4 Essential Elements of a **Kubernetes Platform**

AMBASSADOR | 

Table of contents

Why Cloud Native?

The emergence of “cloud native” technologies and practices, such as microservices, cloud computing, and DevOps, has enabled innovative organisations to respond and adapt to market changes more rapidly than their competitors. Just look at the success of the initial web “unicorns”, Spotify, Netflix, and Google. Obviously not every company can be a unicorn, but there is much to learn from the early adopters of the cloud.

The Benefits of Being Cloud Native

Spotify’s now famous “squad, chapters, and guilds” organisational model ultimately led to the creation of their applications as independent microservices, which in turn supported the rapid rate of change they desired. Through a combination of a compelling vision and the whole-scale adoption of cloud services, Netflix was able to out-innovate existing market incumbents in the video streaming space. And Google’s approach to collaboration, automation, and solving ops problems using techniques inspired from software development enabled them to scale to a global phenomenon over the past two decades.

Obviously strong senior leadership and a willingness to continually change and adapt an organisation’s internal culture has had a large impact on the outcomes. One of the most important focuses has been continually working to sustainably minimise the lead time to delivering value. This can be seen by the drive to minimise the friction from having ideas, to coding, to releasing functionality, and to obtaining feedback.

> **Organisations that have successfully embraced what we now refer to as a “cloud native” approach have invested heavily in two core areas: creating a self-service application platform, and adopting new tools and developer workflows.**

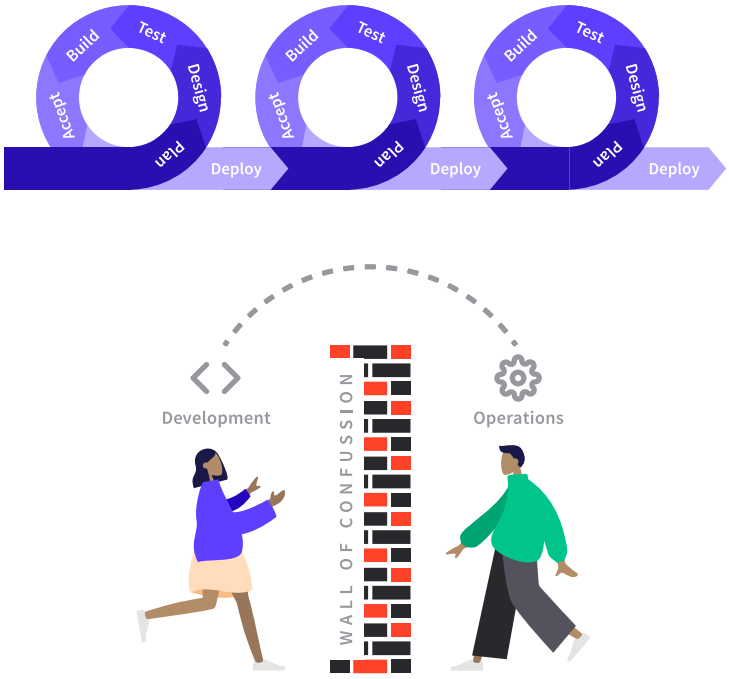
From an organisational perspective, these investments have broken down existing barriers between the operations and development teams that were traditionally mediated via ticketing systems. This has led to the creation of two high-level persona groups that collaborate via the use of well-defined APIs, automation, and focused in-person interaction:

- Platform teams and site reliability engineers (SRE) own the platform, continually evolve the platform functionality, and help curate operational best practices; and
- “Full cycle” development teams that own the organisation’s products and services, and leverage the platform and the new workflows to deliver value to customers.

Although beneficial, introducing these technical and organisational changes has not always been pain free. For better or worse, the traditional software development life cycle (SDLC) has been disrupted by the arrival of the cloud.

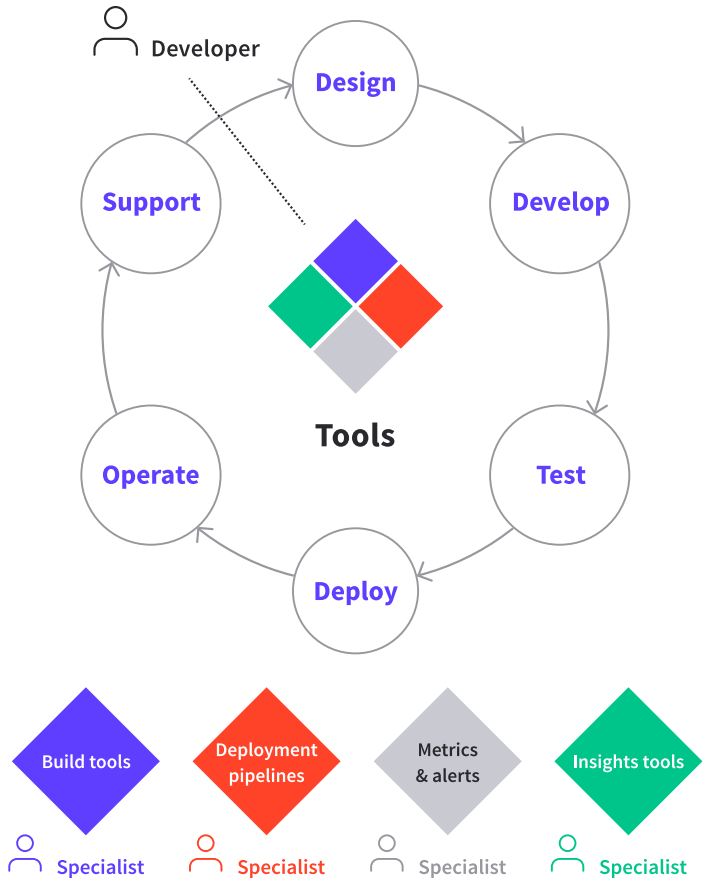
Full Cycle Development: Disrupting the SDLC

Within the traditional approach to the SDLC, engineers were specialised and often worked in silos. Operators built and managed data centers. Architects designed systems, drew boxes and arrows, and provided architectural governance. Developers typically coded and tested a large batch of changes against locally running instances of their monolithic applications. And quality assurance (QA) engineers verified and promoted the systems using a series of gated staging environments. Applications that passed QA were handed-off to operations to deploy and run. After this, any issues or anomalous behavior was identified by the ops team and handed back to developers.



Agile enabled rapid innovation, but didn't fully break down the dev/ops barrier

Embracing cloud technologies, such as Kubernetes, has allowed the operations team to automate platform provisioning and developers to self-service in regards to application deployments. The use of microservices has allowed product-focused development teams to work independently. Accordingly, the cloud native SDLC is very different. Developers are performing just-enough upfront architecture design. Developers are coding small iterative changes against multiple services, some of which may be running locally, and others remotely. Developers are now seeking to automatically execute QA-style verification as part of the coding process. And developers also want to release rapid, controlled experiments into production. This approach is known as full cycle development, and has been popularised by Netflix.



Full cycle development. “You build it, you run it” supported by platform tooling

It is worth taking a pause here to understand two core premises of this move towards “full cycle” development teams. This does not remove the need for specialist operations, sysadmin, or platform teams. This does, however, require upskilling within both development and operations teams.

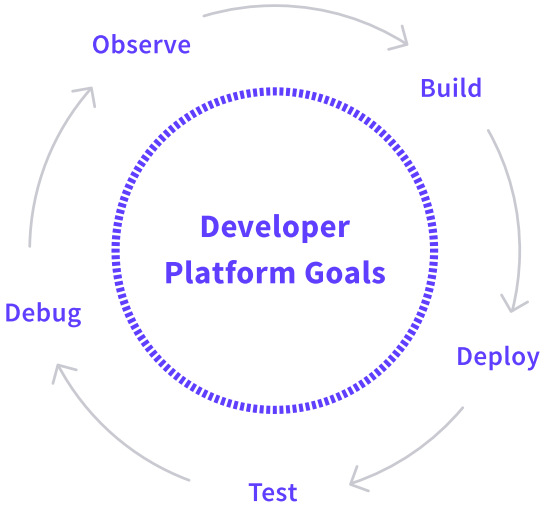
> **Full cycle development teams will have to cultivate an increased business domain expertise, and also extend their understanding of fundamental runtime configuration for their applications. Operations team will have to learn new cloud technologies and understand how these integrate with existing solutions into an effective platform.**

Enabling Full Cycle Development

Cloud computing and container orchestration frameworks provide an excellent foundation for deploying and running modern applications. However, in order for these technologies to support the move to full cycle development, there are several requirements that must be met for both the development and platform/SRE personas.

Full Cycle Developers: More Feedback, Faster

When adopting a cloud native approach, developers need to be able to run through the entire SDLC independently. And they need to do this with speed and with confidence that they are adding value to end-users (and not causing any unintended negative impacts). Developers want to build and package applications within containers, and rely only on self-service interfaces and automation (provided by the platform team) to test, deploy, release, run, and observe applications in production.

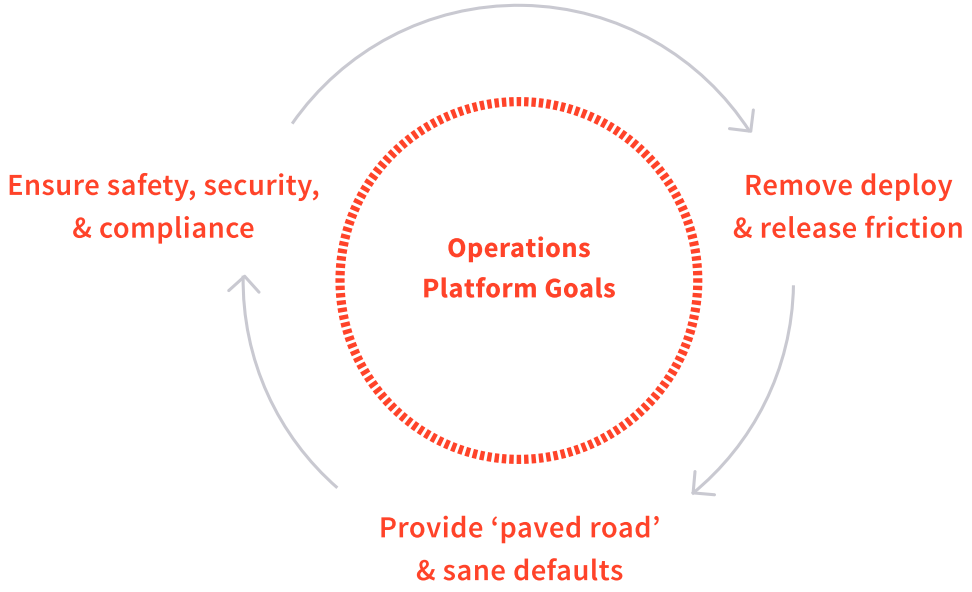


This rapid feedback loop supports developers in becoming more customer-focused. This enables more functionality to be delivered, with the goal of providing more verifiable value to end users in a repeatable manner.

Platform Teams: Remove Friction, Add Safety

Platform teams need to capable of providing three primary functions:

- Removing friction from packaging applications into containers, and from deploying and releasing functionality.
- Enabling the observation of system behaviour, and setting sane defaults and alerts that prevent unexpected functionality within the platform from causing cascading failure.
- Assisting the organisation to understand and meet security and regulatory requirements.



The foundational goals of continuous delivery are focused on being able to deliver functionality to end users in a manner that is as fast and as stable as the business requires. Creating a supportive platform is vital to this, and so is creating an effective developer experience (DevEx) for interacting with the platform via tools and APIs

Four Core Cloud Platform Capabilities

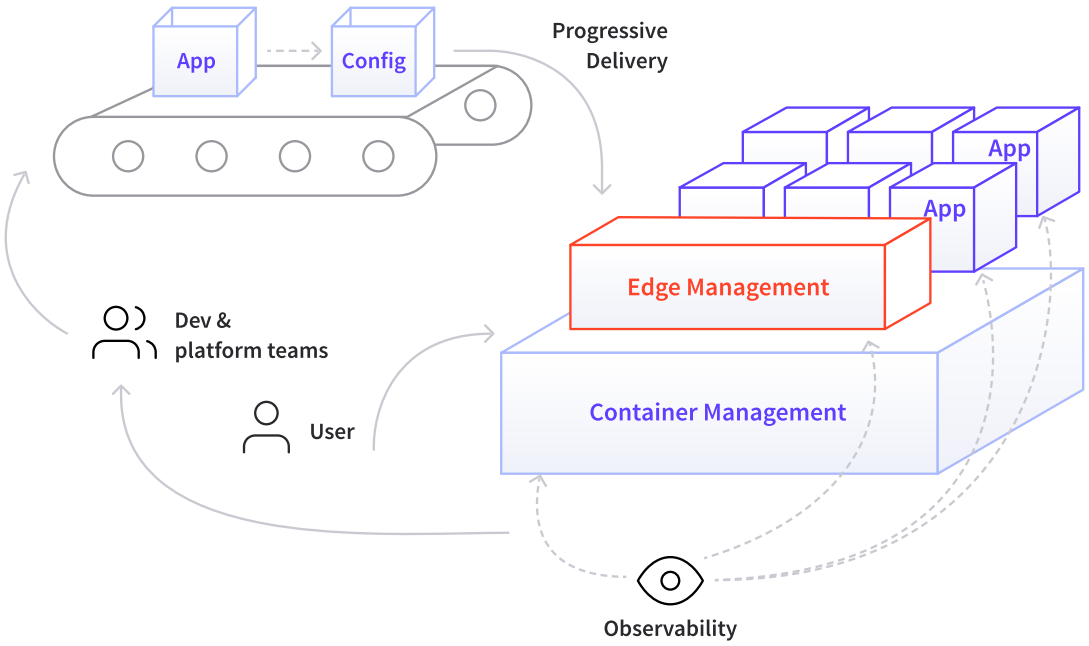
Bringing together all of the requirements discussed so far results in four core capabilities that a cloud native platform must provide: container management, progressive delivery, edge management, and observability management.

Container Management

This is the ability to manage and run container-based applications at scale and on a variety of infrastructures. Developers should be able to perform these operations in a self-service manner that can easily be automated and observed. This capability must also allow the platform team to set policies around access, control, and auditability.

Observability

This capability should support the collection and analysis of end user and application feedback directly by developers and the platform team. This allows product teams to iterate against business goals and KPIs, and supports the platform team in observing and managing infrastructure and ensuring their service level objectives (SLOs) are met.



Avoiding Platform Antipatterns

At first glance, providing a platform that provides all of the four capabilities may appear relatively simple. However, there are a number of platf.orm antipatterns which have been discovered over the recent history of software development. Whether an organisation buys a cloud native platform or builds this one sprint at a time, there are a number of common mistakes that must be avoided

Centralized Design and Ownership: One Size Doesn't Fit All

Many early attempts at creating a platform within an organization were driven by a single operations team without collaboration with the development teams. The ownership of these platforms tended to be centralised, the design work done upfront, and the resulting platforms were typically monolithic in operation. In order to use the platform, developers had to communicate with the operations team via tickets to get anything done, e.g. deploy artifact X, open port Y, and enable route Z.

These platforms often only supported limited "core" use cases that were identified as part of the upfront design process. Problems often emerged when developers wanted to design and deploy more modular systems, or implement new communication protocols. For example, as developers embraced the microservices architectural style, they frequently exposed more APIs at the edge of the system. They also adopted multiple protocols for applications: REST-like interactions for resource manipulations operations, WebSockets for streaming data, and gRPC for low-latency RPC calls.

As developers moved away from the one-size-fits-all specifications, this increasingly meant raising more and more tickets for the operations team. The combination of the high cost of handoffs and the frequently inadequate (or slowing changing) platform meant that developers could not release software at the speed that the business required. Often this led to development teams taking matters into their own hands.

Fragmented Platform Implementation

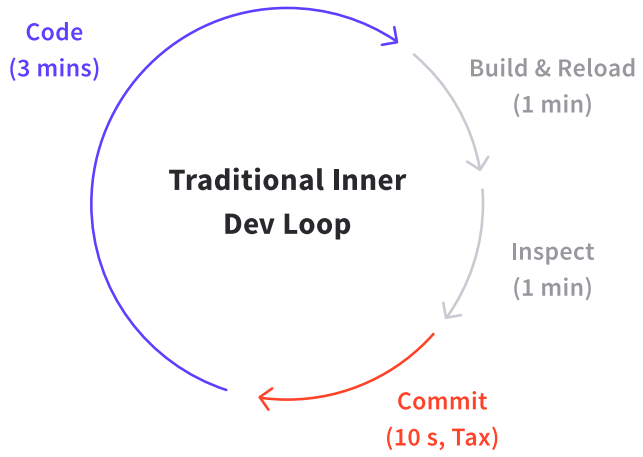
The failure of the centralized platform design and ownership model led to a swing in the opposite direction. Independent service development teams began building their own micro-platforms. This often manifested itself with coarse-grained access points being exposed at the edge of the system that forwarded all end-user requests to one of several development team-managed reverse proxies or API gateways. Developers had full control over these endpoints. However, inter-team collaboration was often low. Individual services were often configured to use third-party authentication and authorization services, leading to authentication sprawl. The capability to support availability and reliability goals were implemented in an ad hoc manner via language-specific libraries.

This failure mode led to lots of teams reinventing the (subtly different) wheel. Even popular open source libraries, such as Netflix's Hystrix circuit-breaker, were reimplemented subtly differently across various language platforms. The fragmentation meant that it was difficult to ensure consistency of availability, reliability, and security. And because all of these features were baked-into each application, each team had to use a different workflow to deploy, test, and observe their applications. This lack of common developer experience often caused additional challenges.

Slow Development Loops: Less Time Coding, More Time Toiling

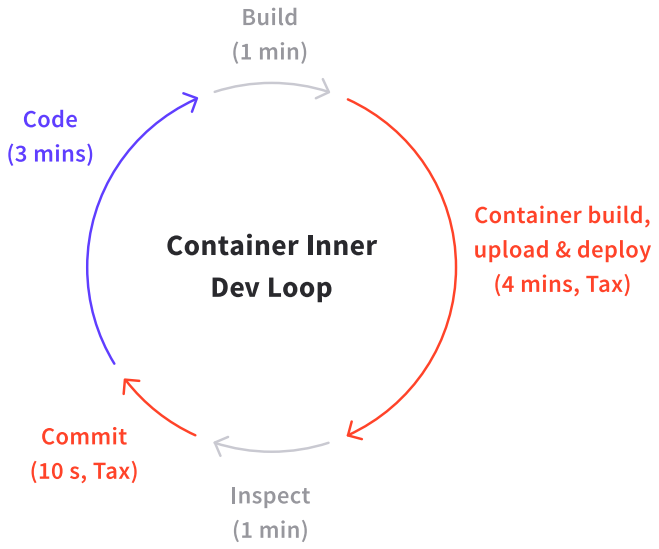
Cloud native technologies also fundamentally altered the developer experience. Not only are engineers now expected to design and build distributed service-based applications, but their entire development loop has been disrupted. No longer can developers rely on monolithic application development best practices, such as checking out the entire codebase and coding locally with a rapid "live-reload" inner developer loop. They now have to manage external dependencies, build containers, and implement orchestration configuration (e.g. Kubernetes YAML). This may appear trivial at first glance, but this has a large impact on development time.

If a typical developer codes for 360 minutes (6 hours) a day, with a traditional local iterative development loop of 5 minutes -- 3 coding, 1 building i.e. compiling/deploying/reloading, 1 testing inspecting, and 10-20 seconds for committing code -- they can expect to make ~70 iterations of their code per day. Any one of these iterations could be a release candidate. The only "developer tax" being paid here is for the commit process, which is negligible.



If the build time is incremented to 5 minutes -- not atypical with a standard container build, registry upload, and deploy -- then the number of possible development iterations per day drops to ~40. At the extreme that's a 40%

decrease in potential new features being released. This new container build step is a hidden tax, which is quite expensive.



Many development teams began using custom proxies to either automatically and continually sync their local development code base with a remote surrogate (enabling “live reload” in a remote cluster), or route all remote service traffic to their local services for testing. The former approach had limited value for compiled languages, and the latter often did not support collaboration within teams where multiple users want to work on the same services.

In addition to the challenges with the inner development loop, the changing outer development loop also caused issues. Over the past 20 years, end users and customers have become more demanding, but also less sure of their requirements. Pioneered by disruptive organisations like Netflix, Spotify, and Google, this has resulted in software delivery teams needing to be capable of rapidly delivering experiments into production. Unit, integration, and component testing is still vitally important, but modern application platforms must also support the incremental release of functionality and applications to end users in order to allow testing in production.

The traditional outer development loop for software engineers of code merge, code review, build artifact, test execution, and deploy has now evolved. A typical modern outer loop now consists of code merge, automated code review, build artifact and container, test execution, deployment, controlled (canary) release, and observation of results. If a developer doesn’t have access to self-service configuration of the release then the time taken for this outer loop increases by at least an order of magnitude e.g. 1 minute to deploy an updated canary release routing configuration versus 10 minutes to raise a ticket for a route to be modified via the platform team.

Next Steps

With the motivations for enabling full cycle development presented, the four capabilities of a cloud native platform defined, and a number of antipatterns highlighted, what is next? The answer is designing and building an effective platform to support the teams and workflow. For an organisation that is moving to cloud in 2020, this platform is likely to be based on a technology that has fast become the de facto vendor-agnostic computing abstraction: Kubernetes.

Building a Kubernetes-Based Platform

Practically every cloud vendor or private cloud solution supports the deployment and operation of the Kubernetes container orchestration framework. Since the initial release of Kubernetes by Google in 2014, a large community has formed around the framework, often facilitated by the organisation that is now the steward of the project, the Cloud Native Computing Foundation (CNCF).

Kubernetes has been widely adopted as a container manager, and has been running in production across a variety of organisations for several years. As such, it provides a solid foundation on which to support the other three capabilities of a cloud native platform: progressive delivery, edge management, and observability. These capabilities can be provided, respectively, with the following technologies: continuous delivery pipelines, an edge stack, and an observability stack.

Starting with Kubernetes, lets explore how each of these technologies integrates to provide the core capabilities of a cloud platform.

Kubernetes

Following from the early success of Docker, containers have become the standard unit ("artifact") of cloud deployment. Applications written in any language can be built, packaged, and "hermetically" sealed within a container image. These containers can then be deployed and run wherever the container image format is supported. This a cloud native implementation of the popular software development concept of "write once, run anywhere", except now the writing of code has been superseded by the building and packaging of this code.

The rise in popularity of containers can be explained by three factors: containers require less resources to run than VMs (with the tradeoff of a shared underlying operating system kernel); the Dockerfile manifest format provided a great abstraction for developers to define "just enough" build and deploy configuration; and Docker pioneered an easy method for developers to assemble applications in a self-service manner (docker build) and enabled the easy sharing and search of containerised applications via shared registries and the public Docker Hub.

Containers themselves, although a powerful abstraction, do not manage operational concerns, such as restarting and rescheduling when the underlying hardware fails. For this, a container orchestration framework is required. Something like Kubernetes.

Control Loops and Shared Abstractions

Kubernetes enables development teams to work in a self-service manner in relation to the operational aspects of running containers. For example, defining liveness and readiness probes of the application, and specifying runtime

resource requirements, such as CPU and memory. This configuration is then parsed by the control loop within the Kubernetes framework, which makes every effort to ensure that the developer's specifications match the actual state of the cluster. Operations teams can also define global access and deployment policies, using role-based access control (RBAC) and admission webhooks. This helps to limit access and guide development teams to best practices when deploying applications.

In addition to providing a container runtime and orchestration framework, Kubernetes allows both developers and the platform team to interact, share, and collaborate using a standardised workflow and toolset. It does this via several core abstractions: the container as the unit of deployment, the pod as the component of runtime configuration (combining containers, and defining deployment, restart, and retry policies), and the service as the high-level, business-focused components of an application.

Kubernetes-as-a-Service or Self-Hosted

Kubernetes itself is somewhat of a complicated framework to deploy, operate, and maintain. Therefore, a core decision when adopting this framework is whether to use a hosted offering, such as Google GKE, Amazon EKS, or Azure AKS, or whether to self-manage this using administrative tooling like kops and kubeadm.

A second important decision is what distribution ("distro") of Kubernetes to use. The default open source Kubernetes upstream distro provides all of the core functionality. It will come as no surprise that cloud vendors often augment their distros to enable easier integration with their surrounding ecosystem. Other platform vendors, such as Red Hat, Rancher, or Pivotal offer distros that run effectively across many cloud platforms, and they also include various enhancements. Typically the additional functionality is concentrated on supporting the enterprise use cases, with a focus on security, homogenized workflows, and providing comprehensive user interfaces (UIs) and administrator dashboards.

The Kubenetes documentation provides additional information to assist with these choices.

Avoiding Platform Antipatterns

The core development abstractions provided by Kubernetes -- containers, pods, and services -- facilitate collaboration across development and operations teams, and help to prevent the siloed ownership. These abstractions also reduce the likelihood that developers need to take things into their own hands and start building "micro-platforms" within the system itself.

Kubernetes can also be deployed locally, which in the early stages of adoption can help with addressing the challenge of slower or limited developer feedback. As an organisation's use of Kubernetes grows, they can leverage a host of tooling to address the local-to-remote development challenge. Tools like Telepresence, Skaffold, Tilt, and Garden all provide mechanisms for developers to tighten the feedback loop from coding (possibly against remote dependencies), building, and verifying.

Continuous Delivery Pipelines

The primary motivation of continuous delivery is to deliver any and all application changes -- including experiments, new features, configuration, and bug fixes -- into production as rapidly and as safely as the organisation requires. This approach is predicated on the idea that being able to iterate fast provides a competitive advantage. Application deployments should be routine and drama free events, initiated on-demand and safely by product-focused development teams, and the organisation should be able to continuously innovate and make changes in a sustainable way.

Improving the Feedback Loops

Progressive delivery extends the approach of continuous delivery by aiming to improve the feedback loop for developers. Taking advantage of cloud native traffic control mechanisms and improved observability tooling allows developers to more easily run controlled experiments in production, see the results in near real time via dashboards, and take corrective action if required.

The successful implementation of both continuous delivery and progressive delivery depends on developers having the ability to define, modify, and maintain pipelines that codify all of the build, quality, and security assertions. The core decisions to make when adopting a cloud native approach to this are primarily based on two factors: how much existing continuous delivery infrastructure has organisation has; and the level of verification required for application artifacts.

Evolving an Organisation's Approach to Continuous Delivery

Organisations with a large investment into existing continuous delivery tooling will typically be reluctant to move away from this. Jenkins can be found within many enterprise environments, and operations teams have often invested a lot of time and energy in understanding this tool. Although this pre-cloud build tool can be adapted to meet new requirements, there is also a complementary cloud native project, Jenkins X, where support for the core concepts of progressive delivery are built into the tool. The extendability of Jenkins has, for better or worse, enabled the creation of many plugins. There are plugins for executing code quality analysis, security scans, and automated test execution. There are also extensive integrations with quality analysis tooling like SonarQube, Veracode, and Fortify.

Organisations with limited existing investment in continuous delivery pipeline tooling often choose to use cloud native hosted options, such as Harness, CircleCI, or Travis. These tools focus on providing easy self-service configuration and execution for developers. However, some are not as extensible as tooling that is deployed and managed on-premises, and the provided functionality is often focused on building artifacts rather than deploying them. Operations teams also typically have less visibility into the pipeline. For this reason, many teams separate build and deployment automation, and use continuous delivery platforms such as Spinnaker to orchestrate these actions.

Avoiding Platform Antipatterns

Continuous delivery pipeline infrastructure is often the bridge between development and operations. This can be

used to address the traditional issues of siloed ownership of code and the runtime. For example, platform teams can work with development teams to provide code buildpacks and templates, which can mitigate the impact of a "one-size fits all" approach, and also remove the temptation for developers to build their own solutions.

Continuous delivery pipelines are also critical for improving developer feedback. A fast pipeline that deploys applications ready for testing in a production-like environment will reduce the need for context switching. Deployment templates and foundational configuration can also be added to the pipeline in order to bake-in common observability requirements to all applications, for example, logging collection, or metric emitters. This can greatly help developers gain and understanding of production systems, and assist with debugging issues, without needing to rely on the operations team to provide access.

The Edge Stack

The primary goals associated with operating an effective datacenter edge, which in a modern cloud platform configuration is often the Kubernetes cluster edge, are threefold:

- Enabling the controlled release of applications and new functionality;
- Supporting the configuration of cross-functional edge requirements, such as security (authentication, transport level security, and DDoS protection) and reliability (rate limiting, circuit breaking, and timeouts);
- Supporting developer onboarding and use of associated APIs.

Separate Release from Deployment

The current best practice within a cloud native software delivery is to separate deployment from release. The continuous delivery pipeline handles the build, verification, and deployment of an application. A "release" occurs when feature change with an intended business impact is made available to end users. Using techniques such as dark launches and canary releases, changes can be deployed to production environments more frequently without the risk of large-scale negative user impact.

More-frequent, iterative deployments reduce the risk associated with change, while developers and business stakeholders retain control over when features are released to end users.

Scaling Edge Operations with Self-Service

Within a cloud-native system that is being built with microservices, the challenges of scaling edge operations and supporting multiple architectures must be implemented effectively. Configuring the edge must be self-service for both developers iterating rapidly within the domain of a single service or API, and also the platform team that are working at a global system scale. The edge technology stack must offer comprehensive support for a range of protocols, architectural styles, and interaction models, that are commonly seen within a polyglot language stack.

Avoiding Platform Antipatterns

Gone are the days where every API exposed within a system was SOAP- or REST-based. With a range of protocols and standards like WebSockets, gRPC, and CloudEvents, there can no longer be a “one size fits all” approach to the edge. It is now table-stakes for all parts of the edge stack to support multiple protocols natively.

The edge of a Kubernetes system is another key collaboration point for developers and the platform teams. Platform teams want to reduce fragmentation by centralizing core functionality such as authentication and authorization. Developers want to avoid having to raise tickets to configure and release services as part of their normal workflow, as this only adds friction and reduces cycle time for delivery of functionality to end users.

The Observability Stack

The concept of “observability” originated from mathematical control theory, and is a measure of how well internal states of a system can be inferred from knowledge of its external outputs. Modern interpretations of software observability have developed almost in lockstep with the rise of cloud native systems; observability in this context is focused on the ability to infer what is occurring within a software system using approaches such as monitoring, logging, and tracing.

As popularised in the Google SRE book, given that a service level indicator (SLI) is an indicator of some aspect of "health" that a system’s consumers would care about (which is often specified via an SLO), there are two fundamental goals with observability:

- Gradually improving an SLI (potentially optimising this over days, weeks, months)
- Rapidly restoring an SLI (reacting immediately, in response to an incident)

Following on from this, there are two fundamental activities that an observability stack must provide: detection, which is the ability to measure SLIs precisely; and refinement, which is the ability to reduce the search space for plausible explanations of an issue.

Understandability, Auditability, and Debuggability

Closely related to the goals of improving or restoring SLIs, are the additional motivations of supporting observability within a software system: understandability, auditability, and debuggability. As software systems have become pervasive and mission critical throughout society, the need to understand and audit them has increased dramatically. People are slow to trust something that they cannot understand. And if a system is believed to have acted incorrectly, or someone claims it has, then the ability to look back through the statement of audit and prove or disprove this is invaluable.

The adoption of cloud native technologies and architectures has unfortunately made implementing observability more challenging. Understanding a distributed system is inherently more difficult when operating at scale. And

existing tooling does not support the effective debugging of a highly modular system communicating over unreliable networks. A new approach to creating a cloud native observability stack is required.

Three Pillars of Observability: One Solution

Thought leaders in this modern observability space, such as Cindy Sridharan, Charity Majors, and Ben Sigelman, have written several great articles that present the “three pillars” of cloud native observability as monitoring, logging, and distributed tracing. However, they have also cautioned that these pillars should not be treated in isolation. Instead a holistic solution should be sought.

Monitoring in the cloud native space is typically implemented via the CNCF-hosted Prometheus application or a similar commercial offering. Metrics are often emitted via containerised applications using the statsd protocol, or a language native Prometheus library. The use of metrics provides a good insight an application and the platform of a snapshot in time, and can also be used to trigger alert

Logging is commonly emitted as events from containerised applications via a common interface, such as STDOUT, or via a logging SDK included within the application. Popular tooling includes the Elasticsearch, Logstash, and Kibana (ELK) stack. Fluentd, a CNCF-hosted project, is often used in place of Logstash. Logging is valuable when retroactively attempting to understand what happened within an application, and can also be used for auditing purposes.

Distributed tracing is commonly implemented using the OpenZipkin or the CNCF-hosted Jaegar tooling, or a commercial equivalent. Tracing is effectively a form of event-based logging that contains some form of correlation identifier that can be used to stitch together events from multiple services that are related to a single end user’s request. This provides end-to-end insight for requests, and can be used to identify problematic services within the system (for example, latent services) or understand how a request traveled through the system in order to satisfy the associated user requirement.

Many of the principles from the data mesh paradigm apply to the topic observability. Platform engineers must provide a series of observability data access tools and APIs for developers to consume in a self-service manner.

Avoiding Platform Antipatterns

With cloud native observability there is no “one-size fits all” approach. Although the emitting and collection of observability data should be standardised to avoid platform fragmentation, the ability to self-serve when defining and analysing application- and service-specific metrics is vital for developers to be able to track health or fix something when the inevitable failures occur. A common antipattern seen within organisations is the requirement to raise a ticket in order to track specific metrics or incorporate these into a dashboard. This completely goes against the principle of enabling a fast development loop, which is especially important during the launch of new functionality or when a production incident is occuring.

Summary and Conclusion

As this report has shown, the emergence of “cloud native” technologies and practices, such as microservices, cloud computing, and DevOps, has enabled innovative organisations to respond and adapt to market changes more rapidly than their competitors. However, for better or worse, the traditional software development life cycle (SDLC) has been disrupted by the arrival of the cloud.

Within the traditional approach to the SDLC, engineers were specialised and often worked in silos. The cloud native SDLC is very different. The use of microservices has allowed “full cycle” product-focused development teams to work independently. This reduced coordination time and cost, and also allowed the selection of the best architectures and protocols for their individual use cases.

Embracing the notion of full cycle development does not remove the need for specialist operations, sysadmin, or platform teams. This does, however, require upskilling within both development and operations teams

Cloud computing and container orchestration frameworks provide an excellent foundation for deploying and running modern applications. However, in order for these technologies to support the move to full cycle development, there are several capabilities that must be provided: container management, progress delivery, edge management, and observability management.

Kubernetes has been widely adopted, and has been running in production across a variety of organisations for several years. As such, it provides a solid foundation on which to support the other three capabilities of a cloud native platform that enables full cycle development. These capabilities can be provided, respectively, with the following technologies: continuous delivery pipelines, an edge stack, and an observability stack.

Investing in these technologies and the associated best practice workflows will speed an organisation’s journey to seeing the benefits from embracing the cloud native and full cycle development principles.

About Ambassador Edge Stack

Ambassador is a self-service, comprehensive edge stack built on Envoy Proxy and Kubernetes by the team at Datawire (datawire.io). With the Ambassador Edge Stack, organizations can ship software with greater agility. Ambassador gives platform engineers and operators a self-service edge proxy, reducing toil. At the same time, Ambassador gives application developers fine-grained, self-service control over their edge proxy needs. The Ambassador Edge Stack provides state-of-the-art functionality for observability, availability, security, traffic management, developer onboarding, and edge policy management.

For more information, visit our website (www.getambassador.io), check out our blog (blog.getambassador.io), or follow us on Twitter (@getambassadorio).