to run in your live (production) environment.



No Process: Irregular Releases

In many small organisations, irregular software releases (new functions 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 releases

Many organisations have periodic scheduled releases—for example, every six months. The contents of these, usually infrequent, releases becomes extremely important and are the result of long planning sessions. Extensive architectural documents for each release are produced by enterprise architects and requirement documents by business analysts. No coding is done before the full architecture is ready. Once the release contents are agreed, any change is subject to a Change

Approval Board. 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 complete system.

Agile: Continuous Integration

Continuous Integration describes an organisation 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 organisation using Continuous Integration typically:

- Has a single codebase (aka a 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.

- Has a fully automated build process that turns new code into runnable applications.
- As part of the build, includes automated testing of all code. That forces developers to fix bugs as they go along, which is easier than fixing them late in the process.
- Requires 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.
- Has a way to deploy code to test or production hardware in an automated fashion.

Cloud Native: Continuous Delivery

Continuous Delivery describes an organisation 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.

A tech organisation using Continuous Delivery typically:

- Has a so-called 'deployment pipeline' where new code from developers is automatically moved through build and test phases.
- New code is accepted (or rejected) for deployment automatically.
- 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.

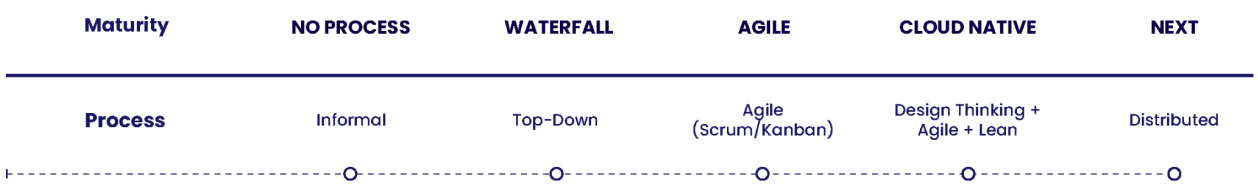
Companies using Continuous Delivery usually display continuous systems improvements. They also run tests on their production systems using methods such as chaos engineering (a way of forcing outages to occur on production systems to ensure those systems recover automatically) or live testing for subsets of users (A/B testing).

Next: Continuous Deployment

The next evolution of delivery is Continuous Deployment. In an organisation using this process, 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—such as, say, user conversion—take a hit.

Process

Process describes how your organisation executes its work.



No Process: Informal

In an informal organisation, 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: Top-Down

In a Top-down organisation, the product-development process is tightly controlled through up-front planning and change-management processes. A sequential process is followed by 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 and any and every handover is well documented and requires forms and procedures.

Agile: Agile (Scrum/Kanban)

In an Agile organisation, 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. Only limited changes, if any, are allowed during sprints to protect the delivery deadlines.

Cloud Native: Design Thinking + Agile + Lean

In a Design Thinking organisation, Design Thinking and other research and experimentation techniques are used for de-risking large and complex projects. Many proofs of concept (PoCs), or small experiments, are developed to compare options. Kanban is often then used to clarify the project further; finally, Scrum is used once the project is well understood by the entire team.

This relatively new process can be used in situations of high uncertainty or where the technology is changing rapidly.

Next: Distributed, self-organised

In the future, self-organised systems will be highly experimental, with less up- front design. Individuals or small teams will generate ideas that are iterated and improved on in the field automatically by the platform.

Team

The Team axis describes how responsibilities, communication and collaboration works across teams in your organisation.

No Process: No organisation, single contributor

Here 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

A hierarchy organisation is organised via ranked positions within and between the teams. Decisions are made by managers and implementation is done by specialised teams (making it difficult to move individuals between teams). There will be separate teams of architects, designers, developers, testers, and operations engineers. Inter-team communication is generally through tools like Jira or via managers. Historically, this has been the most common structure in large organisations.

Agile: Cross-functional teams

In a cross-functional organisation there is less specialisation by 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 hierarchy remains outside (rather than within) teams.

Cloud Native: DevOps/SRE

A DevOps team is a development team capable of designing and building applications as part of a distributed system, and also operating the production platform/tools. Each team has full responsibility for delivering microservices[4] and supporting them. DevOps teams include planning, architecture, testing, dev, and operational capabilities.

However, often there's some 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 Engineers (SRE) or 1st level support teams respond to alerts. However, there is considerable collaboration between those teams and individuals can easily move between them.

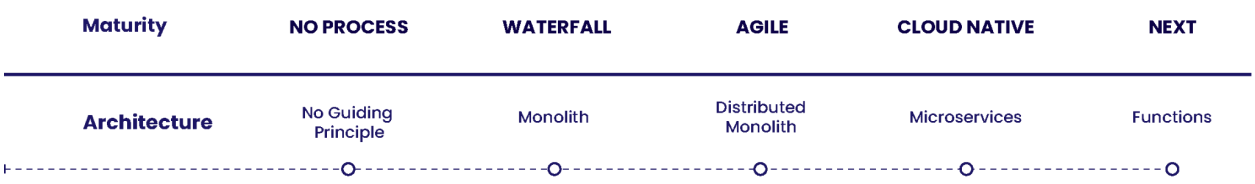
Next: Internal supply chains

In an internal supply chain organisation each service is a separate product, with full tech and business generation responsibilities in the teams—much as many e-commerce teams have been managed for a decade.

[4] An approach to application development in which a large application is built as a suite of modular components or services. Each service runs a unique process and usually manages its own database. A service can generate alerts, log data, support UIs and authentication, and perform various other tasks. Because microservices enable each component to be isolated, rebuilt, redeployed, and managed independently, development teams can take a more decentralised (nonhierarchical) approach to building software.

Architecture

Architecture describes the overall structure of your technology system.



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, 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. Typically, the monolith is written in a single programming language with strong standardisation on tooling. The

application is usually vertically scalable (you can support more users by adding more resources on a 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. Unfortunately, there are few developers or architects who can hold the entire system in their heads. Most people don't understand the full complexity of the app and that the impact of a single bug may be unpredictable and create domino effects that can destabilise the system overall.

Agile: Client server

A client server architecture is the most basic form of distributed system. It is designed to handle a system of multiple components communicating through networks that might be slow or unreliable. Each component is often similar to a monolith with a layered architecture. Each service can be clustered (which enables targeted horizontal scaling and resilience).

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, the system is usually possible to develop while working in parallel to some degree (one group handles the client part, one the server).

Cloud Native: Microservices

A 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).

Operationally, it is common to manage microservice deployment in a fully automated way. 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.

Next: Functions

A functions (aka serverless) architecture is one where no infrastructure needs to be provisioned. Each piece of business logic is in a separate function, which is operated by a fully managed Function-as-a-Service, such as AWS's Lambda, Microsoft's 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.

In comparison to Microservices and Kubernetes, functions potentially allow you move even faster, and we're seeing a number of cases now where organisations have either implemented entirely on Serverless from the start (for example the BBC, Honeycomb for the retriever database layer) or have moved to this approach from Kubernetes (for example cinch). At the same time, whilst there are still use cases where functions don't work as well, these are being chipped away by the various cloud providers over time. Because of this, we recommend taking a "serverless first" approach, where you start with FaaS, and move to containers and Kubernetes if you find that you have use cases functions don't work well for.

Reliability

Reliability describes how quality is managed in software, with respect to its consistency and safety, through to its deployment and operation.

No Process: Inconsistent Software Development Lifecycle (SDLC)

In an environment without a consistent SDLC, there are no established practices for managing the building and deployment of software. Software is built and released in an inconsistent and undocumented way. Updates are made without a process being followed for the consistent future management of the software.

Waterfall: Established Simple SDLC

An established simple SDLC has well-understood, consistent, and documented practices across the various stages of software development and deployment. At the build stage, source control and common and stable development environments exist for engineers to use. The packaging stage is scripted and maintained separately from the software being packaged. The deployment stage is also controlled and managed separately from the development processes.

In production, any changes to the deployed software are managed in a consistent and well-understood way, with updates to configuration or code being reflected in source control under some kind of governance regime.

A waterfall approach to reliability across the SDLC will also separate the requirements gathering, design, and build stages in order to improve the quality, completeness, and effectiveness of the work undertaken.

In this 'established simple' stage, some kind of testing process is standard practice. Often, this process will be manual, and performed by a discrete and siloed QA team.

Agile: Established Software Quality Model

An Agile approach to reliability introduces DevOps principles to the SDLC. Quality is improved through the use of 'shift left' principles which seek to automate quality controls earlier in the SDLC. Typically this will involve some kind of basic linting, as well as some level of automated testing.

These automatic processes will be run continuously, i.e. on each merge to a main branch of source code, or on a regular cadence, in contrast with waterfall which typically runs these processes at a later stage in the SDLC.

Cloud Native: Managed Reliability Fault Tolerant Platform

Building on the DevOps principles of automation and 'shift left' detailed above, managed reliability uses SRE principles and management methodologies to improve software reliability throughout the SDLC.

In this phase, deployment and testing is substantially automated, likely using GitOps methods, with pipelines producing immutable artefacts ready for deployment. Production environments will be zero-touch, using agents to deploy updates.

Management of production environments will be along SRE principles, with Service Level Indicators (SLIs), Service Level Objectives (SLOs), and error budgets determining the focus of the teams' reliability efforts.

Cloud Native architectures are conceived and executed with reliability in mind. To achieve this, distributed low-dependency software architectures are used that help to tolerate failure in both infrastructure and software systems.

Next: Automated Rollback and Automated Failover

The state-of-the-art Cloud Native best practice in reliability involves a high level of automation to manage out-of-band processes that are more normally dealt with with some level of manual intervention.

Automated rollback is a self-healing process that uses native health metrics to determine whether a release has failed. It will be part of the CI/CD pipeline used to build and deploy the software.

In addition, large-scale infrastructure and software failures are managed automatically with failover occurring in as seamless a way as possible. Failure tolerance will work on a number levels, from an individual VM up to cloud regions, and even whole cloud providers. Resilience tests of these failovers will be frequent and, in advanced cases, scheduled switchovers will be part of normal operation.

Provisioning

The Provisioning process describes how you create or update your systems in your live production environment.

No Process: Manual

In a manual system, a developer (who is also your operations engineer) 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, labour-intensive, insecure, and doesn't scale.

Waterfall: Scripted

In a scripted system, 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 developers built and tested their app using

different tools, versions, or environments to those 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 its 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. Commercially available or open source configuration tools like Puppet, Chef, or Ansible allow operations engineers to create standardised 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 (in other words, fast) but there is limited automation (mostly a human presses 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: Orchestration (Kubernetes)

In a system with Orchestration, applications in production are managed by a combination of containerisation (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 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.

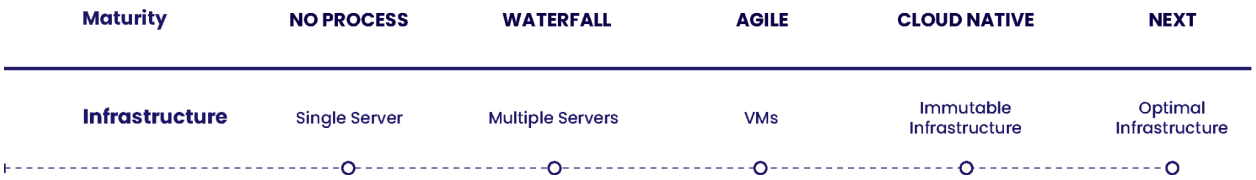
Next: Serverless

It is now becoming more common for companies to be serverless—to give up Ops or even DevOps and allow all hardware maintenance and configuration to be done in a fully automated way by what is usually a cloud platform.

Code is packaged by developers, submitted to the serverless service, 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 standardised.

Infrastructure

Infrastructure describes the physical servers or instances that your production environment consists of: what they are, where they are, and how they are managed.



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 may have some redundancy (if one machine fails, another will take over) and you can have a sophisticated system of multiple interacting applications—for example, front ends and a clustered database. This is probably all sitting in a simple, co-located data centre.

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: Virtual machines (pets)

A virtual machines (pets) based environment is similar to a multiple servers environment in that you have a set of machines and manual server setup. However, this is made easier by using standardised virtual machine images. You use virtualisation software, such as VMWare, to help manage your virtual machine instances. You get better resource utilisation (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' —meaning, 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)

In a containers/hybrid cloud (cattle) environment, individual machines don't matter. Ops or DevOps don't directly provision infrastructure resources like VMs, they are only accessed through automated processes exposed through APIs.

It takes minutes or seconds to provision new infrastructure, always through APIs. Containers are often used for application packaging, which makes it easier to run those applications on multiple different 'hybrid' cloud environments (on prem or public). There is usually full automation of environment creation and maintenance. Under normal conditions, your engineers have no manual access to physical

infrastructure. If any piece of infrastructure fails you don't care—it can easily be recreated (hence, 'cattle').

Next: Edge computing

The next evolution for infrastructure is edge computing. Compute loads are run on edge devices—i.e. outside of your normal data centre. Edge computing returns results fast and works well where, for example, enough data is available locally and network connections to central data centre locations may be unreliable.

Security

Security describes the processes by which you manage external and internal threats to the integrity of your software infrastructure and intellectual property.

No Process: Reactive/ad hoc

In a reactive environment you respond to security-related events when you are forced to. You may discover a breach has taken place, or been alerted to one by a user, and are forced to take action. There is no formal process for managing these events, rather a manager allocates the most eligible person available to the matter as it arises. This approach is typical of early-stage startups, especially in unregulated industries.

Waterfall: Review on Release

In a waterfall environment typical of medium to large enterprises, a process is in place to manage security matters at a point in the software development lifecycle which is easily gated: release. Typically, a separate 'security team' will be brought in as the software is being prepared to release to 'sign off' the implementation.

This security team will usually refer to documented standards and processes which define the scope and nature of the review. Reviews are carried out using a combination of manual inspection, scripted penetration tests, interviews with

developers and architects, and any other processes deemed necessary by the security team for the review. The outcome of reviews is either a direction to proceed (or not proceed) to production, with notes returned to those submitting the release outlining areas of improvement.

Such a 'Review on Release' approach can result in frustration, as both waiting for the separate team to respond, and the notes they return leads to delays as rework needs to be done to mitigate risks that could have been identified earlier in the software development lifecycle.

Agile: Regular Review Within SDLC

In an Agile development environment the 'point of release' review is replaced with a more dynamic approach which aims to the redute the wasted effort that can result from the single pre-release review outlined in the Waterfall section above.

With this approach, developers and architects can get early advice from the security team on their approaches to a particular piece of work and adjust their development plans accordingly. This early access to expertise can obviate major design flaws early, and even change the approach development teams take to thThere may still be a need to review the final submitted solution before release, but this final review should be far less onerous as the context of the development will be already known to the security team and their confidence going into it should be higher as a result.

Cloud Native: Continuous Security

Continuous Security is analogous to Continuous Integration in that it describes an organisation that is ready to release new code at will, without needing to conduct a formal security review. For companies where releases are regularly held up by a

security review process, Continuous Security helps enable more frequent and lower-cost deliveries.

A tech organisation using Continuous Security typically:

- Uses automated tooling to regularly check that software assets and environments are in compliance with the standards defined by the organisation
- Where possible, stores the configuration of this tooling as code under source control. This helps auditing processes, both to be able see the configuration of systems at given points in time in the past, as well as interrogate the configurations at past points, and who is responsible for them
- Will release code or make changes to infrastructure in production when assets and environments are determined to be in compliance with company standards.
- Will consider the passing of these compliance checks to be sufficient to have a degree of confidence about changes.
- Has a skilled security expert assigned to each organisational unit capable of speedily resolving security issues that arise.

Next: Continuous Protection and Remediation

Continuous Protection looks over your running systems and provides you with real-time analysis of security vulnerabilities. Typically these techniques are event-driven off centrally-collected system logs or other events like changes to infrastructure configuration. This is also known as Security Information and Event Management (SIEM).

Tools used for this might use a database of existing vulnerabilities, and run a scan of your particular environment's configuration in order to determine which

vulnerabilities you may be exposed to, or it might look for unusual activity based on prior behaviour when the system was considered in an uncompromised state.

In more advanced products and configurations, remedial action can be to a greater or lesser extent automated, reducing the need for human assistance. For example, serverless functions might be triggered that close down misconfigured firewalls on creation based on an organisation's security policy. This is also known as Security Orchestration and Response (SOAR).

About Container Solutions

Container Solutions is a professional services firm that specialises in Cloud Native computing.

Our company prides itself on helping enterprises migrate to Cloud Native in a way that is sustainable, integrated with business needs, and ready to scale. We help companies increase independence, take control, and reduce risk throughout a Cloud Native transformation.

Container Solutions is one of only a handful of companies in the world that are both part of the Kubernetes Training Partner (KTP) programme and a Kubernetes Certified Service Provider (KCSP). Membership to both programmes is based on real-life, customer experience. When companies like Google, Atos, Shell, and Adidas need help with Cloud Native, they turn to Container Solutions. We are a remote-first company that operates globally, with offices in the Netherlands, the United Kingdom, Germany, and Canada.

Want to Read More?

Check out these Container Solutions publications:

The Cloud Native Attitude, 2nd Edition

By Anne Currie and Charles Humble

What We Do @CS: The Yin and Yang of Technical and Organisational Change

By Ian Miell

WTF is Cloud Native Data Security?

By Anastasiia Voitova

How to Recruit and Onboard Neurodivergent People

By Jennifer Riggins